relax

Edward d'Auvergne

October 15, 2005

Contents

Chapter 1

Alphabetical listing of user functions

The following is a listing with descriptions of all the user functions availible 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)`'.

1.1 The help system

For assistance in using a function, simply type '`help(function)`'. All functions can be viewed by hitting the [TAB] key. 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()`'.

1.2 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.

1.3 The list of functions

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

1.3.1 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.

1.3.2 The default argments

The default arguments list all the argments taken by the function. To invoke the function, type the function name (tab completion is implemented to prevent insanity as the function names can be quite long – a deliberate feature to improve usability), then in brakets 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.

Two types of arguments exist in Python, standard arguments and keyword arguments. The majority of arguments within the relax user functions 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.

1.3.3 angles

Synopsis

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

Default arguments

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 fully anisotropic, then the three angles will be calculated.

1.3.4 calc

Synopsis

Function for calculating the function value.

Default arguments

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

Keyword Arguments

run: The name of the run.

1.3.5 diffusion_tensor.copy

Synopsis

Function for copying diffusion tensor data from run1 to run2.

Default arguments

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

Keyword Arguments

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

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')
```

1.3.6 diffusion_tensor.delete

Synopsis

Function for deleting diffusion tensor data.

Default arguments

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.

1.3.7 diffusion_tensor.display

Synopsis

Function for displaying the diffusion tensor.

Default arguments

diffusion_tensor.display(self, run=None)

Keyword Arguments

run: The name of the run.

1.3.8 diffusion_tensor.set

Synopsis

Function for setting up the diffusion tensor.

Default arguments

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

Keyword Arguments

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

time_scale: The correlation time scaling value.

angle_units: The units for the angle parameters.

axial_type: A string, which if supplied with axially symmetric parameters, will restrict the tensor to either being `'oblate'` or `'prolate'` .

Description

Isotropic diffusion.

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

Axially symmetric diffusion.

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 - (τ_m, $\mathfrak{D}_{ratio}$, θ, ϕ) 2 - ($\mathfrak{D}_\parallel$, $\mathfrak{D}_\perp$, θ, ϕ) 3 - ($\mathfrak{D}_{iso}$, $\mathfrak{D}_a$, θ, ϕ) 4 - ($\mathfrak{D}_{iso}$, $\mathfrak{D}_{ratio}$, θ, ϕ)

where: $\tau_m = 1 \ / \ 6\mathfrak{D}_{iso}$ $\mathfrak{D}_{iso} = 1/3 \ (\mathfrak{D}_\parallel + 2\mathfrak{D}_\perp)$ $\mathfrak{D}_a = 1/3 \ (\mathfrak{D}_\parallel - \mathfrak{D}_\perp)$ $\mathfrak{D}_{ratio} = \mathfrak{D}_\parallel \ / \ \mathfrak{D}_\perp$

The diffusion tensor is defined by the vector $\mathfrak{D}_\parallel$. The angle α describes the bond vector with respect to the diffusion frame while the spherical angles $\{\theta, \phi\}$ describe the diffusion

tensor with respect to the PDB frame. Theta is the polar angle and ϕ is the azimuthal angle defined between: $0 \leq \theta \leq \pi$ $0 \leq \phi \leq 2\pi$ The angle α is defined between: $0 \leq \alpha \leq 2\pi$

The `'axial_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 $\mathfrak{D}_\perp \geq \mathfrak{D}_\parallel$ is used. If `'prolate'` is given, then the constraint $\mathfrak{D}_\perp \leq \mathfrak{D}_\parallel$ is used. If nothing is supplied, then $\mathfrak{D}_\perp$ and $\mathfrak{D}_\parallel$ 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 to be minimised, thereby partitioning the two minima into the two subspaces (the partition is where $\mathfrak{D}_a$ equals 0).

Anisotropic diffusion.

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: $\tau_m = 1 \ / \ 6\mathfrak{D}_{iso}$ $\mathfrak{D}_{iso} = 1/3\ (\mathfrak{D}_x + \mathfrak{D}_y + \mathfrak{D}_z)$ $\mathfrak{D}_a = 1/3\ (\mathfrak{D}_z$ - $(\mathfrak{D}_x + \mathfrak{D}_y)/2)$ $\mathfrak{D}_r = (\mathfrak{D}_x$ - $\mathfrak{D}_y)/2$

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 bond vector is described using the spherical angels 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. Parameters affected by this value are: τ_m.

The `'d_scale'` argument should also be a floating point number. Parameters affected by this value are: $\mathfrak{D}_{iso}$; $\mathfrak{D}_\parallel$; $\mathfrak{D}_\perp$; $\mathfrak{D}_a$; $\mathfrak{D}_r$; $\mathfrak{D}_x$; $\mathfrak{D}_y$; $\mathfrak{D}_z$.

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

Examples

To set an isotropic diffusion tensor with a correlation time of $10ns$, 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 8.5ns, $\mathfrak{D}_{ratio}$ 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 an axially symmetric 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 `'axial'` , type one of:

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

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

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

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

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

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

To select fully anisotropic 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 an isotropic diffusion tensor, type (followed by a minimisation command):

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

1.3.9 dx.execute

Synopsis

Function for running OpenDX.

Default arguments

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. be run in the current directory.

dx_exe: The OpenDX executable file. start-up. The default is 1 which turns execution on. Setting the value to zero turns execution off.

1.3.10 dx.map

Synopsis

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

Default arguments

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

Keyword Arguments

run: The name of the run.

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

swap: An array used to swap the position of the axes. The length of the array should be the same as the number of parameters in the model. The values should be integers specifying which elements to interchange. For example if swap equals [0, 1, 2] for a three parameter model then the axes are not interchanged whereas if swap equals [1, 0, 2] then the first and second dimensions are interchanged. 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. 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. and must return an array of equal length.

labels: The axis labels. If supplied this argument should be an array of strings of length equal to the number of parameters.

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 $\{S_f^2, S_s^2, \tau_s\}$. Files will be output into the directory `'dx'` and will be prefixed by `'map'` . The residue, in this case, is number 6.

