relax

Version 1.2.1

A program for NMR relaxation data analysis

Edward d'Auvergne

January 20, 2006

Acknowledgements

Firstly, I am grateful and would like to thank Paul Gooley for giving me the opportunity to create this program as part of my PhD. The duration of development has been considerable and I thank Paul for allowing me the time to learn Python and to implement many data analysis concepts into relax.

I would also like to thank Andrew James Perry for innumerable, thoughtful discussions on both the science and programming aspects behind relax. Many of the important design features of relax originated as ideas of his and much of the usability and core layout came from these discussions. Examples include the prompt based interface with the help system and tab completion, the automatic generation of the documentation on the user functions by parsing their docstrings, details of the interaction with and control of other programs, and the powerful Python scripting. Without Andrew, this program would not be what it is today.

Finally I would like to thank Haydyn D.T. Mertens and James D. Swarbrick for many discussions and feedback.

Contents

List of Figures

Abbreviations

AIC Akaike's Information Criteria

AICc small sample size corrected AIC

BIC Bayesian Information Criteria

$C(\tau)$ correlation function

χ^2 chi-squared function

CSA chemical shift anisotropy

$\mathfrak{D}$ the set of diffusion tensor parameters

$\mathfrak{D}_{\parallel}$ the eigenvalue of the spheroid diffusion tensor corresponding to the unique axis of the tensor

$\mathfrak{D}_{\perp}$ the eigenvalue of the spheroid diffusion tensor corresponding to the two axes perpendicular to the unique axis

$\mathfrak{D}_a$ the anisotropic component of the Brownian rotational diffusion tensor

$\mathfrak{D}_{iso}$ the isotropic component of the Brownian rotational diffusion tensor

$\mathfrak{D}_r$ the rhombic component of the Brownian rotational diffusion tensor

$\mathfrak{D}_{ratio}$ the ratio of $\mathfrak{D}_{\parallel}$ to $\mathfrak{D}_{\perp}$

$\mathfrak{D}_x$ the eigenvalue of the Brownian rotational diffusion tensor in which the corresponding eigenvector defines the x-axis of the tensor

$\mathfrak{D}_y$ the eigenvalue of the Brownian rotational diffusion tensor in which the corresponding eigenvector defines the y-axis of the tensor

$\mathfrak{D}_z$ the eigenvalue of the Brownian rotational diffusion tensor in which the corresponding eigenvector defines the z-axis of the tensor

ϵ_i elimination value

$J(\omega)$ spectral density function

NOE nuclear Overhauser effect

pdf probability distribution function

r bond length

$\mathbf{R}_1$ spin-lattice relaxation rate

$\mathbf{R}_2$ spin-spin relaxation rate

R_{ex} chemical exchange relaxation rate

S^2, S_f^2, **and** S_s^2 model-free generalised order parameters

τ_e, τ_f, **and** τ_s model-free effective internal correlation times

τ_m global rotational correlation time

Chapter 1

Installation instructions

1.1 Dependencies

The following packages need to be installed before using relax:

Python: Version 2.2 or higher .

Numeric: Version 21 or higher .

ScientificPython: Version 2.2 or higher .

Optik: Version 1.4 or higher. This is only needed if running python 2.2 .

Older versions of these packages may work, use them at your own risk.

1.2 Installation

1.2.1 Linux precompiled distribution

To install the program relax on a GNU/Linux system, download the precompiled distribution labelled `relax-x.x.x.Linux.i386.tar.bz2`. This tar file has had the C code precompiled into shared objects files `*.so` which are loaded directly into relax. Hence compilation is not necessary. Untar and decompress the file using the command

```
$ tar jxvf relax-x.x.x.Linux.i386.tar.bz2
```

Then in the base directory `relax`, switch user to root and type the command

```
$ make install
```

This will create a directory in `/usr/local/` called `relax`, copy all the untarred files into this directory, create a symbolic link in `/usr/local/bin` to the file `/usr/local/relax/relax`, and then finally run relax to create the byte-compiled Python `*.pyc` files to speed up the start time of relax.

To change the installation path, modify the file `Makefile` and change the variable `INSTALL_PATH` to point to the desired location.

1.2.2 Source distribution

The source distribution should be named `relax-x.x.x.src.tar.bz2`. The make utility and a C compiler are required for compilation of the C code into shared objects which are loaded as modules into relax. To build these modules, type

```
$ make
```

Then to install the program, type

```
$ make install
```

The default installation path is currently `/usr/local/`. To change this, edit the `Makefile` and modify the variable `INSTALL_PATH`.

1.2.3 Running a non-compiled version

Compilation of the C code is not essential for running relax, however, certain features of the program will be disabled. One example is relaxation curve fitting. This approach may be necessary for systems, such as MS Windows, where C compilers are not readily available (although the cygwin environment may provide the tools required for compilation).

To run relax without compilation, install the dependencies detailed above, download the source distribution which should be named `relax-x.x.x.src.tar.bz2`, extract the files, and then run the file called `relax` in the base directory.

1.3 Optional programs

The following is a list of programs which can be used by relax, although they are not essential for normal use.

1.3.1 Grace

Grace is a program for plotting two dimensional data sets in a professional looking manner. It is used to visualise parameter values. It can be downloaded from `http://plasma-gate.weizmann.ac.il/Grace/`.

1.3.2 OpenDX

Version 4.1.3 or compatible. OpenDX is used for viewing the output of the space mapping function, and is executed by passing the command `dx` to the command line with various options. The program is designed for visualising multidimensional data and can be found at `http://www.opendx.org/`.

1.3.3 Molmol

Molmol is used for viewing the PDB structures loaded into the program and to display parameter values mapped onto the structure.

1.3.4 Dasha

Dasha is a program used for model-free analysis of NMR relaxation data. It can be used as an optimisation engine to replace the minimisation algorithms implemented within relax.

1.3.5 Modelfree4

Art Palmer's Modelfree4 program is also designed for model-free analysis and can be used as an optimisation engine to replace relax's high precision minimisation algorithms.

Chapter 2

How to use relax

2.1 The prompt

The primary interface of relax is the prompt. After typing `'relax'` within a terminal, you will be presented with

```
relax>
```

This is the Python prompt which has been tailored specifically for relax. You will hence have full access, if necessary, to the power of the Python programing language to manipulate your data. You can for instance type

```
relax> print "Hello World"
```

the result being

```
relax> print "Hello World"
Hello World
relax>
```

Or, using relax as a calculator

```
relax> (1.0 + (2 * 3))/10
0.69999999999999996
relax>
```

2.2 Python

relax has been designed such that knowledge about Python is not required to be able to fully use the program. A few basics, though, will aid in understanding relax.

A number of simple programming axioms includes that of strings , integers , floating point numbers , and lists . A string is text and within Python (as well as relax) this is delimited by either single or double quotes. An integer is a number with no decimal point while

a float is a number with a decimal point. A list in Python (called an array in other languages) is a list of anything separated by commas and delimited by square brackets, an example is [0, 1, 2, 'a', 1.2143235].

Probably the most important detail is that functions in Python require brackets around their arguments. For example

```
relax> minimise()
```

will commence minimisation however

```
relax> minimise
```

will do nothing.

The arguments to a function are simply a comma separated list within the brackets of the function. For example to save the program's current state, type

```
relax> state.save('save', force=1)
```

Two types of arguments exist in Python , standard arguments and keyword arguments. The majority of arguments you will encounter within relax are keyword arguments however you may, in rare cases, encounter a non-keyword argument. For these standard arguments, just type the values in, although they must be in the correct order. Keyword arguments consist of two parts, the key and the value. For example the key may be `file` while the value you would like to supply is `'R1.out'`. Various methods exist for supplying this argument. Firstly you could simply type `'R1.out'` into the correct position in the argument list. Secondly you can type `file='R1.out'`. The power of this second option is that argument order is unimportant. Therefore if you would like to change the default value of the very last argument, you don't have to supply values for all other arguments. The only catch is that standard arguments must come before the keyword arguments.

2.3 User functions

For standard data analysis, a large number of specially tailored functions called 'user functions' have been implemented. These are accessible from the relax prompt by simply typing the name of the function. An example is `'help()'` . An alphabetical listing of all accessible user functions together with full descriptions is presented later within this manual.

A few special objects which are available within the prompt are not actually functions. These objects do not require brackets at their end for them to function. For example to exit relax type

```
relax> exit
```

Another special object is that of the function class . This object is simply a container which holds a number of user functions. You can access the user function within the class by typing the name of the class, then a dot `'.'`, followed by the name of the user function. An example is the user function for reading relaxation data out of a file and loading the data into relax. The function is called `'read'` while the class is called `'relax_data'`. To execute the function, type something like

```
relax> relax_data.read(name, 'R1', '600', 600.0 * 1e6, 'r1.600.out')
```

The relax prompt, on first usage, can be quite daunting. Two features exist, however, to increase the usability of the prompt – the help system and tab completion

2.4 The help system

For assistance in using a function, simply type

```
help(function)
```

In addition to functions, if

```
help(object)
```

is typed, the help for the python object is returned. This system is similar to the help function built into the python interpreter, which has been renamed to `help_python`, with the interactive component removed. For the interactive python help system, type

```
help_python()
```

2.5 Tab completion

Tab completion is implemented to prevent insanity as the function names can be quite long – a deliberate feature to improve usability. The behaviour of the tab completion is very similar to that of the bash prompt.

Not only is tab completion useful for preventing RSI, but it can also be used for finding out what functions are available. To begin with if you hit the [TAB] key without typing any text, all available functions will be listed (along with function classes and other python objects). This extends to the exploration of user functions within a function class . For example, to list the user functions within the function class `'model_free'`, type

```
relax> model_free.
```

The dot character at the end is essential. After hitting the [TAB] key, you should see something like

```
relax> model_free.
model_free.__class__
model_free.__doc__
model_free.__init__
model_free.__module__
model_free.__relax__
model_free.__relax_help__
model_free.copy
model_free.create_model
model_free.delete
model_free.remove_tm
model_free.select_model
```

```
relax> model_free.
```

All the objects beginning with an underscore are "hidden", they contain information about the function class and should be ignored. From the listing the user functions 'copy', 'create_model', 'delete', 'remove_tm', and 'select_model' contained within 'model_free' are all visible.

2.6 The 'run'

Within relax, the majority of operations are assigned to a special construct called a 'run'. For example, to load relaxation data into the program it must be committed to a pre-created 'run'. Within one instance of relax, multiple runs can be created and various operations performed in sequence on these runs. This is useful for operations such as model selection whereby the function 'model_selection' can operate on a number of runs corresponding to different models and then assign the results to a newly created run.

The flow of data through relax can be thought of as travelling through pipes – each pipe is synonymous with a run. User functions exist to transfer data between these pipes, while other functions combine data from multiple pipes into one or vice-verse. The simplest invocation of relax would be the creation of a single run and with the data being processed as it is passing through this pipe.

The primary method for creating a run is through the user function 'run.create'. For example

```
relax> run.create('m1', 'mf')
```

will create a run called 'm1'. The run is also associated with a type which, in this case, is model-free analysis. The following is a table of all the types which can be assigned to a run.

Run type	Description
'jw'	Reduced spectral density mapping
'mf'	Model-free data analysis
'noe'	Steady state NOE calculation
'relax_fit'	Relaxation curve fitting
'srls'	SRLS analysis

Currently only the NOE calculation, model-free analysis, and reduced spectral density mapping features of relax are implemented (if this documentation is out of date, then you may be able to do a lot more).

2.7 Scripting

What ever is done within the prompt is also accessible through scripting. Just type your commands into a text file and then at the terminal type

```
$ relax your_script
```

An example of a simple script which will minimise the model-free model ‘m4’ after loading six relaxation data sets is

```
# Create the run.
name = ‘m4’
run.create(name, ‘mf’)

# Nuclei type
nuclei(‘N’)

# Load the sequence.
sequence.read(name, ‘noe.500.out’)

# Load the relaxation data.
relax_data.read(name, ‘R1’, ‘600’, 600.0 * 1e6, ‘r1.600.out’)
relax_data.read(name, ‘R2’, ‘600’, 600.0 * 1e6, ‘r2.600.out’)
relax_data.read(name, ‘NOE’, ‘600’, 600.0 * 1e6, ‘noe.600.out’)
relax_data.read(name, ‘R1’, ‘500’, 500.0 * 1e6, ‘r1.500.out’)
relax_data.read(name, ‘R2’, ‘500’, 500.0 * 1e6, ‘r2.500.out’)
relax_data.read(name, ‘NOE’, ‘500’, 500.0 * 1e6, ‘noe.500.out’)

# Setup other values.
diffusion_tensor.set(name, (2e-8, 1.3, 60, 290), param_types=1, axial_type=‘prolate’,
fixed=1)
value.set(name, 1.02 * 1e-10, ‘bond_length’)
value.set(name, -160 * 1e-6, ‘csa’)

# Select a preset model-free model.
model_free.select_model(run=name, model=name)

# Grid search.
grid_search(name, inc=11)

# Minimise.
minimise(‘newton’, run=name)

# Finish.
results.write(run=name, file=‘results’, force=1)
state.save(‘save’, force=1)
```

Scripting is much more powerful than the prompt as advanced Python programming can be employed (see the file ‘full_analysis.py’ in the ‘sample_scripts’ directory for an example).

2.8 Sample scripts

A few sample scripts have been provided in the directory 'sample_scripts'. These can be used as a good starting point for using relax.

2.9 The GUI

relax has been designed primarily for scripting and, as such, no graphical user interface (GUI) currently exists. The internal structure of the program has been specifically designed so any type of control mechanism can be easily added, including a GUI, therefore in the future one may be written. A GUI will, however, detract from the power and flexibility inherent in the control by scripting.

2.10 Access to the internals of relax

For highly advanced Python scripting or control of relax, almost every part of relax has been designed in an object oriented fashion. If you would like to play with internals of the program, the entirety of relax is accessible within the object called `'self.relax'`. To access the raw objects within relax which contain the program data, all data is stored within the object called `'self.relax.data'`.

2.11 Usage of the name relax

The program relax is so relaxed that the first letter should always be in lower case!

Chapter 3

Data analysis

3.1 Introduction

This chapter aims to explain not only the steps involved in the data analysis of relaxation data but also how to use the program relax to achieve this. Although a work in progress, it covers how to calculate the NOE, how to optimise and find the R_1 and R_2 relaxation rates, and how to implement model-free analysis.

3.2 Calculating the NOE

The calculation of NOE values is a straight forward and quick procedure which involves two components, the calculation of the value itself and the calculation of the errors. To understand the steps involved, we will follow in detail the execution of a sample NOE calculation script.

The sample script

```
# Script for calculating NOEs.

# Create the run
name = ‘noe’
run.create(name, ‘noe’)

# Load the sequence from a PDB file.
pdb(name, ‘Ap4Aase_new_3.pdb’, load_seq=1)

# Load the reference spectrum and saturated spectrum peak intensities.
noe.read(name, file=‘ref.list’, spectrum_type=‘ref’)
noe.read(name, file=‘sat.list’, spectrum_type=‘sat’)

# Set the errors.
noe.error(name, error=3600, spectrum_type=‘ref’)
```

```
noe.error(name, error=3000, spectrum_type='sat')

# Individual residue errors.
noe.error(name, error=122000, spectrum_type='ref', res_num=114)
noe.error(name, error=8500, spectrum_type='sat', res_num=114)

# Unselect unresolved residues.
unselect.read(name, file='unresolved')

# Calculate the NOEs.
calc(name)

# Save the NOEs.
value.write(name, param='noe', file='noe.out', force=1)

# Create grace files.
grace.write(name, y_data_type='ref', file='ref.agr', force=1)
grace.write(name, y_data_type='sat', file='sat.agr', force=1)
grace.write(name, y_data_type='noe', file='noe.agr', force=1)

# View the grace files.
grace.view(file='ref.agr')
grace.view(file='sat.agr')
grace.view(file='noe.agr')

# Write the results.
results.write(name, file='results', dir=None, force=1)

# Save the program state.
state.save('save', force=1)
```

Initialisation of the run

Firstly to simplify referencing of the run name in the relevent functions, the name `'noe'` is assigned to to the object `name` by the command

```
name = 'noe'
```

Therefore instead of typing `'noe'` each time the run needs to be referenced, `name` can be used instead. The run is created by the command

```
run.create(name, 'noe')
```

This user function will then create a run which is named `'noe'`, the second argument setting the run type to that of calculating the NOE. Setting the run type is important so that the program knows which user functions are compatible with the run, for example the function `minimise()` is meaningless in this sample script as the NOE values are computed by direct calculation rather than through optimisation.

Loading the data

The first thing which need to be done prior to any residue specific command is to load the sequence. In this case, the command

```
pdb(name, 'Ap4Aase_new_3.pdb', load_seq=1)
```

will extract the sequence from the PDB file 'Ap4Aase_new_3.pdb'. The first argument specifies the run into which the sequence will be loaded, the second specifies the file name, while the third causes the function to extract the sequence rather than just load the PDB into relax. Although the PDB coordinates have been loaded into the program, the structure serves no purpose when calculating NOE values.

The next two commands

```
noe.read(name, file='ref.list', spectrum_type='ref')
noe.read(name, file='sat.list', spectrum_type='sat')
```

load the peak heights of the reference and saturated NOE experiments (although the volume could be used instead). The keyword argument `format` has not been specified, hence the default format of a Sparky peak list (saved after typing `'lt'`) is assumed. If the program XEasy was used to analyse the spectra, the argument `format='xeasy'` is necessary. The first column of the file should be the Sparky assignment string, while it is assumed that the 4^{th} column contains either the peak height.

Setting the errors

In this example, the errors where measured from the base plain noise. The Sparky RMSD function was used to estimate the maximal noise levels across the spectrum in regions containing no peaks. For the reference spectrum the RMSD was approximately 3600 while in the saturated spectrum the RMSD was 3000. These errors are set by the commands

```
noe.error(name, error=3600, spectrum_type='ref')
noe.error(name, error=3000, spectrum_type='sat')
```

For the residue G114, the noise levels are significantly increased compared to the rest of the protein as the peak is located close to the water signal. The higher errors for this residue are specified by the commands

```
noe.error(name, error=122000, spectrum_type='ref', res_num=114)
noe.error(name, error=8500, spectrum_type='sat', res_num=114)
```

Unresolved residues

As the peaks of certain residues overlaps to such an extent that the heights of both cannot be resolved, a simple text file was created called `unresolved` in which each line consists of a single residue number. By using the command

```
unselect.read(name, file='unresolved')
```

all residues in the file `unresolved` are excluded from the analysis.

The NOE

At this point, the NOE can be calculated. The user function

```
calc(name)
```

will calculate both the NOE and the errors. The NOE value will be calculated using the formula

$$NOE = \frac{I_{sat}}{I_{ref}}, \tag{3.1}$$

where I_{sat} is the intensity of the peak in the saturated spectrum while I_{ref} is that of the reference spectrum. The error is calculated by

$$\sigma_{NOE} = \sqrt{\frac{(\sigma_{sat} \cdot I_{ref})^2 + (\sigma_{ref} \cdot I_{sat})^2}{I_{ref}}}, \tag{3.2}$$

where σ_{sat} and σ_{ref} are the peak intensity errors in the saturated and reference spectra respectively. To create a file of the NOEs, the command

```
value.write(name, param='noe', file='noe.out', force=1)
```

will create a file called `noe.out` with the NOE values and errors. The force flag will cause any file with the same name to be overwritten. An example of the format of `noe.out` is

```
Num  Name  Value                 Error
1    GLY   None                  None
2    PRO   None                  None
3    LEU   None                  None
4    GLY   0.12479588727508535   0.020551827436105764
5    SER   0.42240815792914105   0.02016346825976852
6    MET   0.45281703194372114   0.026272719841642134
7    ASP   0.60727570079478255   0.032369427242382849
8    SER   0.63871921623680161   0.024695665815261791
9    PRO   None                  None
10   PRO   None                  None
11   GLU   None                  None
12   GLY   0.92927160307645906   0.059569089743604184
13   TYR   0.88832516377296256   0.044119641308479306
14   ARG   0.84945042565860407   0.060533543601110441
```

Viewing the results

Any two dimensional data set can be plotted in relax in conjunction with the program Grace (http://plasma-gate.weizmann.ac.il/Grace/). The program is also known as Xmgrace and was previously known as ACE/gr or Xmgr. The highly flexible relax user function `grace.write` is capable of producing 2D plots of any x-y data sets. The three commands

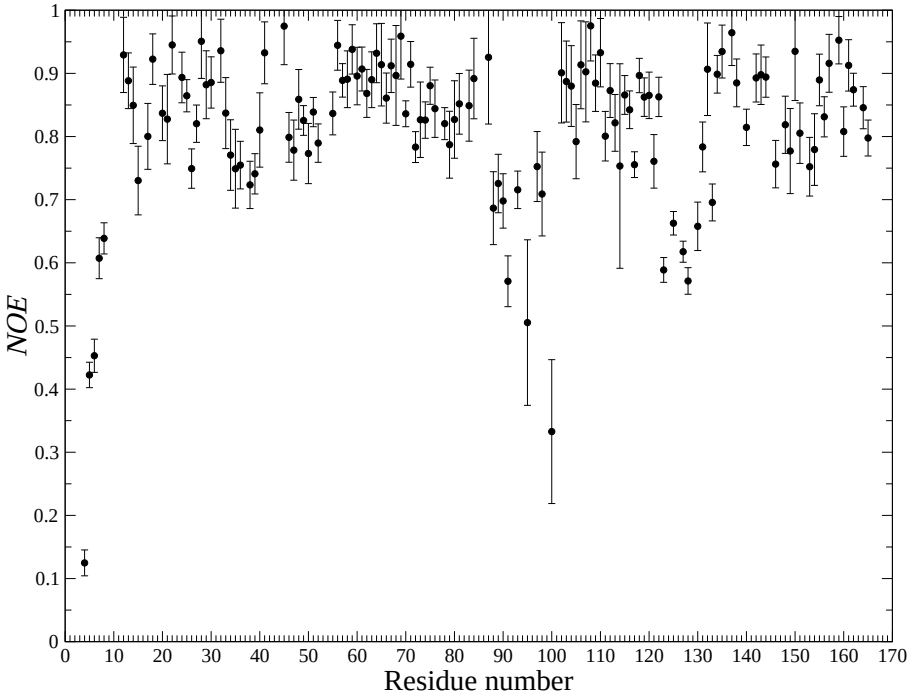

Figure 3.1: A Grace plot of the NOE value and error against the residue number. An example of the output of the user function `grace.write()`.

```
grace.write(name, y_data_type='ref', file='ref.agr', force=1)
grace.write(name, y_data_type='sat', file='sat.agr', force=1)
grace.write(name, y_data_type='noe', file='noe.agr', force=1)
```

create three separate plots of the peak intensity of the reference and saturated spectra as well as the NOE. The x-axis in all three defaults to the residue number. As the x and y-axes can be any parameter, the command

```
grace.write(name, x_data_type='ref', y_data_type='sat', file='ref_vs_sat.agr', force=1)
```

would create a plot of the reference verses the saturated intensity, with one point per residue. Returning to the sample script, three Grace data files are created, `ref.agr`, `sat.agr`, and `noe.agr` and placed in the default directory `./grace`. These can be visualised by opening the file within Grace, however relax will do that for you with the commands

```
grace.view(file='ref.agr')
grace.view(file='sat.agr')
grace.view(file='noe.agr')
```

An example of the output, after modifying the axes, is shown in figure 3.1.

3.3 The R_1 and R_2 relaxation rates - relaxation curve fitting

Relaxation curve fitting involves a number of steps including the loading of data, the calculation of both the average peak intensity across replicated spectra and the standard deviations of those peak intensities, selection of the experiment type, optimisation of the parameters of the fit, Monte Carlo simulations to find the parameter errors, and saving and viewing the results. To simplify the process, a sample script will be followed step by step as was done with the NOE calculation.