```
relax> map('m5', 6)

relax> map('m5', 6, 20, "map", "dx")

relax> map('m5', res_num=6, file="map", dir="dx")

relax> map(run='m5', res_num=6, inc=20, file="map", dir="dx")

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

The following commands will swap the S_s^2 and τ_s axes of this map.

```
relax> map('m5', res_num=6, swap=[0, 2, 1])

relax> map(run='m5', res_num=6, type="Iso3D", inc=20, swap=[0, 2, 1], 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> map('m4', res_num=2, file='test', dir=None)

relax> map(run='m4', res_num=2, inc=100, file='test', dir=None)
```

1.3.11 eliminate

Synopsis

Function for model elimination.

Default arguments

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.

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.

Model-free model elimination rules

Local τ_m.

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:

$\tau_m \geq$ c

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

Internal correlation times $\{\tau_e, \tau_f, \tau_s\}$.

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 ($\tau_e, \tau_f, \tau_s \leq 2\tau_m$). 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).

1.3.12 fix

Synopsis

Function for either fixing or allowing parameter values to change.

Default arguments

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

Keyword Arguments

run: The name of the run.

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.

1.3.13 grace.view

Synopsis

Function for running Grace.

Default arguments

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

Keyword Arguments

file: The name of the file.

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')
```

1.3.14 grace.write

Synopsis

Function for creating a grace '.agr' file.

Default arguments

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.

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

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

file: The name of the file.

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’ .

ˆ - Match the start of the string.

$ - Match the end of the string. For example, ‘ˆ[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$’ or ‘[Jj](0)’
J(wX)	‘jwx’	‘^[Jj]w[Xx]$’ or ‘[Jj](w[Xx])’
J(wH)	‘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]$’

1.3.15 grid_search

Synopsis

The grid search function.

Default arguments

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. 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. 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). output while higher values increase the amount of output. The default value is 1.

1.3.16 init_data

Synopsis

Function for reinitialising self.relax.data

Default arguments

init_data(self)

1.3.17 intro_off

Synopsis

Function for turning the function introductions off.

Default arguments

intro_off(self)

1.3.18 intro_on

Synopsis

Function for turning the function introductions on.

Default arguments

intro_on(self)

1.3.19 jw_mapping.set_frq

Synopsis

Function for selecting which relaxation data to use in the J(w) mapping.

Default arguments

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

Keyword Arguments

run: The name of the run.

Description

This function will select the relaxation data to use in the reduced spectral densiy 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)
```

1.3.20 minimise

Synopsis

Minimisation function.