The sample script

```
# Script for relaxation curve fitting.

# Create the run.
name = 'rx'
run.create(name, 'relax_fit')

# Load the sequence from a PDB file.
pdb(name, 'Ap4Aase_new_3.pdb', load_seq=1)

# Load the peak intensities.
relax_fit.read(name, file='T2_ncyc1.list', relax_time=0.0176)
relax_fit.read(name, file='T2_ncyc1b.list', relax_time=0.0176)
relax_fit.read(name, file='T2_ncyc2.list', relax_time=0.0352)
relax_fit.read(name, file='T2_ncyc4.list', relax_time=0.0704)
relax_fit.read(name, file='T2_ncyc4b.list', relax_time=0.0704)
relax_fit.read(name, file='T2_ncyc6.list', relax_time=0.1056)
relax_fit.read(name, file='T2_ncyc9.list', relax_time=0.1584)
relax_fit.read(name, file='T2_ncyc9b.list', relax_time=0.1584)
relax_fit.read(name, file='T2_ncyc11.list', relax_time=0.1936)
relax_fit.read(name, file='T2_ncyc11b.list', relax_time=0.1936)

# Calculate the peak intensity averages and the standard deviation of all spectra.
relax_fit.mean_and_error(name)

# Unselect unresolved residues.
unselect.read(name, file='unresolved')

# Set the relaxation curve type.
relax_fit.select_model(name, 'exp')

# Grid search.
grid_search(name, inc=11)

# Minimise.
minimise('simplex', run=name, constraints=0)

# Monte Carlo simulations.
```

```
monte_carlo.setup(name, number=500)
monte_carlo.create_data(name)
monte_carlo.initial_values(name)
minimise('simplex', run=name, constraints=0)
monte_carlo.error_analysis(name)

# Save the relaxation rates.
value.write(name, param='rx', file='rx.out', force=1)

# Grace plots of the relaxation rate.
grace.write(name, y_data_type='rx', file='rx.agr', force=1)
grace.view(file='rx.agr')

# Save the program state.
state.save(file=name + '.save', force=1)
```

Initialisation of the run and loading of the data

The start of this sample script is very similar to that of the NOE calculation on page 12. The two commands

```
name = 'rx'
run.create(name, 'relax_fit')
```

initialise the run by setting the variable `name` to `'rx'` to be used in the calls to user functions and creating a run called `'rx'` which is to relaxation curve fitting by the argument `'relax_fit'`. The sequence is extracted from a PDB file using the same command as the NOE calculation script

```
pdb(name, 'Ap4Aase_new_3.pdb', load_seq=1)
```

To load the peak intensities into relax, use the user function `relax_fit.read`. Two important keyword arguments to this command are the file name and the relaxation time period of the experiment in seconds. It is assumed that the file format is that of a Sparky peak list. Using the format argument, this can be changed to XEasy text window output format. The following series of commands will load peak intensities from six different relaxation periods, four of which have been duplicated.

```
relax_fit.read(name, file='T2_ncyc1.list', relax_time=0.0176)
relax_fit.read(name, file='T2_ncyc1b.list', relax_time=0.0176)
relax_fit.read(name, file='T2_ncyc2.list', relax_time=0.0352)
relax_fit.read(name, file='T2_ncyc4.list', relax_time=0.0704)
relax_fit.read(name, file='T2_ncyc4b.list', relax_time=0.0704)
relax_fit.read(name, file='T2_ncyc6.list', relax_time=0.1056)
relax_fit.read(name, file='T2_ncyc9.list', relax_time=0.1584)
relax_fit.read(name, file='T2_ncyc9b.list', relax_time=0.1584)
relax_fit.read(name, file='T2_ncyc11.list', relax_time=0.1936)
```

```
relax_fit.read(name, file='T2_ncyc11b.list', relax_time=0.1936)
```

The rest of the setup

Once all the peak intensity data has been loaded, a few calculations are required prior to optimisation. Firstly the peak intensities for individual residues needs to be averaged across replicated spectra. The peak intensity errors also have to be calculated using the standard deviation formula. These two operations are executed by the user function

```
relax_fit.mean_and_error(name)
```

Any residues which cannot be resolved due to peak overlap were included in a file called `'unresolved'`. This file consists solely of one residue number per line. These residues are excluded from the analysis by the user function

```
unselect.read(name, file='unresolved')
```

Finally the experiment type is specified by the command

```
relax_fit.select_model(name, 'exp')
```

The argument `'exp'` sets the relaxation curve to a two parameter $\{R_x, I_0\}$ exponential which decays to zero. The formula of this function is

$$I(t) = I_0 e^{-R_x \cdot t}, \tag{3.3}$$

where $I(t)$ is the peak intensity at any time point t, I_0 is the initial intensity, and R_x is the relaxation rate, either being the R_1 or R_2. Changing the user function argument to `'inv'` will select the inversion recovery experiment. This curve consists of three paremeters $\{R_1, I_0, I_\infty\}$ and does not decay to zero. The formula is

$$I(t) = I_\infty - I_0 e^{-R_1 \cdot t}, \tag{3.4}$$

Optimisation

Now that everything has been setup, minimision can be used to optimise the parameter values. Firstly a grid search is applied to find a rough starting position for the subsequent optimisation algorithm. Eleven increments per dimension of the model (in this case the two dimensions $\{R_x, I_0\}$) is sufficient. The user function for executing the grid search is

```
grid_search(name, inc=11)
```

The next step is to select one of the minimisation algorithms to optimise the model parameters. Currently for relaxation curve fitting, only simplex minimisation is supported. This is because the relaxation curve fitting C module which implements the functions used during optimisation is incomplete. It is the chi-squared function which is used in optimisation. However, the chi-squared gradient (first partial derivatives) and chi-squared Hessian (second partial derivatives) are not yet present in the C modules and hence only optimisation algorithms which employ function calls are supported. Simplex minimisation is the only technique in relax which fits this critera. In addition, constraints cannot be used

as the constraint algorithm is dependent on gradient calls. Therefore the minimisation command for relaxation curve fitting is forced to be

```
minimise('simplex', run=name, constraints=0)
```

Error analysis

Only one technique adequately estimates parameter errors when the parameter values where found by optimisation – Monte Carlo simulations.

3.4 Model-free analysis

3.5 Reduced spectral density mapping

Chapter 4

Values, gradients, and Hessians

4.1 Introduction

A word of warning before reading this chapter, the topics covered here are quite advanced and are not necessary for understanding how to either use relax or to implement any of the data analysis techniques present within relax. The material of this chapter is intended as an in-depth explanation of the mathematics involved in the optimisation of the parameters of the model-free models. As such it contains the chi-squared equation, relaxation equations, spectral density functions, and diffusion tensor equations as well as their gradients (first partial derivatives) and Hessians (second partial derivatives).

4.1.1 Chi-squared function – $\chi^2(\theta)$

For the minimisation of models, a chain of calculations using different theories is required. At the highest level, the function which is actually minimised is the chi-squared function

$$\chi^2(\theta) = \sum_{i=1}^{n} \frac{(\mathrm{R}_i - \mathrm{R}_i(\theta))^2}{\sigma_i^2}, \tag{4.1}$$

where the index i is the summation index ranging over all the experimentally collected relaxation data, R_i, including the R_1, R_2, and NOE data at all field strengths belonging to the set data set R for an individual residue, a collection of residues, or the entire macromolecule, $\mathrm{R}_i(\theta)$ is the back calculated relaxation data belonging to the set $\mathrm{R}(\theta)$, θ is the model parameter vector which, when minimised, is denoted by the symbol $\hat{\theta}$, and σ_i are the experimental errors.

The significance of equation (4.1) is that the function returns a single value.

4.1.2 Relaxation equations – $\mathrm{R}_i(\theta)$

The chi-squared equation is itself dependent on the relaxation equations through the back calculated relaxation data $\mathrm{R}(\theta)$. These data are the spin-lattice , spin-spin , and cross-

relaxation rates of Abragam (1961) and are respectively

$$\mathrm{R}_1(\theta) = d\Big(J(\omega_H - \omega_X) + 3J(\omega_X) + 6J(\omega_H + \omega_X)\Big) + cJ(\omega_X), \tag{4.2}$$

$$\begin{aligned}\mathrm{R}_2(\theta) = \frac{d}{2}\Big(&4J(0) + J(\omega_H - \omega_X) + 3J(\omega_X) + 6J(\omega_H) \\ &+ 6J(\omega_H + \omega_X)\Big) + \frac{c}{6}\Big(4J(0) + 3J(\omega_X)\Big) + R_{ex},\end{aligned} \tag{4.3}$$

$$\sigma_{\mathrm{NOE}}(\theta) = d\Big(6J(\omega_H + \omega_X) - J(\omega_H - \omega_X)\Big), \tag{4.4}$$

where $J(\omega)$ is the power spectral density function, R_{ex} is the relaxation due to chemical exchange, and the dipolar and CSA constants are defined in SI units as

$$d = \frac{1}{4}\left(\frac{\mu_0}{4\pi}\right)^2 \frac{(\gamma_H \gamma_X \hbar)^2}{\langle r^6 \rangle}, \tag{4.5}$$

$$c = \frac{(\omega_H \Delta\sigma)^2}{3}, \tag{4.6}$$

where μ_0 is the permeability of free space, γ_H and γ_X are the gyromagnetic ratios of the H and X spins respectively, $\hbar$ is Plank's constant divided by 2π, r is the bond length, and $\Delta\sigma$ is the chemical shift anisotropy measured in ppm. The cross-relaxation rate σ_{NOE} is related to the steady state NOE by the equation

$$\mathrm{NOE}(\theta) = 1 + \frac{\gamma_H}{\gamma_X}\frac{\sigma_{\mathrm{NOE}}(\theta)}{\mathrm{R}_1(\theta)}. \tag{4.7}$$

4.1.3 Transformed relaxation equations – $\mathrm{R}'_i(\theta)$

Letting the relaxation equations, $\mathrm{R}_i(\theta)$ be the $\mathrm{R}_1(\theta)$, $\mathrm{R}_2(\theta)$, and $\mathrm{NOE}(\theta)$, an additional layer of abstraction can be used to simply the calculation of the gradients and Hessians. This involved decomposing the NOE equation into the cross relaxation rate constant $\sigma_{\mathrm{NOE}}(\theta)$ and the auto relaxation rate $\mathrm{R}_1(\theta)$. Taking equation (4.7), the relaxation equations are

$$\mathrm{R}'_1(\theta) = \mathrm{R}_1(\theta) \tag{4.8}$$

$$\mathrm{R}'_2(\theta) = \mathrm{R}_2(\theta) \tag{4.9}$$

$$\mathrm{NOE}(\theta) = 1 + \frac{\gamma_H}{\gamma_X}\frac{\sigma_{\mathrm{NOE}}(\theta)}{\mathrm{R}_1(\theta)}. \tag{4.10}$$

while the transformed relaxation equations are $\{\mathrm{R}'_1(\theta), \mathrm{R}'_2(\theta), \sigma_{\mathrm{NOE}}(\theta)\}$.

4.1.4 Spectral density functions – $J(\omega)$

The relaxation equations are themselves dependent on the calculation of the spectral density values $J(\omega)$. Within model-free analysis, these are modelled by the original model-free formula (Lipari and Szabo, 1982a,b)

$$J(\omega) = \frac{2}{5}\sum_{i=-k}^{k} c_i \cdot \tau_i \left(\frac{S^2}{1 + (\omega\tau_i)^2} + \frac{(1 - S^2)(\tau_e + \tau_i)\tau_e}{(\tau_e + \tau_i)^2 + (\omega\tau_e\tau_i)^2}\right), \tag{4.11}$$

where S^2 is the square of the Lipari and Szabo generalised order parameter and τ_e is the effective correlation time. The order parameter reflects the amplitude of the motion while the correlation time in an indication of the time scale of that motion. The theory was extended by Clore et al. (1990) by the modelling of two independent internal motions using the equation

$$J(\omega) = \frac{2}{5}\sum_{i=-k}^{k} c_i \cdot \tau_i \left(\frac{S^2}{1+(\omega\tau_i)^2} + \frac{(1-S_f^2)(\tau_f+\tau_i)\tau_f}{(\tau_f+\tau_i)^2+(\omega\tau_f\tau_i)^2} + \frac{(S_f^2-S^2)(\tau_s+\tau_i)\tau_s}{(\tau_s+\tau_i)^2+(\omega\tau_s\tau_i)^2} \right). \quad (4.12)$$

where S_f^2 and τ_f are the amplitude and timescale of the faster of the two motions while S_s^2 and τ_s are those of the slower motion. S_f^2 and S_s^2 are related by the formula $S^2 = S_f^2 \cdot S_s^2$.

4.1.5 Brownian diffusion

In equations (4.11) and (4.12), the generic Brownian diffusion NMR correlation function presented in **?** has been used. This function is

$$C(\tau) = \frac{1}{5}\sum_{i=-k}^{k} c_i \cdot e^{-\tau/\tau_i}, \quad (4.13)$$

where the summation index i ranges over the number of exponential terms within the correlation function. This equation is generic in that it describes the diffusion of an ellipsoid, a spheroid, and a sphere.

Ellipsoid

For the ellipsoid defined by the parameter set $\{\mathfrak{D}_{iso}, \mathfrak{D}_a, \mathfrak{D}_r, \alpha, \beta, \gamma\}$ the variable k is equal to two, therefore the index $i \in \{-2,-1,0,1,2\}$. The geometric parameters $\{\mathfrak{D}_{iso}, \mathfrak{D}_a, \mathfrak{D}_r\}$ are defined as

$$\mathfrak{D}_{iso} = \tfrac{1}{3}(\mathfrak{D}_x + \mathfrak{D}_y + \mathfrak{D}_z), \quad (4.14a)$$

$$\mathfrak{D}_a = \mathfrak{D}_z - \tfrac{1}{2}(\mathfrak{D}_x + \mathfrak{D}_y), \quad (4.14b)$$

$$\mathfrak{D}_r = \frac{\mathfrak{D}_y - \mathfrak{D}_x}{2\mathfrak{D}_a}, \quad (4.14c)$$

and are constrained by

$$0 < \mathfrak{D}_{iso} < \infty, \quad (4.15a)$$

$$0 \le \mathfrak{D}_a < \frac{\mathfrak{D}_{iso}}{\frac{1}{3}+\mathfrak{D}_r} \le 3\mathfrak{D}_{iso}, \quad (4.15b)$$

$$0 \le \mathfrak{D}_r \le 1. \quad (4.15c)$$

The orientational parameters $\{\alpha,\ \beta,\ \gamma\}$ are the Euler angles using the z-y-z rotation notation.

The five weights c_i are defined as

$$c_{-2} = \tfrac{1}{4}(d+e), \tag{4.16a}$$
$$c_{-1} = 3\delta_y^2\delta_z^2, \tag{4.16b}$$
$$c_0 = 3\delta_x^2\delta_z^2, \tag{4.16c}$$
$$c_1 = 3\delta_x^2\delta_y^2, \tag{4.16d}$$
$$c_2 = \tfrac{1}{4}(d-e), \tag{4.16e}$$

where

$$d = 3\left(\delta_x^4 + \delta_y^4 + \delta_z^4\right) - 1, \tag{4.17}$$
$$e = -\frac{1}{\mathfrak{R}}\left[(1+3\mathfrak{D}_r)\left(\delta_x^4 + 2\delta_y^2\delta_z^2\right) + (1-3\mathfrak{D}_r)\left(\delta_y^4 + 2\delta_x^2\delta_z^2\right) - 2\left(\delta_z^4 + 2\delta_x^2\delta_y^2\right)\right], \tag{4.18}$$

and where

$$\mathfrak{R} = \sqrt{1+3\mathfrak{D}_r^2}, \tag{4.19}$$

The five correlation times τ_i are

$$1/\tau_{-2} = 6\mathfrak{D}_{iso} - 2\mathfrak{D}_a\mathfrak{R}, \tag{4.20a}$$
$$1/\tau_{-1} = 6\mathfrak{D}_{iso} - \mathfrak{D}_a(1+3\mathfrak{D}_r), \tag{4.20b}$$
$$1/\tau_0 = 6\mathfrak{D}_{iso} - \mathfrak{D}_a(1-3\mathfrak{D}_r), \tag{4.20c}$$
$$1/\tau_1 = 6\mathfrak{D}_{iso} + 2\mathfrak{D}_a, \tag{4.20d}$$
$$1/\tau_2 = 6\mathfrak{D}_{iso} + 2\mathfrak{D}_a\mathfrak{R}. \tag{4.20e}$$

Spheroid

The variable k is equal to one in the case of the spheroid defined by the parameter set $\{\mathfrak{D}_{iso},\ \mathfrak{D}_a,\ \theta,\ \phi\}$, hence $i \in \{-1, 0, 1\}$. The geometric parameters $\{\mathfrak{D}_{iso},\ \mathfrak{D}_a\}$ are defined as

$$\mathfrak{D}_{iso} = \tfrac{1}{3}(\mathfrak{D}_\parallel + 2\mathfrak{D}_\perp), \tag{4.21a}$$
$$\mathfrak{D}_a = \mathfrak{D}_\parallel - \mathfrak{D}_\perp. \tag{4.21b}$$

and are constrained by

$$0 < \mathfrak{D}_{iso} < \infty, \tag{4.22a}$$
$$-\tfrac{3}{2}\mathfrak{D}_{iso} < \mathfrak{D}_a < 3\mathfrak{D}_{iso}, \tag{4.22b}$$

The orientational parameters $\{\theta,\ \phi\}$ are the spherical angles defining the orientation of the major axis of the diffusion frame within the lab frame.

The three weights c_i are

$$c_{-1} = \tfrac{1}{4}(3\delta_z^2 - 1)^2, \tag{4.23a}$$
$$c_0 = 3\delta_z^2(1 - \delta_z^2), \tag{4.23b}$$
$$c_1 = \tfrac{3}{4}(\delta_z^2 - 1)^2, \tag{4.23c}$$

The five correlation times τ_i are

$$1/\tau_{-1} = 6\mathfrak{D}_{iso} - 2\mathfrak{D}_a, \tag{4.24a}$$
$$1/\tau_0 = 6\mathfrak{D}_{iso} - \mathfrak{D}_a, \tag{4.24b}$$
$$1/\tau_1 = 6\mathfrak{D}_{iso} + 2\mathfrak{D}_a. \tag{4.24c}$$

Sphere

In the situation of a molecule diffusing as a sphere either described by the single parameter τ_m or $\mathfrak{D}_{iso}$, the variable k is equal to zero. Therefore $i \in \{0\}$. The single weight c_0 is equal to one, while the single correlation time τ_0 is equivalent to the global tumbling time τ_m given by

$$1/\tau_m = 6\mathfrak{D}_{iso}. \tag{4.25}$$

4.2 Minimisation concepts

4.2.1 The function value

At the simplest level, all minimisation techniques require at least a function which will supply a single value for different parameter values θ. For the modelling of NMR relaxation data, this is given by equation (4.1). For certain algorithms, such a Simplex minimisation , this single value suffices.

4.2.2 The gradient

The majority of minimisation algorithms, however, also require the gradient at the point of the given parameter values θ. The gradient is a vector of partial derivatives defined as

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial\theta_1} \\ \frac{\partial}{\partial\theta_2} \\ \vdots \\ \frac{\partial}{\partial\theta_n} \end{pmatrix} \tag{4.26}$$

where n is the total number of parameters in the model.

An example of a powerful algorithm which requires both the value and gradient at given parameter values is BFGS quasi-Newton minimisation .

4.2.3 The Hessian

A few algorithms, including the most powerful, require in addition the Hessian at given parameter values θ. The Hessian is the matrix of second partial derivatives defined as

$$\nabla^2 = \begin{pmatrix} \frac{\partial^2}{\partial\theta_1^2} & \frac{\partial^2}{\partial\theta_1 \cdot \partial\theta_2} & \cdots & \frac{\partial^2}{\partial\theta_1 \cdot \partial\theta_n} \\ \frac{\partial^2}{\partial\theta_2 \cdot \partial\theta_1} & \frac{\partial^2}{\partial\theta_2^2} & \cdots & \frac{\partial^2}{\partial\theta_2 \cdot \partial\theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial\theta_n \cdot \partial\theta_1} & \frac{\partial^2}{\partial\theta_n \cdot \partial\theta_2} & \cdots & \frac{\partial^2}{\partial\theta_n^2} \end{pmatrix} \tag{4.27}$$

The most powerful minimisation algorithm, Newton minimisation , requires the value, gradient, and Hessian at the given parameter values.

4.3 Value, gradient, and Hessian dependency chain

The dependency chain outlined in the introduction to this chapter, that the chi-squared function is dependent on the relaxation equations which are dependent on the transformed relaxation equations which themselves are dependent on the spectral density functions, combine with the values, gradients, and Hessians to create a complex web of dependencies. The relationship between all the values, gradients, and Hessians are outlined in Figure 4.1.

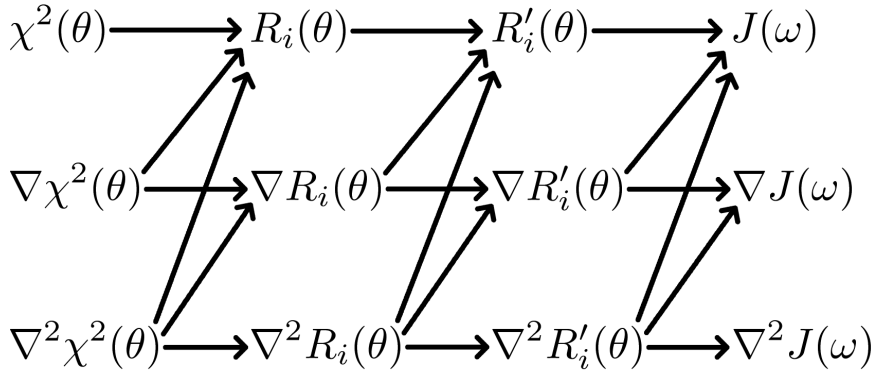

Figure 4.1: Dependencies between the χ^2, transformed relaxation, relaxation, and spectral density equations, gradients, and Hessians.

4.4 χ^2 values, gradients, and Hessians

4.4.1 χ^2 values

The χ^2 value is

$$\chi^2(\theta) = \sum_{i=1}^{n} \frac{(\mathrm{R}_i - \mathrm{R}_i(\theta))^2}{\sigma_i^2}, \tag{4.28}$$

which is the same as equation (4.1) on page 23.

4.4.2 χ^2 gradients

The χ^2 gradient is

$$\nabla\chi^2(\theta) = 2\sum_{i=1}^{n} \frac{(\mathrm{R}_i - \mathrm{R}_i(\theta))^2}{\sigma_i^2} \nabla\mathrm{R}_i(\theta). \tag{4.29}$$

4.4.3 χ^2 Hessians

The χ^2 Hessian is

$$\nabla^2\chi^2(\theta) = 2\sum_{i=1}^{n} \frac{1}{\sigma_i^2} \left(\nabla\mathrm{R}_i(\theta) \cdot \nabla\mathrm{R}_i(\theta)^T - (\mathrm{R}_i - \mathrm{R}_i(\theta))\nabla^2\mathrm{R}_i(\theta)\right). \tag{4.30}$$

4.5 $\mathrm{R}_i(\theta)$ values, gradients, and Hessians

4.5.1 $\mathrm{R}_i(\theta)$ values

The $\mathrm{R}_i(\theta)$ values are given by

$$\mathrm{R}_1(\theta) = \mathrm{R}_1'(\theta), \tag{4.31}$$
$$\mathrm{R}_2(\theta) = \mathrm{R}_2'(\theta), \tag{4.32}$$
$$\mathrm{NOE}(\theta) = 1 + \frac{\gamma_H}{\gamma_X}\frac{\sigma_{\mathrm{NOE}}(\theta)}{\mathrm{R}_1(\theta)}. \tag{4.33}$$

4.5.2 $\mathrm{R}_i(\theta)$ gradients

The $\mathrm{R}_i(\theta)$ gradients are

$$\nabla\mathrm{R}_1(\theta) = \nabla\mathrm{R}_1'(\theta), \tag{4.34}$$
$$\nabla\mathrm{R}_2(\theta) = \nabla\mathrm{R}_2'(\theta), \tag{4.35}$$
$$\nabla\mathrm{NOE}(\theta) = \frac{\gamma_H}{\gamma_X}\frac{1}{\mathrm{R}_1(\theta)^2}\Big(\mathrm{R}_1(\theta)\nabla\sigma_{\mathrm{NOE}}(\theta) - \sigma_{\mathrm{NOE}}(\theta)\nabla\mathrm{R}_1(\theta)\Big). \tag{4.36}$$

4.5.3 $\mathrm{R}_i(\theta)$ Hessians

The $\mathrm{R}_i(\theta)$ Hessians are

$$\nabla^2\mathrm{R}_1(\theta) = \nabla^2\mathrm{R}_1'(\theta), \tag{4.37}$$
$$\nabla^2\mathrm{R}_2(\theta) = \nabla^2\mathrm{R}_2'(\theta), \tag{4.38}$$
$$\begin{aligned}\nabla^2\mathrm{NOE}(\theta) = \frac{\gamma_H}{\gamma_X}\frac{1}{\mathrm{R}_1(\theta)^3}\Big[&\sigma_{\mathrm{NOE}}(\theta)\Big(2\nabla\mathrm{R}_1(\theta)\cdot\nabla\mathrm{R}_1(\theta)^T - \mathrm{R}_1(\theta)\nabla^2\mathrm{R}_1(\theta)\Big)\\ &- \mathrm{R}_1(\theta)\Big(\nabla\sigma_{\mathrm{NOE}}(\theta)\cdot\nabla\mathrm{R}_1(\theta)^T - \mathrm{R}_1(\theta)\nabla^2\sigma_{\mathrm{NOE}}(\theta)\Big)\Big].\end{aligned} \tag{4.39}$$

4.6 $\mathrm{R}'_i(\theta)$ values, gradients, and Hessians

The partial and second partial derivatives of the transformed relaxation equations, $\mathrm{R}'_i(\theta)$, are different for each parameter of the vector θ. The vector representation of the gradient, $\nabla \mathrm{R}'_i(\theta)$, and the matrix representation of the Hessian, $\nabla^2 \mathrm{R}'_i(\theta)$, can be reconstructed from the individual elements presented below.