Default arguments

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. 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.

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.

Description

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 extemely 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*6*1e8)**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)
```

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.

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.

d2func: The function which returns the Hessian.

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

min_options: A tuple to pass to the minimisation function as the min_options keyword. below this value, minimisation is terminated.

grad_tol: The gradient tolerance value.

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

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

c: User supplied constraint function.

d2c: User supplied constraint Hessian function. 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. 0 means no output, 1 means minimal output, and values above 1 increase the amount of output printed.

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'` .

ˆ - 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-Ribiere	`'^[Pp][Rr]$'` or `'^[Pp]olak[-_ ][Rr]ibiere$'`
Polak-Ribiere +	`'^[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.

1.3.21 model_free.copy

Synopsis

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

Default arguments

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

Keyword Arguments

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

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')
```

1.3.22 model_free.create_model

Synopsis

Function to create a model-free model.

Default arguments

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.

equation: The model-free equation.

res_num: The residue number.

Description

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: S^2: The square of the generalised order parameter. τ_e: The effective correlation time. The following parameters are accepted for the extended model-free equation: S_f^2: The square of the generalised order parameter of the faster motion. τ_f: The effective correlation time of the faster motion. S^2: The square of the generalised order parameter $S^2 = S_f^2 * S_s^2$. τ_s: The effective correlation time of the slower motion. The following parameters are accepted for the extended 2 model-free equation: S_f^2: The square of the generalised order parameter of the faster motion. τ_f: The effective correlation time of the faster motion. S_s^2: The square of the generalised order parameter of the slower motion. τ_s: The effective correlation time of the slower motion. The following parameters are accepted for all equations: R_{ex}: 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')
```

1.3.23 model_free.delete

Synopsis

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

Default arguments

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')
```

1.3.24 model_free.remove_tm

Synopsis

Function for removing the local τ_m parameter from a model.

Default arguments

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

Keyword Arguments

run: The run to assign the values to.

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')
```

1.3.25 model_free.select_model

Synopsis

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

Default arguments

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

Keyword Arguments

run: The run to assign the values to.

Description

The 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}]

'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}]

'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}]

'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 R1, R2, 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' =¿ [τ_m]

'tm1' =¿ [τ_m, S^2]

'tm2' =¿ [τ_m, S^2, τ_e]

'tm3' =¿ [τ_m, S^2, R_{ex}]

'tm4' =¿ [τ_m, S^2, τ_e, R_{ex}]

'tm5' =¿ [τ_m, S_f^2, S^2, τ_s]

'tm6' =¿ [τ_m, S_f^2, τ_f, S^2, τ_s]

'tm7' =¿ [τ_m, S_f^2, S^2, τ_s, R_{ex}]

'tm8' =¿ [τ_m, S_f^2, τ_f, S^2, τ_s, R_{ex}]

'tm9' =¿ [τ_m, R_{ex}]

'tm10' =¿ [τ_m, CSA]

'tm11' =¿ [τ_m, CSA, S^2]

'tm12' =¿ [τ_m, CSA, S^2, τ_e]

'tm13' =¿ [τ_m, CSA, S^2, R_{ex}]

'tm14' =¿ [τ_m, CSA, S^2, τ_e, R_{ex}]

'tm15' =¿ [τ_m, CSA, S_f^2, S^2, τ_s]

'tm16' =¿ [τ_m, CSA, S_f^2, τ_f, S^2, τ_s]

'tm17' =¿ [τ_m, CSA, S_f^2, S^2, τ_s, R_{ex}]

'tm18' =¿ [τ_m, CSA, S_f^2, τ_f, S^2, τ_s, R_{ex}]

'tm19' =¿ [τ_m, CSA, R_{ex}]

'tm20' =¿ [τ_m, r]

'tm21' =¿ [τ_m, r, S^2]

'tm22' =¿ [τ_m, r, S^2, τ_e]

'tm23' =¿ [τ_m, r, S^2, R_{ex}]

'tm24' =¿ [τ_m, r, S^2, τ_e, R_{ex}]

'tm25' =¿ [τ_m, r, S_f^2, S^2, τ_s]

'tm26' =¿ [τ_m, r, S_f^2, τ_f, S^2, τ_s]

'tm27' =¿ [τ_m, r, S_f^2, S^2, τ_s, R_{ex}]

'tm28' =¿ [τ_m, r, S_f^2, τ_f, S^2, τ_s, R_{ex}]

'tm29' =¿ [τ_m, r, CSA, R_{ex}]

'tm30' =¿ [τ_m, r, CSA]

'tm31' =¿ [τ_m, r, CSA, S^2]

'tm32' =¿ [τ_m, r, CSA, S^2, τ_e]

'tm33' =¿ [τ_m, r, CSA, S^2, R_{ex}]

'tm34' =¿ [τ_m, r, CSA, S^2, τ_e, R_{ex}]

'tm35' =¿ [τ_m, r, CSA, S_f^2, S^2, τ_s]

'tm36' =¿ [τ_m, r, CSA, S_f^2, τ_f, S^2, τ_s]

'tm37' =¿ [τ_m, r, CSA, S_f^2, S^2, τ_s, R_{ex}]

'tm38' =¿ [τ_m, r, CSA, S_f^2, τ_f, S^2, τ_s, R_{ex}]

'tm39' =¿ [τ_m, r, 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')
```

1.3.26 model_selection

Synopsis

Function for model selection.

Default arguments

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

Keyword arguments

method: The model selection technique (see below).

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'])
```

1.3.27 molmol.clear_history

Synopsis

Function for clearing the Molmol command history.

Default arguments

molmol.clear_history(self)

1.3.28 molmol.command

Synopsis

Function for executing a user supplied Molmol command.

Default arguments

molmol.command(self, command)

Example

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

1.3.29 molmol.view

Synopsis

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

Default arguments

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')
```

1.3.30 monte_carlo.create_data

Synopsis

Function for creating simulation data.

Default arguments

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

Keyword Arguments

run: The name of the run.

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 accidently 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.
```

1.3.31 monte_carlo.error_analysis

Synopsis

Function for calculating parameter errors from the Monte Carlo simulations.

Default arguments

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

Keyword Arguments

run: The name of the run.

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 argment 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 prunning process and hence cause a catastropic 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 accidently 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.
```

1.3.32 monte_carlo.initial_values

Synopsis

Function for setting the initial simulation parameter values.

Default arguments

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 accidently 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 accidently 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.
```

1.3.33 monte_carlo.off

Synopsis

Function for turning simulations off.

Default arguments

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 accidently 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.
```

1.3.34 monte_carlo.on

Synopsis

Function for turning simulations on.

Default arguments

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 accidently 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.
```

1.3.35 monte_carlo.setup

Synopsis

Function for setting up Monte Carlo simulations.

Default arguments

monte_carlo.setup(self, run=None, number=500)

Keyword Arguments

run: The name of the run.

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 accidently 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.
```

1.3.36 noe.error

Synopsis

Function for setting the errors in the reference or saturated NOE spectra.

Default arguments

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.

spectrum_type: The type of spectrum.

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.

1.3.37 noe.read

Synopsis

Function for reading peak intensities from a file for NOE calculations.