4.6.1 Components of the $\mathrm{R}'_i(\theta)$ equations

To simplify the calculations of the gradients and Hessians, the $\mathrm{R}'_i(\theta)$ equations have been broken down into their various components. These include the dipolar and CSA constants as well as the dipolar and CSA spectral density terms for each of the three transformed relaxation data types, R_1, R_2, and σ_{NOE}. The segregation of these components simplifies the maths as many partial derivatives of the components are zero.

Dipolar constant

The dipolar constant is defined as

$$d = \frac{1}{4}\left(\frac{\mu_0}{4\pi}\right)^2 \frac{\left(\gamma_H \gamma_X \hbar\right)^2}{< r^6 >}. \tag{4.40}$$

This component of the relaxation equations is independent of the parameter of the spectral density function, θ_j, the chemical exchange parameter, σ_{ex}, and the CSA parameter, $\Delta\sigma$. Therefore, the partial and second partial derivatives with respect to these parameters is zero. Only the partial derivative with respect to the bond length, r, is non-zero, being

$$d' \equiv \frac{\mathrm{d}d}{\mathrm{d}r} = -\frac{3}{2}\left(\frac{\mu_0}{4\pi}\right)^2 \frac{\left(\gamma_H \gamma_X \hbar\right)^2}{< r^7 >}. \tag{4.41}$$

The double partial derivative with respect to the bond length twice is

$$d'' \equiv \frac{\mathrm{d}^2 d}{\mathrm{d}r^2} = \frac{21}{2}\left(\frac{\mu_0}{4\pi}\right)^2 \frac{\left(\gamma_H \gamma_X \hbar\right)^2}{< r^8 >}. \tag{4.42}$$

CSA constant

The CSA constant is defined as

$$c = \frac{\left(\omega_X \cdot \Delta\sigma\right)^2}{3}. \tag{4.43}$$

The partial derivative of this component with respect to all parameters but the CSA parameter $\Delta\sigma$ is zero. This partial derivative is

$$c' \equiv \frac{\mathrm{d}c}{\mathrm{d}\Delta\sigma} = \frac{2\omega_X^2 \cdot \Delta\sigma}{3}. \tag{4.44}$$

The CSA constant double partial derivative with respect to $\Delta\sigma$ is

$$c'' \equiv \frac{\mathrm{d}^2 c}{\mathrm{d}\Delta\sigma^2} = \frac{2\omega_X^2}{3}. \tag{4.45}$$

Spectral density terms of the R_1 dipolar component

For the dipolar component of the R_1 equation, the spectral density terms are

$$J_d^{R_1} = J(\omega_H - \omega_X) + 3J(\omega_X) + 6J(\omega_H + \omega_X). \tag{4.46}$$

The partial derivative of these terms with respect to the parameter of the spectral density function θ_j is

$$J_d^{R_1\prime} \equiv \frac{\partial J_d^{R_1}}{\partial \theta_j} = \frac{\partial J(\omega_H - \omega_X)}{\partial \theta_j} + 3\frac{\partial J(\omega_X)}{\partial \theta_j} + 6\frac{\partial J(\omega_H + \omega_X)}{\partial \theta_j}. \tag{4.47}$$

The second partial derivative with respect to the parameter of the spectral density function θ_j and θ_k is

$$J_d^{R_1\prime\prime} \equiv \frac{\partial^2 J_d^{R_1}}{\partial \theta_j \cdot \partial \theta_k} = \frac{\partial^2 J(\omega_H - \omega_X)}{\partial \theta_j \cdot \partial \theta_k} + 3\frac{\partial^2 J(\omega_X)}{\partial \theta_j \cdot \partial \theta_k} + 6\frac{\partial^2 J(\omega_H + \omega_X)}{\partial \theta_j \cdot \partial \theta_k}. \tag{4.48}$$

Spectral density terms of the R_1 CSA component

For the CSA component of the R_1 equation, the spectral density terms are

$$J_c^{R_1} = J(\omega_X). \tag{4.49}$$

The partial derivative of these terms with respect to the parameter of the spectral density function θ_j is

$$J_c^{R_1\prime} \equiv \frac{\partial J_c^{R_1}}{\partial \theta_j} = \frac{\partial J(\omega_X)}{\partial \theta_j}. \tag{4.50}$$

The second partial derivative with respect to the parameter of the spectral density function θ_j and θ_k is

$$J_c^{R_1\prime\prime} \equiv \frac{\partial^2 J_c^{R_1}}{\partial \theta_j . \partial \theta_k} = \frac{\partial^2 J(\omega_X)}{\partial \theta_j \cdot \partial \theta_k}. \tag{4.51}$$

Spectral density terms of the R_2 dipolar component

For the dipolar component of the R_2 equation, the spectral density terms are

$$J_d^{R_2} = 4J(0) + J(\omega_H - \omega_X) + 3J(\omega_X) + 6J(\omega_H) + 6J(\omega_H + \omega_X). \tag{4.52}$$

The partial derivative of these terms with respect to the parameter of the spectral density function θ_j is

$$J_d^{R_2\prime} \equiv \frac{\partial J_d^{R_2}}{\partial \theta_j} = 4\frac{\partial J(0)}{\partial \theta_j} + \frac{\partial J(\omega_H - \omega_X)}{\partial \theta_j} + 3\frac{\partial J(\omega_X)}{\partial \theta_j} + 6\frac{\partial J(\omega_H)}{\partial \theta_j} + 6\frac{\partial J(\omega_H + \omega_X)}{\partial \theta_j}. \tag{4.53}$$

The second partial derivative with respect to the parameter of the spectral density function θ_j and θ_k is

$$J_d^{\mathrm{R}_2}{}'' \equiv \frac{\partial^2 J_d^{\mathrm{R}_2}}{\partial\theta_j \cdot \partial\theta_k} = 4\frac{\partial^2 J(0)}{\partial\theta_j \cdot \partial\theta_k} + \frac{\partial^2 J(\omega_H - \omega_X)}{\partial\theta_j \cdot \partial\theta_k} + 3\frac{\partial^2 J(\omega_X)}{\partial\theta_j \cdot \partial\theta_k} + 6\frac{\partial^2 J(\omega_H)}{\partial\theta_j \cdot \partial\theta_k} + 6\frac{\partial^2 J(\omega_H + \omega_X)}{\partial\theta_j \cdot \partial\theta_k}. \quad (4.54)$$

Spectral density terms of the R_2 CSA component

For the CSA component of the R_2 equation, the spectral density terms are

$$J_c^{\mathrm{R}_2} = 4J(0) + 3J(\omega_X). \quad (4.55)$$

The partial derivative of these terms with respect to the parameter of the spectral density function θ_j is

$$J_c^{\mathrm{R}_2}{}' \equiv \frac{\partial J_c^{\mathrm{R}_2}}{\partial\theta_j} = 4\frac{\partial J(0)}{\partial\theta_j} + 3\frac{\partial J(\omega_X)}{\partial\theta_j}. \quad (4.56)$$

The second partial derivative with respect to the parameter of the spectral density function θ_j and θ_k is

$$J_c^{\mathrm{R}_2}{}'' \equiv \frac{\partial^2 J_c^{\mathrm{R}_2}}{\partial\theta_j \cdot \partial\theta_k} = 4\frac{\partial^2 J(0)}{\partial\theta_j \cdot \partial\theta_k} + 3\frac{\partial^2 J(\omega_X)}{\partial\theta_j \cdot \partial\theta_k}. \quad (4.57)$$

Spectral density terms of the σ_{NOE} dipolar component

For the dipolar component of the σ_{NOE} equation, the spectral density terms are

$$J_d^{\sigma_{\mathrm{NOE}}} = 6J(\omega_H + \omega_X) - 6J(\omega_H - \omega_X). \quad (4.58)$$

The partial derivative of these terms with respect to the parameter of the spectral density function θ_j is

$$J_d^{\sigma_{\mathrm{NOE}}{}'} \equiv \frac{\partial J_d^{\sigma_{\mathrm{NOE}}}}{\partial\theta_j} = 6\frac{\partial J(\omega_H + \omega_X)}{\partial\theta_j} - 6\frac{\partial J(\omega_H - \omega_X)}{\partial\theta_j}. \quad (4.59)$$

The second partial derivative with respect to the parameter of the spectral density function θ_j and θ_k is

$$J_d^{\sigma_{\mathrm{NOE}}{}''} \equiv \frac{\partial^2 J_d^{\sigma_{\mathrm{NOE}}}}{\partial\theta_j \cdot \partial\theta_k} = 6\frac{\partial^2 J(\omega_H + \omega_X)}{\partial\theta_j \cdot \partial\theta_k} - 6\frac{\partial^2 J(\omega_H - \omega_X)}{\partial\theta_j \cdot \partial\theta_k}. \quad (4.60)$$

4.6.2 $\mathrm{R}'_i(\theta)$ values

The three relaxation equations, utilising the above components, can be expressed as

$$\mathrm{R}_1(\theta) = dJ_d^{\mathrm{R}_1} + cJ_c^{\mathrm{R}_1}, \tag{4.61}$$

$$\mathrm{R}_2(\theta) = \frac{d}{2}J_d^{\mathrm{R}_2} + \frac{c}{6}J_c^{\mathrm{R}_2}, \tag{4.62}$$

$$\sigma_{\mathrm{NOE}}(\theta) = dJ_d^{\sigma_{\mathrm{NOE}}}. \tag{4.63}$$

4.6.3 $\mathrm{R}'_i(\theta)$ gradients

The partial derivatives with respect to the parameter of the spectral density functions, the chemical exchange parameter, CSA parameter, and bond length parameters are all different and are presented below.

Spectral density function parameter

The partial derivatives of the relaxation equations with respect to the parameter of the spectral density function θ_j are

$$\frac{\partial \mathrm{R}_1(\theta)}{\partial \theta_j} = dJ_d^{\mathrm{R}_1\prime} + cJ_c^{\mathrm{R}_1\prime}, \tag{4.64}$$

$$\frac{\partial \mathrm{R}_2(\theta)}{\partial \theta_j} = \frac{d}{2}J_d^{\mathrm{R}_2\prime} + \frac{c}{6}J_c^{\mathrm{R}_2\prime}, \tag{4.65}$$

$$\frac{\partial \sigma_{\mathrm{NOE}}(\theta)}{\partial \theta_j} = dJ_d^{\sigma_{\mathrm{NOE}}\prime}. \tag{4.66}$$

Chemical exchange parameter

The partial derivatives of the relaxation equations with respect to the chemical exchange parameter R_{ex} are

$$\frac{\partial \mathrm{R}_1(\theta)}{\partial R_{ex}} = 0, \tag{4.67}$$

$$\frac{\partial \mathrm{R}_2(\theta)}{\partial R_{ex}} = 1, \tag{4.68}$$

$$\frac{\partial \sigma_{\mathrm{NOE}}(\theta)}{\partial R_{ex}} = 0. \tag{4.69}$$

CSA parameter

The partial derivatives of the relaxation equations with respect to the CSA parameter $\Delta\sigma$ are

$$\frac{\partial \mathrm{R}_1(\theta)}{\partial \Delta\sigma} = c' J_c^{\mathrm{R}_1}, \tag{4.70}$$

$$\frac{\partial \mathrm{R}_2(\theta)}{\partial \Delta\sigma} = \frac{c'}{6} J_c^{\mathrm{R}_2}, \tag{4.71}$$

$$\frac{\partial \sigma_{\mathrm{NOE}}(\theta)}{\partial \Delta\sigma} = 0. \tag{4.72}$$

Bond length parameter

The partial derivatives of the relaxation equations with respect to the bond length parameter r are

$$\frac{\partial \mathrm{R}_1(\theta)}{\partial r} = d' J_d^{\mathrm{R}_1}, \tag{4.73}$$

$$\frac{\partial \mathrm{R}_2(\theta)}{\partial r} = \frac{d'}{2} J_d^{\mathrm{R}_2}, \tag{4.74}$$

$$\frac{\partial \sigma_{\mathrm{NOE}}(\theta)}{\partial r} = d' J_d^{\sigma_{\mathrm{NOE}}}. \tag{4.75}$$

4.6.4 $\mathrm{R}_i'(\theta)$ Hessians

The second partial derivatives with respect to the parameter of the spectral density functions, the chemical exchange parameter, CSA parameter, and bond length parameters are presented below.

Spectral density function parameter - Spectral density function parameter

The second partial derivatives of the relaxation equations with respect to the parameter of the spectral density functions θ_j and θ_k are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial \theta_j \cdot \partial \theta_k} = d J_d^{\mathrm{R}_1\prime\prime} + c J_c^{\mathrm{R}_1\prime\prime}, \tag{4.76}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial \theta_j \cdot \partial \theta_k} = \frac{d}{2} J_d^{\mathrm{R}_2\prime\prime} + \frac{c}{6} J_c^{\mathrm{R}_2\prime\prime}, \tag{4.77}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial \theta_j \cdot \partial \theta_k} = d J_d^{\sigma_{\mathrm{NOE}}\prime\prime}. \tag{4.78}$$

Spectral density function parameter - Chemical exchange parameter

The second partial derivatives of the relaxation equations with respect to the parameter of the spectral density function θ_j and the chemical exchange parameter R_{ex} are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial\theta_j \cdot \partial R_{ex}} = 0, \tag{4.79}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial\theta_j \cdot \partial R_{ex}} = 0, \tag{4.80}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial\theta_j \cdot \partial R_{ex}} = 0. \tag{4.81}$$

Spectral density function parameter - CSA parameter

The second partial derivatives of the relaxation equations with respect to the parameter of the spectral density function θ_j and the CSA parameter $\Delta\sigma$ are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial\theta_j \cdot \partial \Delta\sigma} = c' J_c^{\mathrm{R}_1\prime}, \tag{4.82}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial\theta_j \cdot \partial \Delta\sigma} = \frac{c'}{6} J_c^{\mathrm{R}_2\prime}, \tag{4.83}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial\theta_j \cdot \partial \Delta\sigma} = 0. \tag{4.84}$$

Spectral density function parameter - Bond length parameter

The second partial derivatives of the relaxation equations with respect to the parameter of the spectral density function θ_j and the bond length parameter r are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial\theta_j \cdot \partial r} = d' J_d^{\mathrm{R}_1\prime}, \tag{4.85}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial\theta_j \cdot \partial r} = \frac{d'}{2} J_d^{\mathrm{R}_2\prime}, \tag{4.86}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial\theta_j \cdot \partial r} = d' J_d^{\sigma_{\mathrm{NOE}}\prime}. \tag{4.87}$$

Chemical exchange parameter - Chemical exchange parameter

The second partial derivatives of the relaxation equations with respect to the chemical exchange parameter R_{ex} twice are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial {R_{ex}}^2} = 0, \tag{4.88}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial {R_{ex}}^2} = 0, \tag{4.89}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial {R_{ex}}^2} = 0. \tag{4.90}$$

Chemical exchange parameter - CSA parameter

The second partial derivatives of the relaxation equations with respect to the chemical exchange parameter R_{ex} and the CSA parameter $\Delta\sigma$ are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial R_{ex} \cdot \partial \Delta\sigma} = 0, \tag{4.91}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial R_{ex} \cdot \partial \Delta\sigma} = 0, \tag{4.92}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial R_{ex} \cdot \partial \Delta\sigma} = 0. \tag{4.93}$$

Chemical exchange parameter - Bond length parameter

The second partial derivatives of the relaxation equations with respect to the chemical exchange parameter R_{ex} and the bond length parameter r are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial R_{ex} \cdot \partial r} = 0, \tag{4.94}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial R_{ex} \cdot \partial r} = 0, \tag{4.95}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial R_{ex} \cdot \partial r} = 0. \tag{4.96}$$

CSA parameter - CSA parameter

The second partial derivatives of the relaxation equations with respect to the CSA parameter $\Delta\sigma$ twice are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial \Delta\sigma^2} = c'' J_c^{\mathrm{R}_1}, \tag{4.97}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial \Delta\sigma^2} = \frac{c''}{6} J_c^{\mathrm{R}_2}, \tag{4.98}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial \Delta\sigma^2} = 0. \tag{4.99}$$

CSA parameter - Bond length parameter

The second partial derivatives of the relaxation equations with respect to the CSA parameter $\Delta\sigma$ and the bond length parameter r are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial \Delta\sigma \cdot \partial r} = 0, \tag{4.100}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial \Delta\sigma \cdot \partial r} = 0, \tag{4.101}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial \Delta\sigma \cdot \partial r} = 0. \tag{4.102}$$

Bond length parameter - Bond length parameter

The second partial derivatives of the relaxation equations with respect to the bond length parameter r twice are

$$\frac{\partial^2 \mathrm{R}_1(\theta)}{\partial r^2} = d'' J_d^{\mathrm{R}_1}, \tag{4.103}$$

$$\frac{\partial^2 \mathrm{R}_2(\theta)}{\partial r^2} = \frac{d''}{2} J_d^{\mathrm{R}_2}, \tag{4.104}$$

$$\frac{\partial^2 \sigma_{\mathrm{NOE}}(\theta)}{\partial r^2} = d'' J_d^{\sigma_{\mathrm{NOE}}}. \tag{4.105}$$

4.7 Ellipsoidal diffusion tensor

4.7.1 Ellipsoid weight derivatives

Chapter 5

Development of relax

This chapter is written for developers or those who would like to extend the functionality of relax. It is not required for using relax.

5.1 Coding conventions

5.1.1 Indentation

Indentation should be set to four spaces rather than a tab character. This is the recommendation given in the python style guide found at `'http://www.python.org/doc/essays/styleguide.html'`. For vi, add the following lines to `'~/.vimrc'`:

```
set tabstop=4
set shiftwidth=4
set expandtab
```

Certain versions of vim, those within the 6.2 series, contain a bug where the tabstop value cannot be changed using the `'~/.vimrc'` file (although typing `':set tabstop=4'` in vim will fix it). One solution is to edit the file `'python.vim'` which is located in the path `'/usr/share/vim/ftplugin/'` or equivalent. It contains the two lines:

```
" Python always uses a 'tabstop' of 8.
setlocal ts=8
```

If these lines are deleted, the bug will be removed. Another way to fix the problem is to install newer versions of the run-time files (which will pretty much do the same thing).

5.1.2 Doc strings

These should be set to no more than 100 characters long including all leading white space. The standard Python convention of a one line description separated from a detailed description by an empty line should be adhered to.

5.2 The make system

For UNIX like systems, including GNU/Linux, various components of the program relax can be created using the Unix make utility. This includes C module compilation, manual creation, distribution creation, and cleaning up and removing certain files. Makefiles exist in the various directories to facilitate this process.

5.2.1 C module compilation

As described in the installation chapter, typing `'make'` in the base directory will create the shared objects which are imported into Python as modules.

5.2.2 Creation of the PDF manual

To create the PDF version of the relax manual, type

```
make manual
```

in the base directory. Make will then shift to the `'docs/latex'` directory and run a series of shell commands to create the manual from the LaTeX sources. This is dependent on the programs `'rm'`, `'latex'`, `'makeindex'`, `'dvips'`, and `'ps2pdf'` being located in the environment's path.

5.2.3 Creation of the HTML manual

The HTML version of the relax manual is made by typing

```
make manual_html
```

in the base directory. Again make will then shift to the `'docs/latex'` directory. One command calling the program `'latex2html'` will be executed which will create HTML pages from the LaTeX sources.

5.2.4 Making distribution archives

This section is incomplete. Creation of the HTML manual

5.2.5 Cleaning up

If the command

```
make clean
```

is run in the base directory, all Python byte compiled files '*.pyc', all C object files '*.o', all C shared object files '*.so', and any backup files with the extension '*.bak' are removed from all subdirectoies. In addition, any temporary LaTeX compilation files are removed from the `'docs/latex'` directory.

Chapter 6

Alphabetical listing of user functions

The following is a listing with descriptions of all the user functions available within the relax prompt and scripting environments. These are simply an alphabetical list of the docstrings which can normally be viewed in prompt mode by typing '`help(function)`' .

6.1 A warning about the formatting

The following documentation of the user functions has been automatically generated by a script which extracts and formats the docstring associated with each function. There may therefore be instances where the formatting has failed or where there are inconsistencies.

6.2 The list of functions

Each user function is presented within it's own subsection with the documentation broken into multiple parts, the synopsis, the default arguments, and the sections from the function's docstring.

6.2.1 The synopsis

The synopsis presents a brief description of the function. It is taken as the first line of the docstring when browsing the help system .

6.2.2 Defaults

This section lists all the arguments taken by the function and their default values. To invoke the function, type the function name then in brackets type a comma separated list of arguments.

The first argument printed is always 'self' but you can safely ignore it. 'self' is part of the object oriented programming within Python and is automatically prefixed to the list of arguments you supply. Therefore you can't provide 'self' as the first argument, even if you do try.

6.2.3 Docstring sectioning

All other sections are created from the sectioning within the actual docstring.

6.2.4 angles

Synopsis

Function for calculating the angles between the XH bond vector and the diffusion tensor.

Defaults

angles(self, run=None)

Keyword Arguments

run: The name of the run.

Description

If the diffusion tensor is isotropic for the run, then nothing will be done.

If the diffusion tensor is axially symmetric, then the angle α will be calculated for each XH bond vector.

If the diffusion tensor is asymmetric, then the three angles will be calculated.

6.2.5 calc

Synopsis

Function for calculating the function value.

Defaults

calc(self, run=None, print_flag=1)

Keyword Arguments

run: The name of the run.

6.2.6 dasha.create

Synopsis

Function for creating the Dasha script.

Defaults

dasha.create(self, run=None, algor='LM', dir=None, force=0)

Keyword Arguments

run: The name of the run.

algor: The minimisation algorithm.

dir: The directory to place the files. The default is the value of '`run`'.

force: A flag which if set to 1 will cause the results file to be overwritten if it already exists.

Description

The script file created is called '`dir/dasha_script`'.

Optimisation algorithms

The two minimisation algorithms within Dasha are accessible through the algor argument which can be set to:

'`LM`' - The Levenberg-Marquardt algorithm.

'`NR`' - Newton-Raphson algorithm.

For Levenberg-Marquardt minimisation, the function '`lmin`' will be called, while for Newton -Raphson, the function '`min`' will be executed.

6.2.7 dasha.execute

Synopsis

Function for executing Dasha.

Defaults

dasha.execute(self, run=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

dir: The directory to place the files. The default is the value of '`run`'.

force: A flag which if set to 1 will cause the results file to be overwritten if it already exists.

Execution

Dasha will be executed as

```
$ dasha < dasha_script | tee dasha_results
```

6.2.8 dasha.extract

Synopsis

Function for extracting data from the Dasha results file.

Defaults

dasha.extract(self, run=None, dir=None)

Keyword Arguments

run: The name of the run.

dir: The directory where the file '`dasha_results`' is found. The default is the value of '`run`'.

6.2.9 diffusion_tensor.copy

Synopsis

Function for copying diffusion tensor data from run1 to run2.

Defaults

diffusion_tensor.copy(self, run1=None, run2=None)

Keyword Arguments

run1: The name of the run to copy the sequence from.

run2: The name of the run to copy the sequence to.

Description

This function will copy the diffusion tensor data from '`run1`' to '`run2`'. '`run2`' must not contain any diffusion tensor data.

Examples

To copy the diffusion tensor from run '`m1`' to run '`m2`', type:

```
relax> diffusion_tensor.copy('m1', 'm2')
```

6.2.10 diffusion_tensor.delete

Synopsis

Function for deleting diffusion tensor data.

Defaults

diffusion_tensor.delete(self, run=None)

Keyword Arguments

run: The name of the run.

Description

This function will delete all diffusion tensor data for the given run.

6.2.11 diffusion_tensor.display

Synopsis

Function for displaying the diffusion tensor.

Defaults

diffusion_tensor.display(self, run=None)

Keyword Arguments

run: The name of the run.

6.2.12 diffusion_tensor.init

Synopsis

Function for initialising the diffusion tensor.

Defaults

diffusion_tensor.init(self, run=None, params=None, time_scale=1.0, d_scale=1.0, angle_units='deg', param_types=0, spheroid_type=None, fixed=1)

Keyword Arguments

run: The name of the run to assign the data to.

params: The diffusion tensor data.

time_scale: The correlation time scaling value.

d_scale: The diffusion tensor eigenvalue scaling value.

angle_units: The units for the angle parameters.

param_types: A flag to select different parameter combinations.

spheroid_type: A string which, if supplied together with spheroid parameters, will restrict the tensor to either being '`oblate`' or '`prolate`'.

fixed: A flag specifying whether the diffusion tensor is fixed or can be optimised.

The sphere (isotropic diffusion)

When the molecule diffuses as a sphere, all three eigenvalues of the diffusion tensor are equal, $\mathfrak{D}_x = \mathfrak{D}_y = \mathfrak{D}_z$. In this case, the orientation of the XH bond vector within the diffusion frame is inconsequential to relaxation, hence, the spherical or Euler angles are undefined. Therefore solely a single geometric parameter, either τ_m or $\mathfrak{D}_{iso}$, can fully and sufficiently parameterise the diffusion tensor. The correlation function for the global rotational diffusion is

$$C(\tau) = \frac{1}{5} e^{-\tau / tm},$$

To select isotropic diffusion, the parameters argument should be a single floating point number. The number is the value of the isotropic global correlation time, τ_m, in seconds. To specify the time in nanoseconds, set the '`time_scale`' argument to 1e-9. Alternative parameters can be used by changing the '`param_types`' flag to the following integers

0 – τ_m (Default),

1 – $\mathfrak{D}_{iso}$,

where

$$1 \,/\, \tau_m = 6\mathfrak{D}_{iso}.$$

The spheroid (axially symmetric diffusion)

When two of the three eigenvalues of the diffusion tensor are equal, the molecule diffuses as a spheroid. Four pieces of information are required to specify this tensor, the two geometric parameters, $\mathfrak{D}_{iso}$ and $\mathfrak{D}_a$, and the two orientational parameters, the polar angle θ and the azimuthal angle ϕ describing the orientation of the axis of symmetry. The correlation function of the global diffusion is

$$C(\tau) = \frac{1}{5}\sum_{i=-1}^{1} c_i \cdot e^{-\tau / \tau_i},$$

where

$$\text{c-1} = 1/4\ (3\ \delta_z\hat{}2 - 1)\hat{}2,$$

$$\text{c0} = 3\ \delta_z\hat{}2\ (1 - \delta_z\hat{}2),$$

$$\text{c1} = 3/4\ (\delta_z\hat{}2 - 1)\hat{}2,$$

and

$$1 \,/\, \tau\ \text{-1} = 6\mathfrak{D}_{iso} - 2\mathfrak{D}_a,$$

$$1 \,/\, \tau\ 0 = 6\mathfrak{D}_{iso} - \mathfrak{D}_a,$$

$$1 \,/\, \tau\ 1 = 6\mathfrak{D}_{iso} + 2\mathfrak{D}_a.$$

The direction cosine δ_z is defined as the cosine of the angle α between the XH bond vector and the unique axis of the diffusion tensor.

To select axially symmetric anisotropic diffusion, the parameters argument should be a tuple of floating point numbers of length four. A tuple is a type of data structure enclosed in round brackets, the elements of which are separated by commas. Alternative sets of parameters, '`param_types`', are

0 – (τ_m, $\mathfrak{D}_a$, θ, ϕ) (Default),

1 – ($\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, θ, ϕ),

2 – (τ_m, $\mathfrak{D}_{ratio}$, θ, ϕ),

3 – ($\mathfrak{D}_{\parallel}$, $\mathfrak{D}_{\perp}$, θ, ϕ),

4 – ($\mathfrak{D}_{iso}$, $\mathfrak{D}_{ratio}$, θ, ϕ),

where

$$\tau_m = 1 \ / \ 6\mathfrak{D}_{iso},$$

$$\mathfrak{D}_{iso} = 1/3 \ (\mathfrak{D}_{\parallel} + 2\mathfrak{D}_{\perp}),$$

$$\mathfrak{D}_a = \mathfrak{D}_{\parallel} \text{ - } \mathfrak{D}_{\perp},$$

$$\mathfrak{D}_{ratio} = \mathfrak{D}_{\parallel} \ / \ \mathfrak{D}_{\perp}.$$

The spherical angles $\{\theta, \phi\}$ orienting the unique axis of the diffusion tensor within the PDB frame are defined between

$$0 \leq \theta \leq \pi,$$

$$0 \leq \phi \leq 2\pi,$$

while the angle α which is the angle between this axis and the given XH bond vector is defined between

$$0 \leq \alpha \leq 2\pi.$$

The '`spheroid_type`' argument should be '`oblate`', '`prolate`', or None. The argument will be ignored if the diffusion tensor is not axially symmetric. If '`oblate`' is given, then the constraint $\mathfrak{D}_a \leq 0$ is used while if '`prolate`' is given, then the constraint $\mathfrak{D}_a \geq 0$ is used. If nothing is supplied, then $\mathfrak{D}_a$ will be allowed to have any values. To prevent minimisation of diffusion tensor parameters in a space with two minima, it is recommended to specify which tensor is to be minimised, thereby partitioning the two minima into the two subspaces along the boundry $\mathfrak{D}_a = 0$.

The ellipsoid (rhombic diffusion)

When all three eigenvalues of the diffusion tensor are different, the molecule diffuses as an ellipsoid. This diffusion is also known as fully anisotropic, asymmetric, or rhombic. The full tensor is specified by six pieces of information, the three geometric parameters $\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, and $\mathfrak{D}_r$ representing the isotropic, anisotropic, and rhombic components of the tensor, and the three Euler angles α, β, and γ orienting the tensor within the PDB frame. The correlation function is

$$C(\tau) = \frac{1}{5}\sum_{i=-2}^{2} c_i \cdot e^{-\tau / \tau_i},$$

where the weights on the exponentials are

c-2 = 1/4 (d + e),

c-1 = 3 δ_y^2 δ_z^2,

c0 = 3 δ_x^2 δ_z^2,

c1 = 3 δ_x^2 δ_y^2,

c2 = 1/4 (d + e).

Let

$\mathfrak{R}$ = sqrt(1 + 3$\mathfrak{D}_r$),

then

d = 3 (δ_x^4 + δ_y^4 + δ_z^4) - 1,

e = - 1 / $\mathfrak{R}$ ((1 + 3$\mathfrak{D}_r$)(δ_x^4 + 2δ_y^2 δ_z^2) + (1 - 3$\mathfrak{D}_r$)(δ_y^4 + 2δ_x^2 δ_z^2) - 2(δ_z^4 + 2δ_x^2 δ_y^2)).

The correlation times are

1 / τ -2 = 6$\mathfrak{D}_{iso}$ - 2$\mathfrak{D}_a$. $\mathfrak{R}$,

$1 / \tau$ -1 = $6\mathfrak{D}_{iso}$ - $\mathfrak{D}_a$ (1 + $3\mathfrak{D}_r$),

$1 / \tau$ 0 = $6\mathfrak{D}_{iso}$ - $\mathfrak{D}_a$ (1 - $3\mathfrak{D}_r$),

$1 / \tau$ 1 = $6\mathfrak{D}_{iso}$ + $2\mathfrak{D}_a$,

$1 / \tau$ 1 = $6\mathfrak{D}_{iso}$ + $2\mathfrak{D}_a$. $\mathfrak{R}$.

The three direction cosines δ_x, δ_y, and δ_z are the coordinates of a unit vector parallel to the XH bond vector. Hence the unit vector is $[\delta_x, \delta_y, \delta_z]$.

To select fully anisotropic diffusion, the parameters argument should be a tuple of length six. A tuple is a type of data structure enclosed in round brackets, the elements of which are separated by commas. Alternative sets of parameters, '`param_types`', are

0 – (τ_m, $\mathfrak{D}_a$, $\mathfrak{D}_r$, α, β, γ) (Default),

1 – ($\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, $\mathfrak{D}_r$, α, β, γ),

2 – ($\mathfrak{D}_x$, $\mathfrak{D}_y$, $\mathfrak{D}_z$, α, β, γ),

where

$\tau_m = 1 / 6\mathfrak{D}_{iso}$,

$\mathfrak{D}_{iso} = 1/3\ (\mathfrak{D}_x + \mathfrak{D}_y + \mathfrak{D}_z)$,

$\mathfrak{D}_a = \mathfrak{D}_z$ - $(\mathfrak{D}_x + \mathfrak{D}_y)/2$,

$\mathfrak{D}_r = (\mathfrak{D}_y$ - $\mathfrak{D}_x)/2\mathfrak{D}_a$.

The angles α, β, and γ are the Euler angles describing the diffusion tensor within the PDB frame. These angles are defined using the z-y-z axis rotation notation where α is the initial rotation angle around the z-axis, β is the rotation angle around the y-axis, and γ is the final rotation around the z-axis again. The angles are defined between

$0 \leq \alpha \leq 2\pi$,

$0 \leq \beta \leq \pi$,

$0 \leq \gamma \leq 2\pi$.

Within the PDB frame, the XH bond vector is described using the spherical angles θ and ϕ where θ is the polar angle and ϕ is the azimuthal angle defined between

$0 \leq \theta \leq \pi$,

$0 \leq \phi \leq 2\pi$.

Units

The '`time_scale`' argument should be a floating point number. The only parameter affected by this value is τ_m.

The '`d_scale`' argument should also be a floating point number. Parameters affected by this value are $\mathfrak{D}_{iso}$, $\mathfrak{D}_{\parallel}$, $\mathfrak{D}_{\perp}$, $\mathfrak{D}_a$, $\mathfrak{D}_x$, $\mathfrak{D}_y$, and $\mathfrak{D}_z$. Significantly, $\mathfrak{D}_r$ is not affected.

The '`angle_units`' argument should either be the string '`deg`' or '`rad`'. Parameters affected are θ, ϕ, α, β, and γ.

Examples

To set an isotropic diffusion tensor with a correlation time of 10 ns, assigning it to the run '`m1`', type:

```
relax> diffusion_tensor('m1', 10e-9)
```

```
relax> diffusion_tensor(run='m1', params=10e-9)
```

```
relax> diffusion_tensor('m1', 10.0, 1e-9)
```

```
relax> diffusion_tensor(run='m1', params=10.0, time_scale=1e-9, fixed=1)
```

To select axially symmetric diffusion with a τ_m value of 8.5 ns, $\mathfrak{D}_{ratio}$ of 1.1, θ value of 20 degrees, and ϕ value of 20 degrees, and assign it to the run '`m8`', type:

```
relax> diffusion_tensor('m8', (8.5e-9, 1.1, 20.0, 20.0), param_types=1)
```

To select a spheroid diffusion tensor with a $\mathfrak{D}_{\parallel}$ value of 1.698e7, $\mathfrak{D}_{\perp}$ value of 1.417e7, θ value of 67.174 degrees, and ϕ value of -83.718 degrees, and assign it to the run '`spheroid`', type one of:

```
relax> diffusion_tensor('spheroid', (1.698e7, 1.417e7, 67.174, -83.718), param_types=1)
```

```
relax> diffusion_tensor(run='spheroid', params=(1.698e7, 1.417e7, 67.174, -83.718),
param_types=1)
```

```
relax> diffusion_tensor('spheroid', (1.698e-1, 1.417e-1, 67.174, -83.718), param_types=1,
d_scale=1e8)
```

```
relax> diffusion_tensor(run='spheroid', params=(1.698e-1, 1.417e-1, 67.174, -83.718),
param_types=1, d_scale=1e8)
```

```
relax> diffusion_tensor('spheroid', (1.698e-1, 1.417e-1, 1.1724, -1.4612), param_types=1,
d_scale=1e8, angle_units='rad')
```

```
relax> diffusion_tensor(run='spheroid', params=(1.698e-1, 1.417e-1, 1.1724, -1.4612),
param_types=1, d_scale=1e8, angle_units='rad', fixed=1)
```

To select ellipsoidal diffusion, type:

```
relax> diffusion_tensor('m5', (1.340e7, 1.516e7, 1.691e7, -82.027, -80.573, 65.568),
param_types=2)
```

To select and minimise a spherical diffusion tensor, type (followed by a minimisation command):

```
relax> diffusion_tensor('diff', 10e-9, fixed=0)
```

6.2.13 dx.execute

Synopsis

Function for running OpenDX.

Defaults

dx.execute(self, file='map', dir='dx', dx_exe='dx', vp_exec=1)

Keyword Arguments

file: The file name prefix. For example if file is set to '`temp`', then the OpenDX program temp.net will be loaded.

dir: The directory to change to for running OpenDX. If this is set to '`None`', OpenDX will be run in the current directory.

δ_x_exe: The OpenDX executable file.

vp_exec: A flag specifying whether to execute the visual program automatically at start-up. The default is 1 which turns execution on. Setting the value to zero turns execution off.

6.2.14 dx.map

Synopsis

Function for creating a map of the given space in OpenDX format.

Defaults

dx.map(self, run=None, params=None, map_type='Iso3D', res_num=None, inc=20, lower=None, upper=None, axis_incs=5, file='map', dir='dx', point=None, point_file='point', remap=None)

Keyword Arguments

run: The name of the run.

params: The parameters to be mapped. This argument should be an array of strings, values of which are described below.

map_type: The type of map to create. For example the default, a 3D isosurface, the type is '`Iso3D`'. See below for more details.

res_num: The residue number.

inc: The number of increments to map in each dimension. This value controls the resolution of the map.

lower: The lower bounds of the space. If you wish to change the lower bounds of the map then supply an array of length equal to the number of parameters in the model. A lower bound for each parameter must be supplied. If nothing is supplied then the defaults will be used.

upper: The upper bounds of the space. If you wish to change the upper bounds of the map then supply an array of length equal to the number of parameters in the model. A upper bound for each parameter must be supplied. If nothing is supplied then the defaults will be used.

axis_incs: The number of increments or ticks displaying parameter values along the axes of the OpenDX plot.

file: The file name. All the output files are prefixed with this name. The main file containing the data points will be called the value of '`file`'. The OpenDX program will be called '`file.net`' and the OpenDX import file will be called '`file.general`'.

dir: The directory to output files to. Set this to '`None`' if you do not want the files to be placed in subdirectory. If the directory does not exist, it will be created.

point: An array of parameter values where a point in the map, shown as a red sphere, will be placed. The length must be equal to the number of parameters.

point_file: The name of that the point output files will be prefixed with.

remap: A user supplied remapping function. This function will receive the parameter array and must return an array of equal length.

Map type

The map type can be changed by supplying the '`map_type`' keyword argument. Here is a list of currently supported map types:

Surface type	Pattern
3D isosurface	'`^[Ii]so3[Dd]`'

Pattern syntax is simply regular expression syntax where square brackets '`[]`' means any character within the brackets, '`^`' means the start of the string, etc.

Examples

The following commands will generate a map of the extended model-free space defined as run '`m5`' which consists of the parameters $\{S^2, S_f^2, \tau_s\}$. Files will be output into the directory '`dx`' and will be prefixed by '`map`'. The residue, in this case, is number 6.

```
relax> dx.map('m5', ['S2', 'S2f', 'ts'], 6)

relax> dx.map('m5', ['S2', 'S2f', 'ts'], 6, 20, 'map', 'dx')

relax> dx.map('m5', ['S2', 'S2f', 'ts'], res_num=6, file='map', dir='dx')

relax> dx.map(run='m5', params=['S2', 'S2f', 'ts'], res_num=6, inc=20, file='map',
dir='dx')

relax> dx.map(run='m5', params=['S2', 'S2f', 'ts'], res_num=6, type='Iso3D', inc=20,
swap=[0, 1, 2], file='map', dir='dx')
```

To map the model-free space '`m4`' defined by the parameters $\{S^2, \tau_e, R_{ex}\}$, name the results '`test`', and not place the files in a subdirectory, use the following commands (assuming residue 2).

```
relax> dx.map('m4', ['S2', 'te', 'Rex'], res_num=2, file='test', dir=None)

relax> dx.map(run='m4', params=['S2', 'te', 'Rex'], res_num=2, inc=100, file='test',
dir=None)
```

Regular expression

The python function '`match`', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

‘`[]`’ – A sequence or set of characters to match to a single character. For example, ‘`[Ss]2`’ will match both ‘`S2`’ and ‘`s2`’.

‘`^`’ – Match the start of the string.

‘`$`’ – Match the end of the string. For example, ‘`^[Ss]2$`’ will match ‘`s2`’ but not ‘`S2f`’ or ‘`s2s`’.

‘`.`’ – Match any character.

‘`x*`’ – Match the character ‘`x`’ any number of times, for example ‘`x`’ will match, as will ‘`xxxxx`’

‘`.*`’ – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Diffusion tensor parameter string matching patterns

Data type	Object name	Patterns
Global correlation time - τ_m	'tm'	'tm'
Isotropic component of the diffusion tensor - $\mathfrak{D}_{iso}$	'Diso'	'[Dd]iso'
Anisotropic component of the diffusion tensor - $\mathfrak{D}_a$	'Da'	'[Dd]a'
Rhombic component of the diffusion tensor - $\mathfrak{D}_r$	'Dr'	'[Dd]r$'
Eigenvalue associated with the x-axis of the diffusion diffusion tensor - $\mathfrak{D}_x$	'Dx'	'[Dd]x'
Eigenvalue associated with the y-axis of the diffusion diffusion tensor - $\mathfrak{D}_y$	'Dy'	'[Dd]y'
Eigenvalue associated with the z-axis of the diffusion diffusion tensor - $\mathfrak{D}_z$	'Dz'	'[Dd]z'
Diffusion coefficient parallel to the major axis of the spheroid diffusion tensor - $\mathfrak{D}_{\parallel}$	'Dpar'	'[Dd]par'
Diffusion coefficient perpendicular to the major axis of the spheroid diffusion tensor - $\mathfrak{D}_{\perp}$	'Dper'	'[Dd]per'
Ratio of the parallel and perpendicular components of the spheroid diffusion tensor - $\mathfrak{D}_{ratio}$	'Dratio'	'[Dd]ratio'
The first Euler angle of the ellipsoid diffusion tensor - α	'alpha'	'^a$' or 'alpha'
The second Euler angle of the ellipsoid diffusion tensor - β	'beta'	'^b$' or 'beta'
The third Euler angle of the ellipsoid diffusion tensor - γ	'gamma'	'^g$' or 'gamma'
The polar angle defining the major axis of the spheroid diffusion tensor - θ	'theta'	'theta'
The azimuthal angle defining the major axis of the spheroid diffusion tensor - ϕ	'phi'	'phi'

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

6.2.15 eliminate

Synopsis

Function for model elimination.

Defaults

eliminate(self, run=None, function=None, args=None)

Keyword arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

function: A user supplied function for model elimination.

args: A tuple of arguments for model elimination.

Description

This function is used for model validation to eliminate or reject models prior to model selection. Model validation is a part of mathematical modelling whereby models are either accepted or rejected.

Empirical rules are used for model rejection and are listed below. However these can be overridden by supplying a function. The function should accept five arguments, a string defining a certain parameter, the value of the parameter, the run name, the minimisation instance (ie the residue index if the model is residue specific), and the function arguments. If the model is rejected, the function should return 1, otherwise it should return 0. The function will be executed multiple times, once for each parameter of the model.

The '`args`' keyword argument should be a tuple, a list enclosed in round brackets, and will be passed to the user supplied function or the inbuilt function. For a description of the arguments accepted by the inbuilt functions, see below.

Once a model is rejected, the select flag corresponding to that model will be set to 0 so that model selection, or any other function, will then skip the model.

Local τ_m model elimination rule

The local τ_m, in some cases, may exceed the value expected for a global correlation time. Generally the τ_m value will be stuck at the upper limit defined for the parameter. These models are eliminated using the rule:

$$\tau_m \geq \mathrm{c}$$

The default value of c is 50 ns, although this can be overridden by supplying the value (in seconds) as the first element of the args tuple.

Internal correlation times {τ_e, τ_f, τ_s} model elimination rules

These parameters may experience the same problem as the local τ_m in that the model fails and the parameter value is stuck at the upper limit. These parameters are constrained using the formula (τ_e, τ_f, $\tau_s \leq 2\tau_m$). These failed models are eliminated using the rule:

$$\tau_e,\ \tau_f,\ \tau_s \geq \mathrm{c}\ .\ \tau_m$$

The default value of c is 1.5. Because of round-off errors and the constraint algorithm, setting c to 2 will result in no models being eliminated as the minimised parameters will always be less than $2\tau_m$. The value can be changed by supplying the value as the second element of the tuple.

Arguments

The '`args`' argument must be a tuple of length 2, the elements of which must be numbers. For example, to eliminate models which have a local τ_m value greater than 25 ns and models with internal correlation times greater than 1.5 times τ_m, set '`args`' to (25 * 1e-9, 1.5).

6.2.16 fix

Synopsis

Function for either fixing or allowing parameter values to change.

Defaults

fix(self, run=None, element=None, fixed=1)

Keyword Arguments

run: The name of the run.

element: Which element to fix.

fixed: A flag specifying if the parameters should be fixed or allowed to change.

Description

The keyword argument '`element`' can be any of the following:

'`diff`' - the diffusion tensor parameters. This will allow all diffusion tensor parameters to be toggled.

an integer - if an integer number is given, then all parameters for the residue corresponding to that number will be toggled.

'`all_res`' - using this keyword, all parameters from all residues will be toggled.

'`all`' - all parameter will be toggled. This is equivalent to combining both '`diff`' and '`all_res`'.

The flag '`fixed`', if set to 1, will fix parameters, while a value of 0 will allow parameters to vary.

Only parameters corresponding to the given run will be affected.

6.2.17 grace.view

Synopsis

Function for running Grace.

Defaults

grace.view(self, file=None, dir='grace', grace_exe='xmgrace')

Keyword Arguments

file: The name of the file.

dir: The directory name.

grace_exe: The Grace executable file.

Description

This function can be used to execute Grace to view the specified file the Grace '`.agr`' file and the execute Grace. If the directory name is set to None, the file will be assumed to be in the current working directory.

Examples

To view the file '`s2.agr`' in the directory '`grace`', type:

```
relax> grace.view(file='s2.agr')
relax> grace.view(file='s2.agr', dir='grace')
```

6.2.18 grace.write

Synopsis

Function for creating a grace '`.agr`' file.

Defaults

grace.write(self, run=None, x_data_type='res', y_data_type=None, res_num=None, res_name=None, plot_data='value', file=None, dir='grace', force=0)

Keyword Arguments

run: The name of the run.

x_data_type: The data type for the X-axis (no regular expression is allowed).

y_data_type: The data type for the Y-axis (no regular expression is allowed).

res_num: The residue number (regular expression is allowed).

res_name: The residue name (regular expression is allowed).

plot_data: The data to use for the plot.

file: The name of the file.

dir: The directory name.

force: A flag which, if set to 1, will cause the file to be overwritten.

Description

This function is designed to be as flexible as possible so that any combination of data can be plotted. The output is in the format of a Grace plot (also known as ACE/gr, Xmgr, and xmgrace) which only supports two dimensional plots. Three types of keyword arguments can be used to create various types of plot. These include the X-axis and Y-axis data types, the residue number and name selection arguments, and an argument for selecting what to actually plot.

The X-axis and Y-axis data type arguments should be plain strings, regular expression is not allowed. If the X-axis data type argument is not given, the plot will default to having the residue number along the x-axis. The two axes of the Grace plot can be absolutely any of the data types listed in the tables below. The only limitation, currently anyway, is that the data must belong to the same run.

The residue number and name arguments can be used to limit the residues used in the plot. The default is that all residues will be used, however, these arguments can be used to select a subset of all residues, or a single residue for plots of Monte Carlo simulations,

etc. Regular expression is allowed for both the residue number and name, and the number can either be an integer or a string.

The property which is actually plotted can be controlled by the 'plot_data' argument. It can be one of the following:

'value' – Plot values (with errors if they exist).

'error' – Plot errors.

'sims' – Plot the simulation values.

Examples

To write the NOE values for all residues from the run 'noe' to the Grace file 'noe.agr', type:

```
relax> grace.write('noe', 'res', 'noe', file='noe.agr')
relax> grace.write('noe', y_data_type='noe', file='noe.agr')
relax> grace.write('noe', x_data_type='res', y_data_type='noe', file='noe.agr')
relax> grace.write(run='noe', y_data_type='noe', file='noe.agr', force=1)
```

To create a Grace file of 'S2' vs. 'te' for all residues, type:

```
relax> grace.write('m2', 'S2', 'te', file='s2_te.agr')
relax> grace.write('m2', x_data_type='S2', y_data_type='te', file='s2_te.agr')
relax> grace.write(run='m2', x_data_type='S2', y_data_type='te', file='s2_te.agr', force=1)
```

To create a Grace file of the Monte Carlo simulation values of 'Rex' vs. 'te' for residue 123, type:

```
relax> grace.write('m4', 'Rex', 'te', res_num=123, plot_data='sims', file='s2_te.agr')
relax> grace.write(run='m4', x_data_type='Rex', y_data_type='te', res_num=123,
plot_data='sims', file='s2_te.agr')
```

Regular expression

The python function 'match', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'[]' – A sequence or set of characters to match to a single character. For example, '[Ss]2' will match both 'S2' and 's2'.

'^' – Match the start of the string.

'$' – Match the end of the string. For example, '`^[Ss]2$`' will match '`s2`' but not '`S2f`' or '`s2s`'.

'.' – Match any character.

'x*' – Match the character '`x`' any number of times, for example '`x`' will match, as will '`xxxxx`'

'.*' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Minimisation statistic data type string matching patterns

Data type	Object name	Patterns
Chi-squared statistic	'`chi2`'	'`^[Cc]hi2$`' or '`^[Cc]hi[-_ ][Ss]quare`'
Iteration count	'`iter`'	'`^[Ii]ter`'
Function call count	'`f_count`'	'`^[Ff].*[ -_][Cc]ount`'
Gradient call count	'`g_count`'	'`^[Gg].*[ -_][Cc]ount`'
Hessian call count	'`h_count`'	'`^[Hh].*[ -_][Cc]ount`'

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

NOE calculation data type string matching patterns

Data type	Object name	Patterns
Reference intensity	'ref'	'^[Rr]ef$' or '[Rr]ef[-_][Ii]nt'
Saturated intensity	'sat'	'^[Ss]at$' or '[Ss]at[-_][Ii]nt'
NOE	'noe'	'^[Nn][Oo][Ee]$'

6.2.19 grid_search

Synopsis

The grid search function.

Defaults

grid_search(self, run=None, lower=None, upper=None, inc=21, constraints=1, print_flag=1)

Keyword Arguments

run: The name of the run to apply the grid search to.

lower: An array of the lower bound parameter values for the grid search. The length of the array should be equal to the number of parameters in the model.

upper: An array of the upper bound parameter values for the grid search. The length of the array should be equal to the number of parameters in the model.

inc: The number of increments to search over. If a single integer is given then the number of increments will be equal in all dimensions. Different numbers of increments in each direction can be set if '`inc`' is set to an array of integers of length equal to the number of parameters.

constraints: A flag specifying whether the parameters should be constrained. The default is to turn constraints on (constraints=1).

print_flag: The amount of information to print to screen. Zero corresponds to minimal output while higher values increase the amount of output. The default value is 1.

6.2.20 init_data

Synopsis

Function for reinitialising self.relax.data

Defaults

init_data(self)

6.2.21 intro_off

Synopsis

Function for turning the function introductions off.

Defaults

intro_off(self)

6.2.22 intro_on

Synopsis

Function for turning the function introductions on.

Defaults

intro_on(self)

6.2.23 jw_mapping.set_frq

Synopsis

Function for selecting which relaxation data to use in the $J(\omega)$ mapping.

Defaults

jw_mapping.set_frq(self, run=None, frq=None)

Keyword Arguments

run: The name of the run.

frq: The spectrometer frequency in Hz.

Description

This function will select the relaxation data to use in the reduced spectral density mapping corresponding to the given frequency.

Examples

```
relax> jw_mapping.set_frq('jw', 600.0 * 1e6)

relax> jw_mapping.set_frq(run='jw', frq=600.0 * 1e6)
```

6.2.24 minimise

Synopsis

Minimisation function.

Defaults

minimise(self, *args, **keywords)

Arguments

The arguments, which should all be strings, specify the minimiser as well as its options. A minimum of one argument is required. As this calls the function '`generic_minimise`' the full list of allowed arguments is shown below in the reproduced '`generic_minimise`' docstring. Ignore all sections except those labelled as minimisation algorithms and minimisation options. Also do not select the Method of Multipliers constraint algorithm as this is used in combination with the given minimisation algorithm if the keyword argument '`constraints`' is set to 1. The grid search algorithm should also not be selected as this is accessed using the '`grid`' function instead. The first argument passed will be set to the minimisation algorithm while all other arguments will be set to the minimisation options.

Keyword arguments differ from normal arguments having the form '`keyword = value`'. All arguments must precede keyword arguments in python. For more information see the examples section below or the python tutorial.

Keyword Arguments

run: The name of the run.

func_tol: The function tolerance. This is used to terminate minimisation once the function value between iterations is less than the tolerance. The default value is 1e-25.

grad_tol: The gradient tolerance. Minimisation is terminated if the current gradient value is less than the tolerance. The default value is None.

max_iterations: The maximum number of iterations. The default value is 1e7.

constraints: A flag specifying whether the parameters should be constrained. The default is to turn constraints on (constraints=1).

scaling: The diagonal scaling flag. The default that scaling is on (scaling=1).

print_flag: The amount of information to print to screen. Zero corresponds to minimal output while higher values increase the amount of output. The default value is 1.

Diagonal scaling

Diagonal scaling is the transformation of parameter values such that each value has a similar order of magnitude. Certain minimisation techniques, for example the trust region methods, perform extremely poorly with badly scaled problems. In addition, methods which are insensitive to scaling such as Newton minimisation may still benefit due to the minimisation of round off errors.

In Model-free analysis for example, if $S^2 = 0.5$, $\tau_e = 200$ ps, and $R_{ex} = 15$ 1/s at 600 MHz, the unscaled parameter vector would be [0.5, 2.0e-10, 1.055e-18]. R_{ex} is divided by (2 * π * 600,000,000)**2 to make it field strength independent. The scaling vector for this model may be something like [1.0, 1e-9, 1/(2 * π * 6e8)**2]. By dividing the unscaled parameter vector by the scaling vector the scaled parameter vector is [0.5, 0.2, 15.0]. To revert to the original unscaled parameter vector, the scaled parameter vector and scaling vector are multiplied.

Examples

To minimise the model-free run '`m4`' using Newton minimisation together with the GMW81 Hessian modification algorithm, the More and Thuente line search algorithm, a function tolerance of 1e-25, no gradient tolerance, a maximum of 10,000,000 iterations, constraints turned on to limit parameter values, and have normal printout, type any combination of:

```
relax> minimise('newton', run='m4')

relax> minimise('Newton', run='m4')

relax> minimise('newton', 'gmw', run='m4')

relax> minimise('newton', 'mt', run='m4')

relax> minimise('newton', 'gmw', 'mt', run='m4')

relax> minimise('newton', 'mt', 'gmw', run='m4')

relax> minimise('newton', run='m4', func_tol=1e-25)

relax> minimise('newton', run='m4', func_tol=1e-25, grad_tol=None)

relax> minimise('newton', run='m4', max_iter=1e7)

relax> minimise('newton', run=name, constraints=1, max_iter=1e7)

relax> minimise('newton', run='m4', print_flag=1)
```

To minimise the model-free run '`m5`' using constrained Simplex minimisation with a maximum of 5000 iterations, type:

```
relax> minimise('simplex', run='m5', constraints=1, max_iter=5000)
```

Note

```
All the text which follows is a reproduction of the docstring of the generic_minimise
function.  Only take note of the minimisation algorithms and minimisation options sections,
the other sections are not relevant for this function.  The Grid search and Method of
```

```
Multipliers algorithms CANNOT be selected as minimisation algorithms for this function.

The section entitled Keyword Arguments is also completely inaccessible therefore please
ignore that text.
```

Generic minimisation function.

This is a generic function which can be used to access all minimisers using the same set of function arguments. These are the function tolerance value for convergence tests, the maximum number of iterations, a flag specifying which data structures should be returned, and a flag specifying the amount of detail to print to screen.

Keyword Arguments

func: The function which returns the value.

dfunc: The function which returns the gradient.

d2func: The function which returns the Hessian.

args: The tuple of arguments to supply to the functions func, dfunc, and d2func.

x0: The vector of initial parameter value estimates (as an array).

min_algor: A string specifying which minimisation technique to use.

min_options: A tuple to pass to the minimisation function as the min_options keyword.

func_tol: The function tolerance value. Once the function value between iterations decreases below this value, minimisation is terminated.

grad_tol: The gradient tolerance value.

maxiter: The maximum number of iterations.

A: Linear constraint matrix m*n (A.x $\geq b$).

b: Linear constraint scalar vector (A.x $\geq b$).

l: Lower bound constraint vector ($l \leq x \leq u$).

u: Upper bound constraint vector ($l \leq x \leq u$).

c: User supplied constraint function.

dc: User supplied constraint gradient function.

d2c: User supplied constraint Hessian function.

full_output: A flag specifying which data structures should be returned.

print_flag: A flag specifying how much information should be printed to standard output during minimisation. 0 means no output, 1 means minimal output, and values above 1 increase the amount of output printed.

Minimisation output

The following values of the '`full_output`' flag will return, in tuple form, the following data

0 – '`xk`',

1 – '`(xk, fk,` k`, f_count, g_count, h_count, warning)`',

where the data names correspond to

'`xk`' – The array of minimised parameter values,

'`fk`' – The minimised function value,

'`k`' – The number of iterations,

'`f_count`' – The number of function calls,

'`g_count`' – The number of gradient calls,

'`h_count`' – The number of Hessian calls,

'`warning`' – The warning string.

Minimisation algorithms

A minimisation function is selected if the minimisation algorithm argument, which should be a string, matches a certain pattern. Because the python regular expression '`match`' statement is used, various strings can be supplied to select the same minimisation algorithm. Below is a list of the minimisation algorithms available together with the corresponding patterns.

This is a short description of python regular expression, for more information, see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'`[]`' – A sequence or set of characters to match to a single character. For example, '`[Nn]ewton`' will match both '`Newton`' and '`newton`'.

'`^`' – Match the start of the string.

'`$`' – Match the end of the string. For example, '`^[Ll][Mm]$`' will match '`lm`' and '`LM`' but will not match if characters are placed either before or after these strings.

To select a minimisation algorithm, set the argument to a string which matches the given pattern.

Parameter initialisation methods:

Minimisation algorithm	Patterns
Grid search	'`^[Gg]rid`'

Unconstrained line search methods:

Minimisation algorithm	Patterns
Back-and-forth coordinate descent	'`^[Cc][Dd]$`' or '`^[Cc]oordinate[ _-][Dd]escent$`'
Steepest descent	'`^[Ss][Dd]$`' or '`^[Ss]teepest[ _-][Dd]escent$`'
Quasi-Newton BFGS	'`^[Bb][Ff][Gg][Ss]$`'
Newton	'`^[Nn]ewton$`'
Newton-CG	'`^[Nn]ewton[ _-][Cc][Gg]$`' or '`^[Nn][Cc][Gg]$`'

Unconstrained trust-region methods:

Minimisation algorithm	Patterns
Cauchy point	'`^[Cc]auchy`'
Dogleg	'`^[Dd]ogleg`'
CG-Steihaug	'`^[Cc][Gg][-_ ][Ss]teihaug`' or '`^[Ss]teihaug`'
Exact trust region	'`^[Ee]xact`'

Unconstrained conjugate gradient methods:

Minimisation algorithm	Patterns
Fletcher-Reeves	'`^[Ff][Rr]$`' or '`^[Ff]letcher[-_ ][Rr]eeves$`'
Polak-Ribière	'`^[Pp][Rr]$`' or '`^[Pp]olak[-_ ][Rr]ibiere$`'
Polak-Ribière +	'`^[Pp][Rr]\+$`' or '`^[Pp]olak[-_ ][Rr]ibiere\+$`'
Hestenes-Stiefel	'`^[Hh][Ss]$`' or '`^[Hh]estenes[-_ ][Ss]tiefel$`'

Miscellaneous unconstrained methods:

Minimisation algorithm	Patterns
Simplex	'`^[Ss]implex$`'
Levenberg-Marquardt	'`^[Ll][Mm]$`' or '`^[Ll]evenburg-[Mm]arquardt$`'

Constrained methods:

Minimisation algorithm	Patterns
Method of Multipliers	'`^[Mm][Oo][Mm]$`' or '`[Mm]ethod of [Mm]ultipliers$`'

Minimisation options

The minimisation options can be given in any order.

Line search algorithms. These are used in the line search methods and the conjugate gradient methods. The default is the Backtracking line search.

Line search algorithm	Patterns
Backtracking line search	'`^[Bb]ack`'
Nocedal and Wright interpolation based line search	'`^[Nn][Ww][Ii]`' or '`^[Nn]ocedal[ _][Ww]right[ _][Ii]nt`'
Nocedal and Wright line search for the Wolfe conditions	'`^[Nn][Ww][Ww]`' or '`^[Nn]ocedal[ _][Ww]right[ _][Ww]olfe`'
More and Thuente line search	'`^[Mm][Tt]`' or '`^[Mm]ore[ _][Tt]huente$`'
No line search	'`^[Nn]one$`'

Hessian modifications. These are used in the Newton, Dogleg, and Exact trust region algorithms.

Hessian modification	Patterns
Unmodified Hessian	'`[Nn]one`'
Eigenvalue modification	'`^[Ee]igen`'
Cholesky with added multiple of the identity	'`^[Cc]hol`'
The Gill, Murray, and Wright modified Cholesky algorithm	'`^[Gg][Mm][Ww]$`'
The Schnabel and Eskow 1999 algorithm	'`^[Ss][Ee]99`'

Hessian type, these are used in a few of the trust region methods including the Dogleg and Exact trust region algorithms. In these cases, when the Hessian type is set to Newton, a Hessian modification can also be supplied as above. The default Hessian type is Newton, and the default Hessian modification when Newton is selected is the GMW algorithm.

Hessian type	Patterns
Quasi-Newton BFGS	'^[Bb][Ff][Gg][Ss]$'
Newton	'^[Nn]ewton$'

For Newton minimisation, the default line search algorithm is the More and Thuente line search, while the default Hessian modification is the GMW algorithm.

6.2.25 model_free.copy

Synopsis

Function for copying model-free data from run1 to run2.

Defaults

model_free.copy(self, run1=None, run2=None, sim=None)

Keyword Arguments

run1: The name of the run to copy the sequence from.

run2: The name of the run to copy the sequence to.

sim: The simulation number.

Description

This function will copy all model-free data from '`run1`' to '`run2`'. Any model-free data in '`run2`' will be overwritten. If the argument '`sim`' is an integer, then only data from that simulation will be copied.

Examples

To copy all model-free data from the run '`m1`' to the run '`m2`', type:

```
relax> model_free.copy('m1', 'm2')
relax> model_free.copy(run1='m1', run2='m2')
```

6.2.26 model_free.create_model

Synopsis

Function to create a model-free model.

Defaults

model_free.create_model(self, run=None, model=None, equation=None, params=None, res_num=None)

Keyword Arguments

run: The run to assign the values to.

model: The name of the model-free model.

equation: The model-free equation.

params: The array of parameter names of the model.

res_num: The residue number.

Model-free equation

'mf_orig' selects the original model-free equations with parameters $\{S^2, \tau_e\}$. 'mf_ext' selects the extended model-free equations with parameters $\{S_f^2, \tau_f, S^2, \tau_s\}$. 'mf_ext2' selects the extended model-free equations with parameters $\{S_f^2, \tau_f, S_s^2, \tau_s\}$.

Model-free parameters

The following parameters are accepted for the original model-free equation:

'S2' – The square of the generalised order parameter.

'te' – The effective correlation time.

The following parameters are accepted for the extended model-free equation:

'S2f' – The square of the generalised order parameter of the faster motion.

'tf' – The effective correlation time of the faster motion.

'S2' – The square of the generalised order parameter $S^2 = S_f^2 * S_s^2$.

'ts' – The effective correlation time of the slower motion.

The following parameters are accepted for the extended 2 model-free equation:

'S2f' – The square of the generalised order parameter of the faster motion.

'tf' – The effective correlation time of the faster motion.

'S2s' – The square of the generalised order parameter of the slower motion.

'ts' – The effective correlation time of the slower motion.

The following parameters are accepted for all equations:

'Rex' – The chemical exchange relaxation.

'r' – The average bond length $<r>$.

'CSA' – The chemical shift anisotropy.

Residue number

If 'res_num' is supplied as an integer then the model will only be created for that residue, otherwise the model will be created for all residues.

Examples

The following commands will create the model-free model 'm1' which is based on the original model-free equation and contains the single parameter 'S2'.

```
relax> model_free.create_model('m1', 'm1', 'mf_orig', ['S2'])
```

```
relax> model_free.create_model(run='m1', model='m1', params=['S2'], equation='mf_orig')
```

The following commands will create the model-free model 'large_model' which is based on the extended model-free equation and contains the seven parameters 'S2f', 'tf', 'S2', 'ts', 'Rex', 'CSA', 'r'.

```
relax> model_free.create_model('test', 'large_model', 'mf_ext', ['S2f', 'tf', 'S2', 'ts',
'Rex', 'CSA', 'r'])
```

```
relax> model_free.create_model(run='test', model='large_model', params=['S2f', 'tf', 'S2',
'ts', 'Rex', 'CSA', 'r'], equation='mf_ext')
```

6.2.27 model_free.delete

Synopsis

Function for deleting all model-free data corresponding to the run.

Defaults

model_free.delete(self, run=None)

Keyword Arguments

run: The name of the run.

Examples

To delete all model-free data corresponding to the run 'm2', type:

```
relax> model_free.delete('m2')
```

6.2.28 model_free.remove_tm

Synopsis

Function for removing the local τ_m parameter from a model.

Defaults

model_free.remove_tm(self, run=None, res_num=None)

Keyword Arguments

run: The run to assign the values to.

res_num: The residue number.

Description

This function will remove the local τ_m parameter from the model-free parameters of the given run. Model-free parameters must already exist within the run yet, if there is no local τ_m, nothing will happen.

If no residue number is given, then the function will apply to all residues.

Examples

The following commands will remove the parameter '`tm`' from the run '`local_tm`':

```
relax> model_free.remove_tm('local_tm')

relax> model_free.remove_tm(run='local_tm')
```

6.2.29 model_free.select_model

Synopsis

Function for the selection of a preset model-free model.

Defaults

model_free.select_model(self, run=None, model=None, res_num=None)

Keyword Arguments

run: The run to assign the values to.

model: The name of the preset model.

The preset models

The standard preset model-free models are

'm0' – [],

'm1' – [S^2],

'm2' – [S^2, τ_e],

'm3' – [S^2, R_{ex}],

'm4' – [S^2, τ_e, R_{ex}],

'm5' – [S_f^2, S^2, τ_s],

'm6' – [S_f^2, τ_f, S^2, τ_s],

'm7' – [S_f^2, S^2, τ_s, R_{ex}],

'm8' – [S_f^2, τ_f, S^2, τ_s, R_{ex}],

'm9' – [R_{ex}].

The preset model-free models with optimisation of the CSA value are

'm10' – [CSA],

'm11' – [CSA, S^2],

'm12' – [CSA, S^2, τ_e],

'm13' – [CSA, S^2, R_{ex}],

'm14' – [CSA, S^2, τ_e, R_{ex}],

'm15' – [CSA, S_f^2, S^2, τ_s],

'm16' – [CSA, S_f^2, τ_f, S^2, τ_s],

'm17' – [CSA, S_f^2, S^2, τ_s, R_{ex}],

'm18' – [CSA, S_f^2, τ_f, S^2, τ_s, R_{ex}],

'm19' – [CSA, R_{ex}].

The preset model-free models with optimisation of the bond length are

'm20' – [r],

'm21' – [r, S^2],

'm22' – [r, S^2, τ_e],

'm23' – [r, S^2, R_{ex}],

'm24' – [r, S^2, τ_e, R_{ex}],

'm25' – [r, S_f^2, S^2, τ_s],

'm26' – [r, S_f^2, τ_f, S^2, τ_s],

'm27' – [r, S_f^2, S^2, τ_s, R_{ex}],

'm28' – [r, S_f^2, τ_f, S^2, τ_s, R_{ex}],

'm29' – [r, CSA, R_{ex}].

The preset model-free models with both optimisation of the bond length and CSA are

'm30' – [r, CSA],

'm31' – [r, CSA, S^2],

'm32' – [r, CSA, S^2, τ_e],

'm33' – [r, CSA, S^2, R_{ex}],

'm34' – [r, CSA, S^2, τ_e, R_{ex}],

'm35' – [r, CSA, S_f^2, S^2, τ_s],

'm36' – [r, CSA, S_f^2, τ_f, S^2, τ_s],

'm37' – [r, CSA, S_f^2, S^2, τ_s, R_{ex}],

'm38' – [r, CSA, S_f^2, τ_f, S^2, τ_s, R_{ex}],

'm39' – [r, CSA, R_{ex}].

Warning: The models in the thirties range fail when using standard R_1, R_1, and NOE relaxation data. This is due to the extreme flexibly of these models where a change in the parameter '`r`' is compensated by a corresponding change in the parameter '`CSA`' and vice versa.

Additional preset model-free models, which are simply extensions of the above models with the addition of a local τ_m parameter are:

'`tm0`' – $[\tau_m]$,

'`tm1`' – $[\tau_m, S^2]$,

'`tm2`' – $[\tau_m, S^2, \tau_e]$,

'`tm3`' – $[\tau_m, S^2, R_{ex}]$,

'`tm4`' – $[\tau_m, S^2, \tau_e, R_{ex}]$,

'`tm5`' – $[\tau_m, S_f^2, S^2, \tau_s]$,

'`tm6`' – $[\tau_m, S_f^2, \tau_f, S^2, \tau_s]$,

'`tm7`' – $[\tau_m, S_f^2, S^2, \tau_s, R_{ex}]$,

'`tm8`' – $[\tau_m, S_f^2, \tau_f, S^2, \tau_s, R_{ex}]$,

'`tm9`' – $[\tau_m, R_{ex}]$.

The preset model-free models with optimisation of the CSA value are

'`tm10`' – $[\tau_m, \text{CSA}]$,

'`tm11`' – $[\tau_m, \text{CSA}, S^2]$,

'`tm12`' – $[\tau_m, \text{CSA}, S^2, \tau_e]$,

'`tm13`' – $[\tau_m, \text{CSA}, S^2, R_{ex}]$,

'`tm14`' – $[\tau_m, \text{CSA}, S^2, \tau_e, R_{ex}]$,

'`tm15`' – $[\tau_m, \text{CSA}, S_f^2, S^2, \tau_s]$,

'`tm16`' – $[\tau_m, \text{CSA}, S_f^2, \tau_f, S^2, \tau_s]$,

'`tm17`' – $[\tau_m, \text{CSA}, S_f^2, S^2, \tau_s, R_{ex}]$,

'`tm18`' – $[\tau_m, \text{CSA}, S_f^2, \tau_f, S^2, \tau_s, R_{ex}]$,

'`tm19`' – $[\tau_m, \text{CSA}, R_{ex}]$.

The preset model-free models with optimisation of the bond length are

'`tm20`' – $[\tau_m, r]$,

'tm21' – $[\tau_m, r, S^2]$,

'tm22' – $[\tau_m, r, S^2, \tau_e]$,

'tm23' – $[\tau_m, r, S^2, R_{ex}]$,

'tm24' – $[\tau_m, r, S^2, \tau_e, R_{ex}]$,

'tm25' – $[\tau_m, r, S_f^2, S^2, \tau_s]$,

'tm26' – $[\tau_m, r, S_f^2, \tau_f, S^2, \tau_s]$,

'tm27' – $[\tau_m, r, S_f^2, S^2, \tau_s, R_{ex}]$,

'tm28' – $[\tau_m, r, S_f^2, \tau_f, S^2, \tau_s, R_{ex}]$,

'tm29' – $[\tau_m, r, \text{CSA}, R_{ex}]$.

The preset model-free models with both optimisation of the bond length and CSA are

'tm30' – $[\tau_m, r, \text{CSA}]$,

'tm31' – $[\tau_m, r, \text{CSA}, S^2]$,

'tm32' – $[\tau_m, r, \text{CSA}, S^2, \tau_e]$,

'tm33' – $[\tau_m, r, \text{CSA}, S^2, R_{ex}]$,

'tm34' – $[\tau_m, r, \text{CSA}, S^2, \tau_e, R_{ex}]$,

'tm35' – $[\tau_m, r, \text{CSA}, S_f^2, S^2, \tau_s]$,

'tm36' – $[\tau_m, r, \text{CSA}, S_f^2, \tau_f, S^2, \tau_s]$,

'tm37' – $[\tau_m, r, \text{CSA}, S_f^2, S^2, \tau_s, R_{ex}]$,

'tm38' – $[\tau_m, r, \text{CSA}, S_f^2, \tau_f, S^2, \tau_s, R_{ex}]$,

'tm39' – $[\tau_m, r, \text{CSA}, R_{ex}]$.

Residue number

If 'res_num' is supplied as an integer then the model will only be selected for that residue, otherwise the model will be selected for all residues.

Examples

To pick model 'm1' for all selected residues and assign it to the run 'mixed', type:

```
relax> model_free.select_model('mixed', 'm1')

relax> model_free.select_model(run='mixed', model='m1')
```

6.2.30 model_selection

Synopsis

Function for model selection.

Defaults

model_selection(self, method=None, modsel_run=None, runs=None)

Keyword arguments

method: The model selection technique (see below).

modsel_run: The run name to assign to the results of model selection.

runs: An array containing the names of all runs to include in model selection.

Description

The following model selection methods are supported:

AIC: Akaike's Information Criteria.

AICc: Small sample size corrected AIC.

BIC: Bayesian or Schwarz Information Criteria.

Bootstrap: Bootstrap model selection.

CV: Single-item-out cross-validation.

Expect: The expected overall discrepancy (the true values of the parameters are required).

Farrow: Old model-free method by Farrow et al., 1994.

Palmer: Old model-free method by Mandel et al., 1995.

Overall: The realised overall discrepancy (the true values of the parameters are required).

For the methods '`Bootstrap`', '`Expect`', and '`Overall`', the function '`monte_carlo`' should have previously been run with the type argument set to the appropriate value to modify its behaviour.

If the runs argument is not supplied then all runs currently set or loaded will be used for model selection, although this could cause problems.

Example

For model-free analysis, if the preset models 1 to 5 are minimised and loaded into the program, the following commands will carry out AIC model selection and assign the results to the run name '`mixed`':

```
relax> model_selection('AIC', 'mixed')

relax> model_selection(method='AIC', modsel_run='mixed')

relax> model_selection('AIC', 'mixed', ['m1', 'm2', 'm3', 'm4', 'm5'])

relax> model_selection(method='AIC', modsel_run='mixed', runs=['m1', 'm2', 'm3', 'm4',
'm5'])
```

6.2.31 molmol.clear_history

Synopsis

Function for clearing the Molmol command history.

Defaults

molmol.clear_history(self)

6.2.32 molmol.command

Synopsis

Function for executing a user supplied Molmol command.

Defaults

molmol.command(self, command)

Example

```
relax> molmol.command("InitAll yes")
```

6.2.33 molmol.view

Synopsis

Function for viewing the collection of molecules extracted from the PDB file.

Defaults

molmol.view(self, run=None)

Keyword Arguments

run: The name of the run which the PDB belongs to.

Example

```
relax> molmol.view('m1')
relax> molmol.view(run='pdb')
```

6.2.34 monte_carlo.create_data

Synopsis

Function for creating simulation data.

Defaults

monte_carlo.create_data(self, run=None, method='back_calc')

Keyword Arguments

run: The name of the run.

method: The simulation method.

Description

The method argument can either be set to '`back_calc`' or '`direct`', the choice of which determines the simulation type. If the values or parameters of a run are calculated rather than minimised, this option will have no effect, hence, '`back_calc`' and '`direct`' are identical.

For error analysis, the method argument should be set to '`back_calc`' which will result in proper Monte Carlo simulations. The data used for each simulation is back calculated from the minimised model parameters and is randomised using Gaussian noise where the standard deviation is from the original error set. When the method is set to '`back_calc`', this function should only be called after the model or run is fully minimised.

The simulation type can be changed by setting the method argument to '`direct`'. This will result in simulations which cannot be used in error analysis and which are no longer Monte Carlo simulations. However, these simulations are required for certain model selection techniques (see the documentation for the model selection function for details), and can be used for other purposes. Rather than the data being back calculated from the fitted model parameters, the data is generated by taking the original data and randomising using Gaussian noise with the standard deviations set to the original error set.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search('m1', inc=11) # Step 2.

relax> minimise('newton', run='m1') # Step 2.

relax> monte_carlo.setup('m1', number=500) # Step 3.

relax> monte_carlo.create_data('m1', method='back_calc') # Step 4.

relax> monte_carlo.initial_values('m1') # Step 5.

relax> minimise('newton', run='m1') # Step 6.

relax> eliminate('m1') # Step 7.

relax> monte_carlo.error_analysis('m1') # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc('600MHz') # Step 2.

relax> monte_carlo.setup('600MHz', number=500) # Step 3.
```

```
relax> monte_carlo.create_data(‘600MHz’, method=‘back_calc’) # Step 4.

relax> calc(‘600MHz’) # Step 6.

relax> monte_carlo.error_analysis(‘600MHz’) # Step 8.
```

6.2.35 monte_carlo.error_analysis

Synopsis

Function for calculating parameter errors from the Monte Carlo simulations.

Defaults

monte_carlo.error_analysis(self, run=None, prune=0.0)

Keyword Arguments

run: The name of the run.

prune: Legacy argument corresponding to '`trim`' in Art Palmer's Modelfree program.

Description

Parameter errors are calculated as the standard deviation of the distribution of parameter values. This function should never be used if parameter values are obtained by minimisation and the simulation data are generated using the method '`direct`'. The reason is because only true Monte Carlo simulations can give the true parameter errors.

The prune argument is legacy code which corresponds to the '`trim`' option in Art Palmer's Modelfree program. To remove failed simulations, the eliminate function should be used prior to this function. Eliminating the simulations specifically identifies and removes the failed simulations whereas the prune argument will only, in a few cases, positively identify failed simulations but only if severe parameter limits have been imposed. Most failed models will pass through the pruning process and hence cause a catastrophic increase in the parameter errors. If the argument must be used, the following must be taken into account. If the values or parameters of a run are calculated rather than minimised, the prune argument must be set to zero. The value of this argument is proportional to the number of simulations removed prior to error calculation. If prune is set to 0.0, all simulations are used for calculating errors, whereas a value of 1.0 excludes all data. In almost all cases prune must be set to zero, any value greater than zero will result in an underestimation of the error values. If a value is supplied, the lower and upper tails of the distribution of chi-squared values will be excluded from the error calculation.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search('m1', inc=11) # Step 2.

relax> minimise('newton', run='m1') # Step 2.

relax> monte_carlo.setup('m1', number=500) # Step 3.

relax> monte_carlo.create_data('m1', method='back_calc') # Step 4.

relax> monte_carlo.initial_values('m1') # Step 5.

relax> minimise('newton', run='m1') # Step 6.

relax> eliminate('m1') # Step 7.

relax> monte_carlo.error_analysis('m1') # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc('600MHz') # Step 2.

relax> monte_carlo.setup('600MHz', number=500) # Step 3.
```

```
relax> monte_carlo.create_data('600MHz', method='back_calc') # Step 4.

relax> calc('600MHz') # Step 6.

relax> monte_carlo.error_analysis('600MHz') # Step 8.
```

6.2.36 monte_carlo.initial_values

Synopsis

Function for setting the initial simulation parameter values.

Defaults

monte_carlo.initial_values(self, run=None)

Keyword Arguments

run: The name of the run.

Description

This function only effects runs where minimisation occurs and can therefore be skipped if the values or parameters of a run are calculated rather than minimised. However, if accidentally run in this case, the results will be unaffected. It should only be called after the model or run is fully minimised. Once called, the functions '`grid_search`' and '`minimise`' will only effect the simulations and not the model parameters.

The initial values of the parameters for each simulation is set to the minimised parameters of the model. A grid search can be undertaken for each simulation instead, although this is computationally expensive and unnecessary. The minimisation function should be executed for a second time after running this function.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search('m1', inc=11) # Step 2.

relax> minimise('newton', run='m1') # Step 2.

relax> monte_carlo.setup('m1', number=500) # Step 3.

relax> monte_carlo.create_data('m1', method='back_calc') # Step 4.

relax> monte_carlo.initial_values('m1') # Step 5.

relax> minimise('newton', run='m1') # Step 6.

relax> eliminate('m1') # Step 7.

relax> monte_carlo.error_analysis('m1') # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc('600MHz') # Step 2.

relax> monte_carlo.setup('600MHz', number=500) # Step 3.

relax> monte_carlo.create_data('600MHz', method='back_calc') # Step 4.

relax> calc('600MHz') # Step 6.

relax> monte_carlo.error_analysis('600MHz') # Step 8.
```

6.2.37 monte_carlo.off

Synopsis

Function for turning simulations off.

Defaults

monte_carlo.off(self, run=None)

Keyword Arguments

run: The name of the run.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search(‘m1’, inc=11) # Step 2.

relax> minimise(‘newton’, run=‘m1’) # Step 2.

relax> monte_carlo.setup(‘m1’, number=500) # Step 3.

relax> monte_carlo.create_data(‘m1’, method=‘back_calc’) # Step 4.

relax> monte_carlo.initial_values(‘m1’) # Step 5.

relax> minimise(‘newton’, run=‘m1’) # Step 6.

relax> eliminate(‘m1’) # Step 7.

relax> monte_carlo.error_analysis(‘m1’) # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc(‘600MHz’) # Step 2.

relax> monte_carlo.setup(‘600MHz’, number=500) # Step 3.

relax> monte_carlo.create_data(‘600MHz’, method=‘back_calc’) # Step 4.

relax> calc(‘600MHz’) # Step 6.

relax> monte_carlo.error_analysis(‘600MHz’) # Step 8.
```

6.2.38 monte_carlo.on

Synopsis

Function for turning simulations on.

Defaults

monte_carlo.on(self, run=None)

Keyword Arguments

run: The name of the run.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search(‘m1’, inc=11) # Step 2.

relax> minimise(‘newton’, run=‘m1’) # Step 2.

relax> monte_carlo.setup(‘m1’, number=500) # Step 3.

relax> monte_carlo.create_data(‘m1’, method=‘back_calc’) # Step 4.

relax> monte_carlo.initial_values(‘m1’) # Step 5.

relax> minimise(‘newton’, run=‘m1’) # Step 6.

relax> eliminate(‘m1’) # Step 7.

relax> monte_carlo.error_analysis(‘m1’) # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc(‘600MHz’) # Step 2.

relax> monte_carlo.setup(‘600MHz’, number=500) # Step 3.

relax> monte_carlo.create_data(‘600MHz’, method=‘back_calc’) # Step 4.

relax> calc(‘600MHz’) # Step 6.

relax> monte_carlo.error_analysis(‘600MHz’) # Step 8.
```

6.2.39 monte_carlo.setup

Synopsis

Function for setting up Monte Carlo simulations.

Defaults

monte_carlo.setup(self, run=None, number=500)

Keyword Arguments

run: The name of the run.

number: The number of Monte Carlo simulations.

Description

This function must be called prior to any of the other Monte Carlo functions. The effect is that the number of simulations for the given run will be set and that simulations will be turned on.

Monte Carlo Simulation Overview

For proper error analysis using Monte Carlo simulations, a sequence of function calls is required for running the various simulation components. The steps necessary for implementing Monte Carlo simulations are:

1. The measured data set together with the corresponding error set should be loaded into relax.

2. Either minimisation is used to optimise the parameters of the chosen model, or a calculation is run.

3. To initialise and turn on Monte Carlo simulations, the number of simulations, n, needs to be set.

4. The simulation data needs to be created either by back calculation from the fully minimised model parameters from step 2 or by direct calculation when values are calculated rather than minimised. The error set is used to randomise each simulation data set by assuming Gaussian errors. This creates a synthetic data set for each Monte Carlo simulation.

5. Prior to minimisation of the parameters of each simulation, initial parameter estimates are required. These are taken as the optimised model parameters. An alternative is to use a grid search for each simulation to generate initial estimates, however this is extremely computationally expensive. For the case where values are calculated rather

than minimised, this step should be skipped (although the results will be unaffected if this is accidentally run).

6. Each simulation requires minimisation or calculation. The same techniques as used in step 2, excluding the grid search when minimising, should be used for the simulations.

7. Failed simulations are removed using the techniques of model elimination.

8. The model parameter errors are calculated from the distribution of simulation parameters.

Monte Carlo simulations can be turned on or off using functions within this class. Once the function for setting up simulations has been called, simulations will be turned on. The effect of having simulations turned on is that the functions used for minimisation (grid search, minimise, etc) or calculation will only affect the simulation parameters and not the model parameters. By subsequently turning simulations off using the appropriate function, the functions used in minimisation will affect the model parameters and not the simulation parameters.

An example, for model-free analysis, which includes only the functions required for implementing the above steps is:

```
relax> grid_search('m1', inc=11) # Step 2.

relax> minimise('newton', run='m1') # Step 2.

relax> monte_carlo.setup('m1', number=500) # Step 3.

relax> monte_carlo.create_data('m1', method='back_calc') # Step 4.

relax> monte_carlo.initial_values('m1') # Step 5.

relax> minimise('newton', run='m1') # Step 6.

relax> eliminate('m1') # Step 7.

relax> monte_carlo.error_analysis('m1') # Step 8.
```

An example for reduced spectral density mapping is:

```
relax> calc('600MHz') # Step 2.

relax> monte_carlo.setup('600MHz', number=500) # Step 3.

relax> monte_carlo.create_data('600MHz', method='back_calc') # Step 4.

relax> calc('600MHz') # Step 6.

relax> monte_carlo.error_analysis('600MHz') # Step 8.
```

6.2.40 noe.error

Synopsis

Function for setting the errors in the reference or saturated NOE spectra.

Defaults

noe.error(self, run=None, error=0.0, spectrum_type=None, res_num=None, res_name=None)

Keyword Arguments

run: The name of the run.

error: The error.

spectrum_type: The type of spectrum.

res_num: The residue number.

res_name: The residue name.

Description

The spectrum_type argument can have the following values:

'`ref`' – The NOE reference spectrum.

'`sat`' – The NOE spectrum with proton saturation turned on.

If the '`res_num`' and '`res_name`' arguments are left as the defaults of None, then the error value for all residues will be set to the supplied value. Otherwise the residue number can be set to either an integer for selecting a single residue or a python regular expression string for selecting multiple residues. The residue name argument must be a string and can use regular expression as well.

6.2.41 noe.read

Synopsis

Function for reading peak intensities from a file for NOE calculations.

Defaults

noe.read(self, run=None, file=None, dir=None, spectrum_type=None, format='sparky', heteronuc='N', proton='HN', int_col=None)

Keyword Arguments

run: The name of the run.

file: The name of the file containing the sequence data.

dir: The directory where the file is located.

spectrum_type: The type of spectrum.

format: The type of file containing peak intensities.

heteronuc: The name of the heteronucleus as specified in the peak intensity file.

proton: The name of the proton as specified in the peak intensity file.

int_col: The column containing the peak intensity data (for a non-standard formatted file).

Description

The peak intensity can either be from peak heights or peak volumes.

The '`spectrum_type`' argument can have the following values:

'`ref`' – The NOE reference spectrum.

'`sat`' – The NOE spectrum with proton saturation turned on.

The '`format`' argument can currently be set to:

'`sparky`'

'`xeasy`'

If the '`format`' argument is set to '`sparky`', the file should be a Sparky peak list saved after typing the command '`lt`'. The default is to assume that columns 0, 1, 2, and 3 (1^{st}, 2^{nd}, 3^{rd}, and 4^{th}) contain the Sparky assignment, w1, w2, and peak intensity data respectively. The frequency data w1 and w2 are ignored while the peak intensity data can either be the peak height or volume displayed by changing the window options. If the peak intensity data is not within column 3, set the argument '`int_col`' to the appropriate value (column numbering starts from 0 rather than 1).

If the '`format`' argument is set to '`xeasy`', the file should be the saved XEasy text window output of the list peak entries command, '`tw`' followed by '`le`'. As the columns are fixed, the peak intensity column is hardwired to number 10 (the 11^{th} column) which contains either the peak height or peak volume data. Because the columns are fixed, the '`int_col`' argument will be ignored.

The '`heteronuc`' and '`proton`' arguments should be set respectively to the name of the heteronucleus and proton in the file. Only those lines which match these labels will be used.

Examples

To read the reference and saturated spectra peak heights from the Sparky formatted files '`ref.list`' and '`sat.list`' to the run '`noe`', type:

```
relax> noe.read('noe', file='ref.list', spectrum_type='ref')

relax> noe.read('noe', file='sat.list', spectrum_type='sat')
```

To read the reference and saturated spectra peak heights from the XEasy formatted files '`ref.text`' and '`sat.text`' to the run '`noe`', type:

```
relax> noe.read('noe', file='ref.text', spectrum_type='ref', format='xeasy')

relax> noe.read('noe', file='sat.text', spectrum_type='sat', format='xeasy')
```

6.2.42 nuclei

Synopsis

Function for setting the gyromagnetic ratio of the heteronucleus.

Defaults

nuclei(self, heteronuc='N')

Keyword arguments

heteronuc: The type of heteronucleus.

Description

The heteronuc argument can be set to the following strings:

N – Nitrogen, -2.7126e7

C – Carbon, 2.2e7

6.2.43 palmer.create

Synopsis

Function for creating the Modelfree4 input files.

Defaults

palmer.create(self, run=None, dir=None, force=0, diff_search='none', sims=0, sim_type='pred', trim=0, steps=20, constraints=1, nucleus='15N', atom1='N', atom2='H')

Keyword Arguments

run: The name of the run.

dir: The directory to place the files. The default is the value of '`run`'.

force: A flag which if set to 1 will cause the results file to be overwritten if it already exists.

diff_search: See the Modelfree4 manual for '`diffusion_search`'.

sims: The number of Monte Carlo simulations.

sim_type: See the Modelfree4 manual.

trim: See the Modelfree4 manual.

steps: See the Modelfree4 manual.

constraints: A flag specifying whether the parameters should be constrained. The default is to turn constraints on (constraints=1).

nucleus: A three letter string describing the nucleus type, ie 15N, 13C, etc.

atom1: The symbol of the X nucleus in the pdb file.

atom2: The symbol of the H nucleus in the pdb file.

Description

The following files are created

'`dir/mfin`',

'`dir/mfdata`',

'`dir/mfpar`',

'`dir/mfmodel`',

'`dir/run.sh`'.

The file '`run/run.sh`' contains the single command,

'`modelfree4 -i mfin -d mfdata -p mfpar -m mfmodel -o mfout -e out`',

which can be used to execute modelfree4.

6.2.44 palmer.execute

Synopsis

Function for executing Modelfree4.

Defaults

palmer.execute(self, run=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

dir: The directory to place the files. The default is the value of '`run`'.

force: A flag which if set to 1 will cause the results file to be overwritten if it already exists.

Description

Modelfree 4 will be executed as

```
$ modelfree4 -i mfin -d mfdata -p mfpar -m mfmodel -o mfout -e out
```

If a PDB file is loaded and non-isotropic diffusion is selected, then the file name will be placed on the command line as '`-s pdb_file_name`'.

6.2.45 palmer.extract

Synopsis

Function for extracting data from the Modelfree4 '`mfout`' star formatted file.

Defaults

palmer.extract(self, run=None, dir=None)

Keyword Arguments

run: The name of the run.

dir: The directory where the file '`mfout`' is found. The default is the value of '`run`'.

6.2.46 pdb

Synopsis

The pdb loading function.

Defaults

pdb(self, run=None, file=None, dir=None, model=None, heteronuc='N', proton='H', load_seq=1)

Keyword Arguments

run: The run to assign the structure to.

file: The name of the PDB file.

dir: The directory where the file is located.

model: The PDB model number.

heteronuc: The name of the heteronucleus as specified in the PDB file.

proton: The name of the proton as specified in the PDB file.

load_seq: A flag specifying whether the sequence should be loaded from the PDB file.

Description

To load a specific model from the PDB file, set the model flag to an integer i. The structure beginning with the line '`MODEL` i' in the PDB file will be loaded. Otherwise all structures will be loaded starting from the model number 1.

To load the sequence from the PDB file, set the '`load_seq`' flag to 1. If the sequence has previously been loaded, then this flag will be ignored.

Once the PDB structures are loaded, unit XH bond vectors will be calculated. The vectors are calculated using the atomic coordinates of the atoms specified by the arguments heteronuc and proton. If more than one model structure is loaded, the unit XH vectors for each model will be calculated and the final unit XH vector will be taken as the average.

Example

To load all structures from the PDB file '`test.pdb`' in the directory '`~/pdb`' for use in the model-free analysis run '`m8`' where the heteronucleus in the PDB file is '`N`' and the proton is '`H`', type:

```
relax> pdb('m8', 'test.pdb', '~/pdb', 1, 'N', 'H')
```

```
relax> pdb(run='m8', file='test.pdb', dir='pdb', model=1, heteronuc='N', proton='H')
```

To load the 10^{th} model from the file '`test.pdb`', use:

```
relax> pdb('m1', 'test.pdb', model=10)
```

```
relax> pdb(run='m1', file='test.pdb', model=10)
```

6.2.47 relax_data.back_calc

Synopsis

Function for back calculating relaxation data.

Defaults

relax_data.back_calc(self, run=None, ri_label=None, frq_label=None, frq=None)

Keyword Arguments

run: The name of the run.

ri_label: The relaxation data type, ie '`R1`', '`R2`', or '`NOE`'.

frq_label: The field strength label.

frq: The spectrometer frequency in Hz.

6.2.48 relax_data.copy

Synopsis

Function for copying relaxation data from run1 to run2.

Defaults

relax_data.copy(self, run1=None, run2=None, ri_label=None, frq_label=None)

Keyword Arguments

run1: The name of the run to copy the sequence from.

run2: The name of the run to copy the sequence to.

ri_label: The relaxation data type, ie 'R1', 'R2', or 'NOE'.

frq_label: The field strength label.

Description

This function will copy relaxation data from 'run1' to 'run2'. If ri_label and frq_label are not given then all relaxation data will be copied, otherwise only a specific data set will be copied.

Examples

To copy all relaxation data from run 'm1' to run 'm9', type one of:

```
relax> relax_data.copy('m1', 'm9')

relax> relax_data.copy(run1='m1', run2='m9')

relax> relax_data.copy('m1', 'm9', None, None)

relax> relax_data.copy(run1='m1', run2='m9', ri_label=None, frq_label=None)
```

To copy only the NOE relaxation data with the frq_label of '800' from 'm3' to 'm6', type one of:

```
relax> relax_data.copy('m3', 'm6', 'NOE', '800')

relax> relax_data.copy(run1='m3', run2='m6', ri_label='NOE', frq_label='800')
```

6.2.49 relax_data.delete

Synopsis

Function for deleting the relaxation data corresponding to ri_label and frq_label.

Defaults

relax_data.delete(self, run=None, ri_label=None, frq_label=None)

Keyword Arguments

run: The name of the run.

ri_label: The relaxation data type, ie 'R1', 'R2', or 'NOE'.

frq_label: The field strength label.

Examples

To delete the relaxation data corresponding to ri_label='NOE', frq_label='600', and the run 'm4', type:

```
relax> relax_data.delete('m4', 'NOE', '600')
```

6.2.50 relax_data.display

Synopsis

Function for displaying the relaxation data corresponding to ri_label and frq_label.

Defaults

relax_data.display(self, run=None, ri_label=None, frq_label=None)

Keyword Arguments

run: The name of the run.

ri_label: The relaxation data type, ie 'R1', 'R2', or 'NOE'.

frq_label: The field strength label.

Examples

To display the NOE relaxation data at 600 MHz from the run 'm4', type

```
relax> relax_data.display('m4', 'NOE', '600')
```

6.2.51 relax_data.read

Synopsis

Function for reading R_1, R_1, or NOE relaxation data from a file.

Defaults

relax_data.read(self, run=None, ri_label=None, frq_label=None, frq=None, file=None, dir=None, num_col=0, name_col=1, data_col=2, error_col=3, sep=None)

Keyword Arguments

run: The name of the run.

ri_label: The relaxation data type, ie '`R1`', '`R2`', or '`NOE`'.

frq_label: The field strength label.

frq: The spectrometer frequency in Hz.

file: The name of the file containing the relaxation data.

dir: The directory where the file is located.

num_col: The residue number column (the default is 0, ie the first column).

name_col: The residue name column (the default is 1).

data_col: The relaxation data column (the default is 2).

error_col: The experimental error column (the default is 3).

sep: The column separator (the default is white space).

Description

The frequency label argument can be anything as long as data collected at the same field strength have the same label.

Examples

The following commands will read the NOE relaxation data collected at 600 MHz out of a file called '`noe.600.out`' where the residue numbers, residue names, data, errors are in the first, second, third, and forth columns respectively.

```
relax> relax_data.read('m1', 'NOE', '600', 599.7 * 1e6, 'noe.600.out')
```

```
relax> relax_data.read('m1', ri_label='NOE', frq_label='600', frq=600.0 * 1e6,
file='noe.600.out')
```

The following commands will read the R_1 data out of the file '`r2.out`' where the residue numbers, residue names, data, errors are in the second, third, fifth, and sixth columns respectively. The columns are separated by commas.

```
relax> relax_data.read('m1', 'R2', '800 MHz', 8.0 * 1e8, 'r2.out', 1, 2, 4, 5, ',')
```

```
relax> relax_data.read('m1', ri_label='R2', frq_label='800 MHz', frq=8.0*1e8,
file='r2.out', num_col=1, name_col=2, data_col=4, error_col=5, sep=',')
```

The following commands will read the R_1 data out of the file '`r1.out`' where the columns are separated by the symbol '`%`'

```
relax> relax_data.read('m1', 'R1', '300', 300.1 * 1e6, 'r1.out', sep='%')
```

6.2.52 relax_data.write

Synopsis

Function for writing R_1, R_1, or NOE relaxation data to a file.

Defaults

relax_data.write(self, run=None, ri_label=None, frq_label=None, file=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

ri_label: The relaxation data type, ie '`R1`', '`R2`', or '`NOE`'.

frq_label: The field strength label.

file: The name of the file.

dir: The directory name.

force: A flag which, if set to 1, will cause the file to be overwritten.

Description

If no directory name is given, the file will be placed in the current working directory. The '`ri_label`' and '`frq_label`' arguments are required for selecting which relaxation data to write to file.

6.2.53 relax_fit.mean_and_error

Synopsis

Function for calculating the average intensity and standard deviation of all spectra.

Defaults

relax_fit.mean_and_error(self, run=None)

Keyword Arguments

run: The name of the run.

Errors of individual residues at a single time point

The standard deviation for a single residue at a single time point is calculated by the formula

```
                 ----------------------------
sd =    \/ sum({Ii - Iav}^2) / (n - 1) ,
```

where n is the total number of collected spectra for the time point and i is the corresponding index, Ii is the peak intensity for spectrum i, Iav is the mean over all spectra, ie the sum of all peak intensities divided by n.

Averaging of the errors

As the value of n in the above equation is always very low, normally only a couple of spectra are collected per time point, the standard deviation of all residues is averaged for a single time point. Although this results in all residues having the same error, the accuracy of the error estimate is significantly improved.

Errors across multiple time points

If all spectra are collected in duplicate (triplicate or higher number of spectra are supported), the each time point will have its own error estimate. However, if there are time points in the series which only consist of a single spectrum, then the standard deviations of replicated time points will be averaged. Hence, for the entire experiment there will be a single error value for all residues and for all time points.

A better approach rather than averaging across all time points would be to use a form of interpolation as the errors across time points generally decreases for longer time periods. This is currently not implemented.

6.2.54 relax_fit.read

Synopsis

Function for reading peak intensities from a file.

Defaults

relax_fit.read(self, run=None, file=None, dir=None, relax_time=0.0, format='sparky', heteronuc='N', proton='HN', int_col=None)

Keyword Arguments

run: The name of the run.

file: The name of the file containing the sequence data.

dir: The directory where the file is located.

relax_time: The time, in seconds, of the relaxation period.

format: The type of file containing peak intensities.

heteronuc: The name of the heteronucleus as specified in the peak intensity file.

proton: The name of the proton as specified in the peak intensity file.

int_col: The column containing the peak intensity data (for a non-standard formatted file).

Description

The peak intensity can either be from peak heights or peak volumes.

The format argument can currently be set to:

'`sparky`'

'`xeasy`'

If the format argument is set to '`sparky`', the file should be a Sparky peak list saved after typing the command '`lt`'. The default is to assume that columns 0, 1, 2, and 3 (1^{st}, 2^{nd}, 3^{rd}, and 4^{th}) contain the Sparky assignment, w1, w2, and peak intensity data respectively. The frequency data w1 and w2 are ignored while the peak intensity data can either be the peak height or volume displayed by changing the window options. If the peak intensity

data is not within column 3, set the argument int_col to the appropriate value (column numbering starts from 0 rather than 1).

If the format argument is set to '`xeasy`', the file should be the saved XEasy text window output of the list peak entries command, '`tw`' followed by '`le`'. As the columns are fixed, the peak intensity column is hardwired to number 10 (the 11^{th} column) which contains either the peak height or peak volume data. Because the columns are fixed, the int_col argument will be ignored.

The heteronuc and proton arguments should be set respectively to the name of the heteronucleus and proton in the file. Only those lines which match these labels will be used.

6.2.55 relax_fit.select_model

Synopsis

Function for the selection of the relaxation curve type.

Defaults

relax_fit.select_model(self, run=None, model='exp')

Keyword Arguments

run: The name of the run.

model: The type of relaxation curve to fit.

The preset models

The supported relaxation experiments include the default two parameter exponential fit, selected by setting the '`fit_type`' argument to '`exp`', and the three parameter inversion recovery experiment in which the peak intensity limit is a non-zero value, selected by setting the argument to '`inv`'.

The parameters of these two models are

'`exp`' – [Rx, I0],

'`inv`' – [Rx, I0, Iinf].

6.2.56 results.display

Synopsis

Function for displaying the results of the run.

Defaults

results.display(self, run=None, format='columnar')

Keyword Arguments

run: The name of the run.

format: The format of the output.

6.2.57 results.read

Synopsis

Function for reading results from a file.

Defaults

results.read(self, run=None, file='results', dir='run', format='columnar')

Keyword Arguments

run: The name of the run.

file: The name of the file to read results from.

dir: The directory where the file is located.

Description

If no directory name is given, the results file will be searched for in a directory named after the run name. To search for the results file in the current working directory, set dir to None.

This function is able to handle uncompressed, bzip2 compressed files, or gzip compressed files automatically. The full file name including extension can be supplied, however, if the file cannot be found, this function will search for the file name with '`.bz2`' appended followed by the file name with '`.gz`' appended.

6.2.58 results.write

Synopsis

Function for writing results of the run to a file.

Defaults

results.write(self, run=None, file='results', dir='run', force=0, format='columnar', compress_type=1)

Keyword Arguments

run: The name of the run.

file: The name of the file to output results to. The default is '`results`'.

dir: The directory name.

force: A flag which, if set to 1, will cause the results file to be overwritten.

format: The format of the output.

compress_type: The type of compression to use when creating the file.

Description

If no directory name is given, the results file will be placed in a directory named after the run name. To place the results file in the current working directory, set dir to None.

The default behaviour of this function is to compress the file using bzip2 compression. If the extension '`.bz2`' is not included in the file name, it will be added. The compression can, however, be changed to either no compression or gzip compression. This is controlled by the compress_type argument which can be set to

0 – No compression (no file extension),

1 – bzip2 compression ('`.bz2`' file extension),

2 – gzip compression ('`.gz`' file extension).

The complementary read function will automatically handle the compressed files.

6.2.59 run.create

Synopsis

Function for setting up a run type.

Defaults

run.create(self, run=None, run_type=None)

Keyword Arguments

run: The name of the run.

type: The type of run.

Description

The run name can be any string however the run type can only be one of the following

'jw' – Reduced spectral density mapping,

'mf' – Model-free analysis,

'noe' – Steady state NOE calculation,

'relax_fit' – Relaxation curve fitting,

'srls' – SRLS analysis.

Examples

To set up a model-free analysis run with the name 'm5', type:

```
relax> run.create('m5', 'mf')
```

6.2.60 run.delete

Synopsis

Function for deleting a run.

Defaults

run.delete(self, run=None)

Keyword Arguments

run: The name of the run.

Description

This function will destroy all data corresponding to the given run.

6.2.61 select.all

Synopsis

Function for selecting all residues.

Defaults

select.all(self, run=None)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

Examples

To select all residues for all runs type:

```
relax> select.all()
```

To select all residues for the run '`srls_m1`', type:

```
relax> select.all('srls_m1')
```

```
relax> select.all(run='srls_m1')
```

6.2.62 select.read

Synopsis

Function for selecting the residues contained in a file.

Defaults

select.read(self, run=None, file=None, dir=None, change_all=0)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

file: The name of the file containing the list of residues to select.

dir: The directory where the file is located.

change_all: A flag specifying if all other residues should be changed.

Description

The file must contain one residue number per line. The number is taken as the first column of the file and all other columns are ignored. Empty lines and lines beginning with a hash are ignored.

The '`change_all`' flag argument default is zero meaning that all residues currently either selected or unselected will remain that way. Setting the argument to 1 will cause all residues not specified in the file to be unselected.

Examples

To select all residues in the file '`isolated_peaks`', type:

```
relax> select.read('noe', 'isolated_peaks')

relax> select.read(run='noe', file='isolated_peaks')
```

6.2.63 select.res

Synopsis

Function for selecting specific residues.

Defaults

select.res(self, run=None, num=None, name=None, change_all=0)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

num: The residue number.

name: The residue name.

change_all: A flag specifying if all other residues should be changed.

Description

The residue number can be either an integer for selecting a single residue or a python regular expression, in string form, for selecting multiple residues. For details about using regular expression, see the python documentation for the module '`re`'.

The residue name argument must be a string. Regular expression is also allowed.

The '`change_all`' flag argument default is zero meaning that all residues currently either selected or unselected will remain that way. Setting the argument to 1 will cause all residues not specified by '`num`' or '`name`' to become unselected.

Examples

To select only glycines and alanines for the run '`m3`', assuming they have been loaded with the names GLY and ALA, type:

```
relax> select.res(run='m3', name='GLY|ALA', change_all=1)
```

```
relax> select.res(run='m3', name='[GA]L[YA]', change_all=1)
```

To select residue 5 CYS in addition to the currently selected residues, type:

```
relax> select.res('m3', 5)
```

```
relax> select.res('m3', 5, 'CYS')
```

```
relax> select.res('m3', '5')
```

```
relax> select.res('m3', '5', 'CYS')
```

```
relax> select.res(run='m3', num='5', name='CYS')
```

6.2.64 select.reverse

Synopsis

Function for the reversal of the residue selection.

Defaults

select.reverse(self, run=None)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

Examples

To unselect all currently selected residues and select those which are unselected type:

```
relax> select.reverse()
```

6.2.65 sequence.add

Synopsis

Function for adding a residue onto the sequence.

Defaults

sequence.add(self, run=None, res_num=None, res_name=None, select=1)

Keyword Arguments

run: The name of the run.

res_num: The residue number.

res_name: The name of the residue.

select: A flag specifying if the residue should be selected.

Description

Using this function a new sequence can be generated without having to load the sequence from a file. However if the sequence already exists, the new residue will be added to the end. The same residue number cannot be used more than once.

Examples

The following sequence of commands will generate the sequence 1 ALA, 2 GLY, 3 LYS and assign it to the run 'm3':

```
relax> run = 'm3'

relax> sequence.add(run, 1, 'ALA')

relax> sequence.add(run, 2, 'GLY')

relax> sequence.add(run, 3, 'LYS')
```

6.2.66 sequence.copy

Synopsis

Function for copying the sequence from run1 to run2.

Defaults

sequence.copy(self, run1=None, run2=None)

Keyword Arguments

run1: The name of the run to copy the sequence from.

run2: The name of the run to copy the sequence to.

Description

This function will copy the sequence from '`run1`' to '`run2`'. '`run1`' must contain sequence information, while '`run2`' must have no sequence loaded.

Examples

To copy the sequence from the run '`m1`' to the run '`m2`', type:

```
relax> sequence.copy('m1', 'm2')
```

```
relax> sequence.copy(run1='m1', run2='m2')
```

6.2.67 sequence.delete

Synopsis

Function for deleting the sequence.

Defaults

sequence.delete(self, run=None)

Keyword Arguments

run: The name of the run.

Description

This function has the same effect as using the '`delete`' function to delete all residue specific data.

6.2.68 sequence.display

Synopsis

Function for displaying the sequence.

Defaults

sequence.display(self, run=None)

Keyword Arguments

run: The name of the run.

6.2.69 sequence.read

Synopsis

Function for reading sequence data.

Defaults

sequence.read(self, run=None, file=None, dir=None, num_col=0, name_col=1, sep=None)

Keyword Arguments

run: The name of the run.

file: The name of the file containing the sequence data.

dir: The directory where the file is located.

num_col: The residue number column (the default is 0, ie the first column).

name_col: The residue name column (the default is 1).

sep: The column separator (the default is white space).

Description

If no directory is given, the file will be assumed to be in the current working directory.

Examples

The following commands will read the sequence data out of a file called '`seq`' where the residue numbers and names are in the first and second columns respectively and assign it to the run '`m1`'.

```
relax> sequence.read('m1', 'seq')

relax> sequence.read('m1', 'seq', num_col=0, name_col=1)

relax> sequence.read(run='m1', file='seq', num_col=0, name_col=1, sep=None)
```

The following commands will read the sequence out of the file '`noe.out`' which also contains the NOE values.

```
relax> sequence.read('m1', 'noe.out')

relax> sequence.read('m1', 'noe.out', num_col=0, name_col=1)

relax> sequence.read(run='m1', file='noe.out', num_col=0, name_col=1)
```

The following commands will read the sequence out of the file '`noe.600.out`' where the residue numbers are in the second column, the names are in the sixth column and the columns are separated by commas and assign it to the run '`m5`'.

```
relax> sequence.read(‘m5’, ‘noe.600.out’, num_col=1, name_col=5, sep=‘,’)

relax> sequence.read(run=‘m5’, file=‘noe.600.out’, num_col=1, name_col=5, sep=‘,’)
```

6.2.70 sequence.sort

Synopsis

Function for numerically sorting the sequence by residue number.

Defaults

sequence.sort(self, run=None)

Keyword Arguments

run: The name of the run.

6.2.71 sequence.write

Synopsis

Function for writing the sequence to a file.

Defaults

sequence.write(self, run=None, file=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

file: The name of the file.

dir: The directory name.

force: A flag which, if set to 1, will cause the file to be overwritten.

Description

If no directory name is given, the file will be placed in the current working directory.

6.2.72 state.load

Synopsis

Function for loading a saved program state.

Defaults

state.load(self, file=None, dir=None)

Keyword Arguments

file: The file name, which must be a string, of a saved program state.

dir: Directory which the file is found in.

Description

This function is able to handle uncompressed, bzip2 compressed files, or gzip compressed files automatically. The full file name including extension can be supplied, however, if the file cannot be found, this function will search for the file name with '`.bz2`' appended followed by the file name with '`.gz`' appended.

Examples

The following commands will load the state saved in the file '`save`'.

```
relax> state.load('save')
relax> state.load(file='save')
```

The following commands will load the state saved in the bzip2 compressed file '`save.bz2`'.

```
relax> state.load('save')
relax> state.load(file='save')
relax> state.load('save.bz2')
relax> state.load(file='save.bz2')
```

6.2.73 state.save

Synopsis

Function for saving the program state.

Defaults

state.save(self, file=None, dir=None, force=0, compress_type=1)

Keyword Arguments

file: The file name, which must be a string, to save the current program state in.

dir: The directory to place the file in.

force: A flag which if set to 1 will cause the file to be overwritten.

Description

The default behaviour of this function is to compress the file using bzip2 compression. If the extension '`.bz2`' is not included in the file name, it will be added. The compression can, however, be changed to either no compression or gzip compression. This is controlled by the compress_type argument which can be set to

0 − No compression (no file extension).

1 − bzip2 compression ('`.bz2`' file extension).

2 − gzip compression ('`.gz`' file extension).

Examples

The following commands will save the current program state into the file '`save`':

```
relax> state.save('save', compress_type=0)

relax> state.save(file='save', compress_type=0)
```

The following commands will save the current program state into the bzip2 compressed file '`save.bz2`':

```
relax> state.save('save')

relax> state.save(file='save')

relax> state.save('save.bz2')

relax> state.save(file='save.bz2')
```

If the file ‘save’ already exists, the following commands will save the current program state by overwriting the file.

```
relax> state.save(‘save’, 1)

relax> state.save(file=‘save’, force=1)
```

6.2.74 system

Synopsis

Function which executes the user supplied shell command.

Defaults

system(command)

6.2.75 thread.read

Synopsis

Function for reading a file containing entries for each computer to run calculations on.

Defaults

thread.read(self, file='hosts', dir='~/.relax')

Keyword Arguments

file: The name of the file containing the host entries.

dir: The directory where the hosts file is located.

Description

Certain functions within relax are coded to handle threading. This is achieved by running multiple instances of relax on different processes or computers for each thread. The default behaviour is that the parent instance of relax will execute all the code, however if a hosts file is read or a hosts entry manually entered, then the threaded code will run on the specified hosts. This function is for reading a hosts file which should contain an an entry for each computer on which to run calculations.

For remote computers, a SSH connection will be attempted. Public key authentication must be enabled to run calculations on remote machines so that thread can be created without asking for a password. Details on how to do this are given below.

The format of the hosts file is as follows. Default values are specified by placing the character '`-`' in the corresponding column. Columns can be separated by any whitespace character, and all columns must contain an entry. Any lines beginning with a hash will be ignored.

Column 1: The host name or IP address of the computer on which to run a thread.

Column 2: The login name of the user on the remote machine. The default is to use the same name as the current user.

Column 3: The full program path. The default is to run '`relax`'. This only works if relax can be found in the environmental variable $PATH, as alias are not recognised.

Column 4: The working directory where thread specific files are stored. The default is '`~/.relax`' where the tilde '`~`' symbol represents the user's home directory on the remote machine.

Column 5: The priority value for running the program. The default is 15. The remote instances of relax will be niced to this value.

Column 6: The number of CPU or CPU cores on the machine. The default is 1. A thread is started for each CPU.

An example is:

```
# Host          User name       Program path            Working directory    Priority  CPUs
localhost       -               -                       -                    0         2
192.168.0.10    dauvergne       /usr/local/bin/relax    -                    -         -
192.168.0.11    edward          -                       -                    -         -
```

In this case, two threads will be run on the parent computer which would be either a dual CPU system or a dual core '`Hyper threaded`' Pentium processor. These threads will have the highest level user priority of 0. The other two machines will have single threads running with a low priority of 15.

Once threading is enabled, to allow calculations to run on the parent machine a '`localhost`' entry should be included.

If the keyword argument '`dir`' is set to None, the hosts file will be assumed to be in the current working directory.

SSH Public Key Authentication

To enable SSH Public Key Authentication for the use of ssh, sftp, and scp without having to type a password, use the following steps. This is essential for running a thread on a remote machine.

If the files '`id_rsa`' and '`id_rsa.pub`' do not exist in the directory '`~/.ssh`', type:

```
$ ssh-keygen -t rsa
```

Press enter three times when asked for input. This will generate the two identification files. Then, to copy the public key into the '`authorized_keys`' file on the remote machine, type:

```
$ ssh zucchini "echo $(cat ~/.ssh/id_rsa.pub) >> ~/.ssh/authorized_keys"
```

Make sure you replace '`zucchini`' with the name or IP address of the remote machine. To use DSA rather than RSA authentication, replace '`rsa`' with '`dsa`' in the above commands. Normally the sshd keyword StrictModes, which is found in the file '`/etc/ssh/sshd_config`', is set to '`yes`' or, if unspecified, defaults to '`yes`'. In this case, public key authentication may fail as the permissions of the remote file '`~/.ssh/authorized_keys`' may be too permissive. The file should only be read/write for the user, ie 600. To remotely change the permissions, type:

```
$ ssh zucchini "chmod 600 ~/.ssh/authorized_keys"
```

One last keyword may need to be changed in the file '`/etc/ssh/sshd_config`'. If the keyword PubkeyAuthentication is set to '`no`', change this to '`yes`'. The default is yes, so if the keyword is missing or is commented out, nothing needs to be done.

Public key authentication should now work. To test, type:

```
$ ssh zucchini
```

This should securely login into the remote machine without asking for a password. If a password prompt appears, check all the permissions on the directory '`~/.ssh`' and all files within or set the sshd_config keyword StrictModes to '`no`'.

```
$ ssh zucchini "chmod 700 ~/.ssh/"
```

```
$ ssh zucchini "chmod 600 ~/.ssh/*"
```

```
$ ssh zucchini "chmod 644 ~/.ssh/*.pub"
```

Finally, if all else fails, make sure the three lines

```
RSAAuthentication yes
PubkeyAuthentication yes
AuthorizedKeysFile      .ssh/authorized_keys
```

of the file '`sshd_config`' found within the directory '`/etc/ssh/`' are uncommented and not set to '`no`' or the '`AuthorizedKeysFile`' set to another file name.

6.2.76 unselect.all

Synopsis

Function for unselecting all residues.

Defaults

unselect.all(self, run=None)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

Examples

To unselect all residues type:

```
relax> unselect.all()
```

To unselect all residues for the run '`srls_m1`', type:

```
relax> select.all('srls_m1')
```

```
relax> select.all(run='srls_m1')
```

6.2.77 unselect.read

Synopsis

Function for unselecting the residues contained in a file.

Defaults

unselect.read(self, run=None, file=None, dir=None, change_all=0)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

file: The name of the file containing the list of residues to unselect.

dir: The directory where the file is located.

change_all: A flag specifying if all other residues should be changed.

Description

The file must contain one residue number per line. The number is taken as the first column of the file and all other columns are ignored. Empty lines and lines beginning with a hash are ignored.

The '`change_all`' flag argument default is zero meaning that all residues currently either selected or unselected will remain that way. Setting the argument to 1 will cause all residues not specified in the file to be selected.

Examples

To unselect all overlapped residues in the file '`unresolved`', type:

```
relax> unselect.read('noe', 'unresolved')

relax> unselect.read(run='noe', file='unresolved')
```

6.2.78 unselect.res

Synopsis

Function for unselecting specific residues.

Defaults

unselect.res(self, run=None, num=None, name=None, change_all=0)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

num: The residue number.

name: The residue name.

change_all: A flag specifying if all other residues should be changed.

Description

The residue number can be either an integer for unselecting a single residue or a python regular expression, in string form, for unselecting multiple residues. For details about using regular expression, see the python documentation for the module '`re`'.

The residue name argument must be a string. Regular expression is also allowed.

The '`change_all`' flag argument default is zero meaning that all residues currently either selected or unselected will remain that way. Setting the argument to 1 will cause all residues not specified by '`num`' or '`name`' to become selected.

Examples

To unselect all glycines for the run '`m5`', type:

```
relax> unselect.res(run='m5', name='GLY|ALA')
```

```
relax> unselect.res(run='m5', name='[GA]L[YA]')
```

To unselect residue 12 MET type:

```
relax> unselect.res('m5', 12)
```

```
relax> unselect.res('m5', 12, 'MET')
```

```
relax> unselect.res('m5', '12')
```

```
relax> unselect.res('m5', '12', 'MET')
```

```
relax> unselect.res(run='m5', num='12', name='MET')
```

6.2.79 unselect.reverse

Synopsis

Function for the reversal of the residue selection.

Defaults

unselect.reverse(self, run=None)

Keyword Arguments

run: The name of the run(s). By supplying a single string, array of strings, or None, a single run, multiple runs, or all runs will be selected respectively.

Examples

To unselect all currently selected residues and select those which are unselected type:

```
relax> unselect.reverse()
```

6.2.80 value.copy

Synopsis

Function for copying residue specific data values from run1 to run2.

Defaults

value.copy(self, run1=None, run2=None, param=None)

Keyword Arguments

run1: The name of the run to copy from.

run2: The name of the run to copy to.

param: The parameter to copy.

Description

Only one parameter may be selected, therefore the '`param`' argument should be a string.

If this function is used to change values of previously minimised runs, then the minimisation statistics (chi-squared value, iteration count, function count, gradient count, and Hessian count) will be reset to None.

Examples

To copy the CSA values from the run '`m1`' to '`m2`', type:

```
relax> value.copy('m1', 'm2', 'CSA')
```

Regular expression

The python function '`match`', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'`[]`' – A sequence or set of characters to match to a single character. For example, '`[Ss]2`' will match both '`S2`' and '`s2`'.

'`^`' – Match the start of the string.

'$' – Match the end of the string. For example, '`^[Ss]2$`' will match '`s2`' but not '`S2f`' or '`s2s`'.

'.' – Match any character.

'x*' – Match the character '`x`' any number of times, for example '`x`' will match, as will '`xxxxx`'

'.*' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Model-free set details

Setting a parameter value may have no effect depending on which model-free model is chosen, for example if S_f^2 values and S_s^2 values are set but the run corresponds to model-free model '`m4`' then, because these data values are not parameters of the model, they will have no effect.

Note that the R_{ex} values are scaled quadratically with field strength and should be supplied as a field strength independent value. Use the following formula to get the correct value:

value = R_{ex} / (2.0 * π * frequency) ** 2

where:

R_{ex} is the chemical exchange value for the current frequency.

π is in the namespace of relax, ie just type '`pi`'.

frequency is the proton frequency corresponding to the data.

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping set details

In reduced spectral density mapping, only two values can be set, the bond length and CSA value. These must be set prior to the calculation of spectral density values.

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Relaxation curve fitting set details

Only three parameters can be set, the relaxation rate (Rx), the initial intensity (I0), and the intensity at infinity (Iinf). Setting the parameter Iinf has no effect if the chosen model

is that of the exponential curve which decays to zero.

Relaxation curve fitting data type string matching patterns

Data type	Object name	Patterns
Relaxation rate	'`rx`'	'`^[Rr]x$`'
Initial intensity	'`i0`'	'`^[Ii]0$`'
Intensity at infinity	'`iinf`'	'`^[Ii]inf$`'

6.2.81 value.display

Synopsis

Function for displaying residue specific data values.

Defaults

value.display(self, run=None, param=None)

Keyword Arguments

run: The name of the run.

param: The parameter to display.

Description

Only one parameter may be selected, therefore the '`param`' argument should be a string.

Examples

To show all CSA values for the run '`m1`', type:

```
relax> value.display('m1', 'CSA')
```

Regular expression

The python function '`match`', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'`[]`' – A sequence or set of characters to match to a single character. For example, '`[Ss]2`' will match both '`S2`' and '`s2`'.

'`^`' – Match the start of the string.

'`$`' – Match the end of the string. For example, '`^[Ss]2$`' will match '`s2`' but not '`S2f`' or '`s2s`'.

'`.`' – Match any character.

'x*' – Match the character 'x' any number of times, for example 'x' will match, as will 'xxxxx'

'.*' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Relaxation curve fitting data type string matching patterns

Data type	Object name	Patterns
Relaxation rate	'rx'	'^[Rr]x$'
Initial intensity	'i0'	'^[Ii]0$'
Intensity at infinity	'iinf'	'^[Ii]inf$'

6.2.82 value.read

Synopsis

Function for reading residue specific data values from a file.

Defaults

value.read(self, run=None, param=None, scaling=1.0, file=None, num_col=0, name_col=1, data_col=2, error_col=3, sep=None)

Keyword Arguments

run: The name of the run.

param: The parameter.

scaling: The factor to scale parameters by.

file: The name of the file containing the relaxation data.

num_col: The residue number column (the default is 0, ie the first column).

name_col: The residue name column (the default is 1).

data_col: The relaxation data column (the default is 2).

error_col: The experimental error column (the default is 3).

sep: The column separator (the default is white space).

Description

Only one parameter may be selected, therefore the '`param`' argument should be a string. If the file only contains values and no errors, set the error column argument to None.

If this function is used to change values of previously minimised runs, then the minimisation statistics (chi-squared value, iteration count, function count, gradient count, and Hessian count) will be reset to None.

Examples

To load CSA values for the run '`m1`' from the file '`csa_values`' in the directory '`data`', type any of the following:

```
relax> value.read('m1', 'CSA', 'data/csa_value')

relax> value.read('m1', 'CSA', 'data/csa_value', 0, 1, 2, 3, None, 1)

relax> value.read(run='m1', param='CSA', file='data/csa_value', num_col=0, name_col=1,
data_col=2, error_col=3, sep=None)
```

Regular expression

The python function '`match`', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'`[]`' – A sequence or set of characters to match to a single character. For example, '`[Ss]2`' will match both '`S2`' and '`s2`'.

'`^`' – Match the start of the string.

'`$`' – Match the end of the string. For example, '`^[Ss]2$`' will match '`s2`' but not '`S2f`' or '`s2s`'.

'`.`' – Match any character.

'`x*`' – Match the character '`x`' any number of times, for example '`x`' will match, as will '`xxxxx`'

'`.*`' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Model-free set details

Setting a parameter value may have no effect depending on which model-free model is chosen, for example if S_f^2 values and S_s^2 values are set but the run corresponds to model-free model '`m4`' then, because these data values are not parameters of the model, they will have no effect.

Note that the R_{ex} values are scaled quadratically with field strength and should be supplied as a field strength independent value. Use the following formula to get the correct value:

value = R_{ex} / (2.0 * π * frequency) ** 2

where:

R_{ex} is the chemical exchange value for the current frequency.

π is in the namespace of relax, ie just type '`pi`'.

frequency is the proton frequency corresponding to the data.

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S^2_f	's2f'	'^[Ss]2f$'
Order parameter S^2_s	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping set details

In reduced spectral density mapping, only two values can be set, the bond length and CSA value. These must be set prior to the calculation of spectral density values.

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Relaxation curve fitting set details

Only three parameters can be set, the relaxation rate (Rx), the initial intensity (I0), and the intensity at infinity (Iinf). Setting the parameter Iinf has no effect if the chosen model

is that of the exponential curve which decays to zero.

Relaxation curve fitting data type string matching patterns

Data type	Object name	Patterns
Relaxation rate	'`rx`'	'`^[Rr]x$`'
Initial intensity	'`i0`'	'`^[Ii]0$`'
Intensity at infinity	'`iinf`'	'`^[Ii]inf$`'

6.2.83 value.set

Synopsis

Function for setting residue specific data values.

Defaults

value.set(self, run=None, value=None, param=None, res_num=None, res_name=None)

Keyword arguments

run: The run to assign the values to.

value: The value(s).

param: The parameter(s).

res_num: The residue number.

res_name: The residue name.

Description

If this function is used to change values of previously minimised runs, then the minimisation statistics (chi-squared value, iteration count, function count, gradient count, and Hessian count) will be reset to None.

The value argument can be None, a single value, or an array of values while the parameter argument can be None, a string, or array of strings. The choice of which combination determines the behaviour of this function. The following table describes what occurs in each instance. The Value column refers to the '`value`' argument while the Param column refers to the '`param`' argument. In these columns, '`None`' corresponds to None, '`1`' corresponds to either a single value or single string, and '`n`' corresponds to either an array of values or an array of strings.

Value	Param	Description
None	None	This case is used to set the model parameters prior to minimisation or calculation. The model parameters are set to the default values.
1	None	Invalid combination.
n	None	This case is used to set the model parameters prior to minimisation or calculation. The length of the value array must be equal to the number of model parameters for an individual residue. The parameters will be set to the corresponding number.
None	1	The parameter matching the string will be set to the default value.
1	1	The parameter matching the string will be set to the supplied number.
n	1	Invalid combination.
None	n	Each parameter matching the strings will be set to the default values.
1	n	Each parameter matching the strings will be set to the supplied number.
n	n	Each parameter matching the strings will be set to the corresponding number. Both arrays must be of equal length.

Residue number and name argument

If the '`res_num`' and '`res_name`' arguments are left as the defaults of None, then the function will be applied to all residues. Otherwise the residue number can be set to either an integer for selecting a single residue or a python regular expression string for selecting multiple residues. The residue name argument must be a string and can use regular expression as well. If the data is global non-residue specific data, such as diffusion tensor parameters, supplying the residue number and name will terminate the program with an error.

Examples

To set the parameter values for the run '`test`' to the default values, for all residues, type:

```
relax> value.set('test')
```

To set the parameter values of residue 10, which is the model-free run '`m4`' and has the parameters $\{S^2, \tau_e, R_{ex}\}$, the following can be used. R_{ex} term is the value for the first given field strength.

```
relax> value.set('m4', [0.97, 2.048*1e-9, 0.149], res_num=10)
```

```
relax> value.set('m4', value=[0.97, 2.048*1e-9, 0.149], res_num=10)
```

To set the CSA value for the model-free run '`tm3`' to the default value, type:

```
relax> value.set('tm3', param='csa')
```

To set the CSA value of all residues in the reduced spectral density mapping run '600MHz' to -170 ppm, type:

```
relax> value.set('600MHz', -170 * 1e-6, 'csa')
```

```
relax> value.set('600MHz', value=-170 * 1e-6, param='csa')
```

To set the NH bond length of all residues in the model-free run 'm5' to 1.02 Å, type:

```
relax> value.set('m5', 1.02 * 1e-10, 'bond_length')
```

```
relax> value.set('m5', value=1.02 * 1e-10, param='r')
```

To set both the bond length and the CSA value for the run 'new' to the default values, type:

```
relax> value.set('new', param=['bond length', 'csa'])
```

To set both τ_f and τ_s in the model-free run 'm6' to 100 ps, type:

```
relax> value.set('m6', 100e-12, ['tf', 'ts'])
```

```
relax> value.set('m6', value=100e-12, param=['tf', 'ts'])
```

To set the S^2 and τ_e parameter values for model-free run 'm4' which has the parameters $\{S^2, \tau_e, R_{ex}\}$ to 0.56 and 13 ps, type:

```
relax> value.set('m4', [0.56, 13e-12], ['S2', 'te'], 10)
```

```
relax> value.set('m4', value=[0.56, 13e-12], param=['S2', 'te'], res_num=10)
```

```
relax> value.set(run='m4', value=[0.56, 13e-12], param=['S2', 'te'], res_num=10)
```

Regular expression

The python function 'match', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'[]' – A sequence or set of characters to match to a single character. For example, '[Ss]2' will match both 'S2' and 's2'.

'^' – Match the start of the string.

**'$'** – Match the end of the string. For example, '^[Ss]2$' will match 's2' but not 'S2f' or 's2s'.

'.' – Match any character.

'x*' – Match the character 'x' any number of times, for example 'x' will match, as will 'xxxxx'

'.*' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Model-free set details

Setting a parameter value may have no effect depending on which model-free model is chosen, for example if S_f^2 values and S_s^2 values are set but the run corresponds to model-free model '`m4`' then, because these data values are not parameters of the model, they will have no effect.

Note that the R_{ex} values are scaled quadratically with field strength and should be supplied as a field strength independent value. Use the following formula to get the correct value:

value = R_{ex} / (2.0 * π * frequency) ** 2

where:

R_{ex} is the chemical exchange value for the current frequency.

π is in the namespace of relax, ie just type '`pi`'.

frequency is the proton frequency corresponding to the data.

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Model-free default values

Data type	Object name	Value
Local τ_m	'tm'	10 * 1e-9
Order parameters S^2, S_f^2, and S_s^2	's2', 's2f', 's2s'	0.8
Correlation time τ_e	'te'	100 * 1e-12
Correlation time τ_f	'tf'	10 * 1e-12
Correlation time τ_s	'ts'	1000 * 1e-12
Chemical exchange relaxation	'rex'	0.0
Bond length	'r'	1.02 * 1e-10
CSA	'csa'	-170 * 1e-6

Reduced spectral density mapping set details

In reduced spectral density mapping, only two values can be set, the bond length and CSA value. These must be set prior to the calculation of spectral density values.

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping default values

Data type	Object name	Value
Bond length	'r'	1.02 * 1e-10
CSA	'csa'	-170 * 1e-6

Diffusion tensor set details

If the diffusion tensor has not been setup, use the more powerful function 'diffusion_tensor.init' to initialise the tensor parameters.

The diffusion tensor parameters can only be set when the run corresponds to model-free analysis. The units of the parameters are:

Inverse seconds for τ_m.

Seconds for $\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, $\mathfrak{D}_x$, $\mathfrak{D}_y$, $\mathfrak{D}_z$, $\mathfrak{D}_\parallel$, $\mathfrak{D}_\perp$.

Unitless for $\mathfrak{D}_{ratio}$ and $\mathfrak{D}_r$.

Radians for all angles (α, β, γ, θ, ϕ).

When setting a diffusion tensor parameter, the residue number has no effect. As the internal parameters of spherical diffusion are {tm}, spheroidal diffusion are {τ_m, $\mathfrak{D}_a$, θ, ϕ}, and ellipsoidal diffusion are {τ_m, $\mathfrak{D}_a$, $\mathfrak{D}_r$, α, β, γ}, supplying geometric parameters must be done in the following way. If a single geometric parameter is supplied, it must be one of τ_m, $\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, $\mathfrak{D}_r$, or $\mathfrak{D}_{ratio}$. For the parameters $\mathfrak{D}_\parallel$, $\mathfrak{D}_\perp$, $\mathfrak{D}_x$, $\mathfrak{D}_y$, and $\mathfrak{D}_x$, it is not possible to determine how to use the currently set values together with the supplied value to calculate the new internal parameters. For spheroidal diffusion, when supplying multiple geometric parameters, the set must belong to one of

$\{\tau_m, \mathfrak{D}_a\}$,

$\{\mathfrak{D}_{iso}, \mathfrak{D}_a\}$,

$\{\tau_m, \mathfrak{D}_{ratio}\}$,

$\{\mathfrak{D}_{\parallel}, \mathfrak{D}_{\perp}\}$,

$\{\mathfrak{D}_{iso}, \mathfrak{D}_{ratio}\}$,

where either θ, ϕ, or both orientational parameters can be additionally supplied. For ellipsoidal diffusion, again when supplying multiple geometric parameters, the set must belong to one of

$\{\tau_m, \mathfrak{D}_a, \mathfrak{D}_r\}$,

$\{\mathfrak{D}_{iso}, \mathfrak{D}_a, \mathfrak{D}_r\}$,

$\{\mathfrak{D}_x, \mathfrak{D}_y, \mathfrak{D}_z\}$,

where any number of the orientational parameters, α, β, or γ can be additionally supplied.

Diffusion tensor parameter string matching patterns

Data type	Object name	Patterns
Global correlation time - τ_m	'tm'	'tm'
Isotropic component of the diffusion tensor - $\mathfrak{D}_{iso}$	'Diso'	'[Dd]iso'
Anisotropic component of the diffusion tensor - $\mathfrak{D}_a$	'Da'	'[Dd]a'
Rhombic component of the diffusion tensor - $\mathfrak{D}_r$	'Dr'	'[Dd]r$'
Eigenvalue associated with the x-axis of the diffusion diffusion tensor - $\mathfrak{D}_x$	'Dx'	'[Dd]x'
Eigenvalue associated with the y-axis of the diffusion diffusion tensor - $\mathfrak{D}_y$	'Dy'	'[Dd]y'
Eigenvalue associated with the z-axis of the diffusion diffusion tensor - $\mathfrak{D}_z$	'Dz'	'[Dd]z'
Diffusion coefficient parallel to the major axis of the spheroid diffusion tensor - $\mathfrak{D}_{\parallel}$	'Dpar'	'[Dd]par'
Diffusion coefficient perpendicular to the major axis of the spheroid diffusion tensor - $\mathfrak{D}_{\perp}$	'Dper'	'[Dd]per'
Ratio of the parallel and perpendicular components of the spheroid diffusion tensor - $\mathfrak{D}_{ratio}$	'Dratio'	'[Dd]ratio'
The first Euler angle of the ellipsoid diffusion tensor - α	'alpha'	'^a$' or 'alpha'
The second Euler angle of the ellipsoid diffusion tensor - β	'beta'	'^b$' or 'beta'
The third Euler angle of the ellipsoid diffusion tensor - γ	'gamma'	'^g$' or 'gamma'
The polar angle defining the major axis of the spheroid diffusion tensor - θ	'theta'	'theta'
The azimuthal angle defining the major axis of the spheroid diffusion tensor - ϕ	'phi'	'phi'

Diffusion tensor parameter default values

Data type	Object name	Value
τ_m	'tm'	10 * 1e-9
$\mathfrak{D}_{iso}$	'Diso'	1.666 * 1e7
$\mathfrak{D}_a$	'Da'	0.0
$\mathfrak{D}_r$	'Dr'	0.0
$\mathfrak{D}_x$	'Dx'	1.666 * 1e7
$\mathfrak{D}_y$	'Dy'	1.666 * 1e7
$\mathfrak{D}_z$	'Dz'	1.666 * 1e7
$\mathfrak{D}_{\parallel}$	'Dpar'	1.666 * 1e7
$\mathfrak{D}_{\perp}$	'Dper'	1.666 * 1e7
$\mathfrak{D}_{ratio}$	'Dratio'	1.0
α	'alpha'	0.0
β	'beta'	0.0
γ	'gamma'	0.0
θ	'theta'	0.0
ϕ	'phi'	0.0

Relaxation curve fitting set details

Only three parameters can be set, the relaxation rate (Rx), the initial intensity (I0), and the intensity at infinity (Iinf). Setting the parameter Iinf has no effect if the chosen model is that of the exponential curve which decays to zero.

Relaxation curve fitting data type string matching patterns

Data type	Object name	Patterns
Relaxation rate	'rx'	'^[Rr]x$'
Initial intensity	'i0'	'^[Ii]0$'
Intensity at infinity	'iinf'	'^[Ii]inf$'

Relaxation curve fitting default values

These values are completely arbitrary as peak heights (or volumes) are extremely variable and the Rx value is a compensation for both the R_1 and R_1 values.

Data type	Object name	Value
Relaxation rate	'rx'	8.0
Initial intensity	'i0'	10000.0
Intensity at infinity	'iinf'	0.0

6.2.84 value.write

Synopsis

Function for writing residue specific data values to a file.

Defaults

value.write(self, run=None, param=None, file=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

param: The parameter.

file: The name of the file.

dir: The directory name.

force: A flag which, if set to 1, will cause the file to be overwritten.

Description

If no directory name is given, the file will be placed in the current working directory.

The parameter argument should be a string.

Examples

To write the CSA values for the run '`m1`' to the file '`csa.txt`', type:

```
relax> value.write('m1', 'CSA', 'csa.txt')

relax> value.write(run='m1', param='CSA', file='csa.txt')
```

To write the NOE values from the run '`noe`' to the file '`noe`', type:

```
relax> value.write('noe', 'noe', 'noe.out')

relax> value.write('noe', param='noe', file='noe.out')

relax> value.write(run='noe', param='noe', file='noe.out')

relax> value.write(run='noe', param='noe', file='noe.out', force=1)
```

Regular expression

The python function '`match`', which uses regular expression, is used to determine which data type to set values to, therefore various data_type strings can be used to select the same data type. Patterns used for matching for specific data types are listed below.

This is a short description of python regular expression, for more information see the regular expression syntax section of the Python Library Reference. Some of the regular expression syntax used in this function is:

'`[]`' – A sequence or set of characters to match to a single character. For example, '`[Ss]2`' will match both '`S2`' and '`s2`'.

'`^`' – Match the start of the string.

'`$`' – Match the end of the string. For example, '`^[Ss]2$`' will match '`s2`' but not '`S2f`' or '`s2s`'.

'`.`' – Match any character.

'`x*`' – Match the character '`x`' any number of times, for example '`x`' will match, as will '`xxxxx`'

'`.*`' – Match any sequence of characters of any length.

Importantly, do not supply a string for the data type containing regular expression. The regular expression is implemented so that various strings can be supplied which all match the same data type.

Model-free data type string matching patterns

Data type	Object name	Patterns
Local τ_m	'tm'	'^tm$'
Order parameter S^2	's2'	'^[Ss]2$'
Order parameter S_f^2	's2f'	'^[Ss]2f$'
Order parameter S_s^2	's2s'	'^[Ss]2s$'
Correlation time τ_e	'te'	'^te$'
Correlation time τ_f	'tf'	'^tf$'
Correlation time τ_s	'ts'	'^ts$'
Chemical exchange	'rex'	'^[Rr]ex$' or '[Cc]emical[-_][Ee]xchange'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
$J(0)$	'j0'	'^[Jj]0$' or '[Jj](0)'
$J(\omega_X)$	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
$J(\omega_H)$	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

NOE calculation data type string matching patterns

Data type	Object name	Patterns
Reference intensity	'ref'	'^[Rr]ef$' or '[Rr]ef[-_][Ii]nt'
Saturated intensity	'sat'	'^[Ss]at$' or '[Ss]at[-_][Ii]nt'
NOE	'noe'	'^[Nn][Oo][Ee]$'

Relaxation curve fitting data type string matching patterns

Data type	Object name	Patterns
Relaxation rate	'rx'	'^[Rr]x$'
Initial intensity	'i0'	'^[Ii]0$'
Intensity at infinity	'iinf'	'^[Ii]inf$'

6.2.85 vmd.view

Synopsis

Function for viewing the collection of molecules extracted from the PDB file.

Defaults

vmd.view(self, run=None)

Keyword Arguments

run: The name of the run which the PDB belongs to.

Example

```
relax> vmd.view('m1')
relax> vmd.view(run='pdb')
```

Chapter 7

Licence

7.1 Copying, modification, sublicencing, and distribution of relax

To ensure that the program relax, including all future versions, will remain legally available for perpetuity to anyone who wishes to use the program, it has been decided to release the code under the GNU General Public Licence. The freedom of relax is guaranteed by the GPL. This is a licence which has been very carefully crafted to protect both the author of the program as well as the public by means of copyright law. If the licence is violated by improper copying, modification, sublicencing, or distribution then the licence terminates – hence the violator is copying, modifying, sublicencing, or distributing the program illegally in full violation of copyright law. For a better understanding of the protections afforded by the GPL, the licence is reprinted in whole within the next section.

7.2 The GPL

The following is a verbatim copy of the GNU General Public Licence. A text version is available in the relax 'docs' directory within the file 'COPYING'.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those

sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published

by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
```

```
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘`show w`’ and ‘`show c`’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bibliography

Anatole Abragam. *The Principles of Nuclear Magnetism*. Clarendon Press, Oxford, 1961.

G.M. Clore, A. Szabo, A. Bax, L.E. Kay, P.C. Driscoll, and A.M. Gronenborn. Deviations from the simple 2-parameter model-free approach to the interpretation of N-15 nuclear magnetic-relaxation of proteins. *Journal of the American Chemical Society*, 112(12): 4989 – 4991, JUN 6 1990. ISSN 0002-7863.

G. Lipari and A. Szabo. Model-free approach to the interpretation of nuclear magnetic-resonance relaxation in macromolecules I. Theory and range of validity. *Journal of the American Chemical Society*, 104(17):4546 – 4559, 1982a. ISSN 0002-7863.

G. Lipari and A. Szabo. Model-free approach to the interpretation of nuclear magnetic-resonance relaxation in macromolecules II. Analysis of experimental results. *Journal of the American Chemical Society*, 104(17):4559 – 4570, 1982b. ISSN 0002-7863.

Index