Default arguments

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.

dir: The directory where the file is located.

format: The type of file containing peak intensities.

proton: The name of the proton as specified in the peak intensity 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')
```

1.3.38 nuclei

Synopsis

Function for setting the gyromagnetic ratio of the heteronucleus.

Default arguments

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

1.3.39 palmer.create

Synopsis

Function for creating the Modelfree4 input files.

Default arguments

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.

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

sims: The number of Monte Carlo simulations.

trim: See the Modelfree4 manual.

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

atom1: The symbol of the X 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'`

This can be used to execute modelfree4.

1.3.40 palmer.execute

Synopsis

Function for executing Modelfree4.

Default arguments

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

Keyword Arguments

run: The name of the 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'` .

1.3.41 palmer.extract

Synopsis

Function for extracting data from the Modelfree4 'mfout' star formatted file.

Default arguments

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

Keyword Arguments

run: The name of the run.

1.3.42 pdb

Synopsis

The pdb loading function.

Default arguments

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.

dir: The directory where the file is located.

heteronuc: The name of the heteronucleus 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)
```

1.3.43 relax_data.back_calc

Synopsis

Function for back calculating relaxation data.

Default arguments

relax_data.back_calc(self, run=None, ri_label=None, frq_label=None, frq=None)

Keyword Arguments

run: The name of the run.

frq_label: The field strength label.

1.3.44 relax_data.copy

Synopsis

Function for copying relaxation data from run1 to run2.

Default arguments

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.

ri_label: The relaxation data type, ie `'R1'` , `'R2'` , or `'NOE'` .

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')
```

1.3.45 relax_data.delete

Synopsis

Function for deleting the relaxation data corresponding to ri_label and frq_label.

Default arguments

relax_data.delete(self, run=None, ri_label=None, frq_label=None)

Keyword Arguments

run: The name of the run.

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')
```

1.3.46 relax_data.display

Synopsis

Function for displaying the relaxation data corresponding to ri_label and frq_label.

Default arguments

relax_data.display(self, run=None, ri_label=None, frq_label=None)

Keyword Arguments

run: The name of the run.

frq_label: The field strength label.

Examples

To show the relaxation data corresponding to ri_label= `'NOE'` , frq_label= `'600'` , and the run `'m4'` , type:

```
relax> relax_data.display('m4', 'NOE', '600')
```

1.3.47 relax_data.read

Synopsis

Function for reading R1, R2, or NOE relaxation data from a file.

Default arguments

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.

frq_label: The field strength label.

file: The name of the file containing the relaxation data.

num_col: The residue number column (the default is 0, ie the first column).

data_col: The relaxation data column (the default is 2).

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 R2 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 R1 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='%')
```

1.3.48 relax_data.write

Synopsis

Function for writing R1, R2, or NOE relaxation data to a file.

Default arguments

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.

frq_label: The field strength label.

dir: The directory name.

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.

1.3.49 relax_fit.read

Synopsis

Function for reading peak intensities from a file.

Default arguments

relax_fit.read(self, run=None, file=None, dir=None, relax_time=0.0, fit_type='exp', format='sparky', heteronuc='N', proton='HN', int_col=None)

Keyword Arguments

run: The name of the run.

dir: The directory where the file is located.

fit_type: The type of relaxation curve to fit.

heteronuc: The name of the heteronucleus 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 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 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.

1.3.50 results.display

Synopsis

Function for displaying the results of the run.

Default arguments

results.display(self, run=None, format='columnar')

Keyword Arguments

run: The name of the run.

1.3.51 results.read

Synopsis

Function for reading results from a file.

Default arguments

results.read(self, run=None, file='results', dir='run', format='columnar')

Keyword Arguments

run: The name of the run.

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.

1.3.52 results.write

Synopsis

Function for writing results of the run to a file.

Default arguments

results.write(self, run=None, file='results', dir='run', force=0, format='columnar', compress_type=1)

Keyword Arguments

run: The name of the run.

dir: The directory name.

format: The format of the output.

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.

1.3.53 run.create

Synopsis

Function for setting up a run type.

Default arguments

run.create(self, run=None, run_type=None)

Keyword Arguments

run: The name of the 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')
```

1.3.54 run.delete

Synopsis

Function for deleting a run.

Default arguments

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.

1.3.55 select.all

Synopsis

Function for selecting all residues.

Default arguments

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')
```

1.3.56 select.read

Synopsis

Function for selecting the residues contained in a file.

Default arguments

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.

dir: The directory where the file is located.

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')
```

1.3.57 select.res

Synopsis

Function for selecting specific residues.

Default arguments

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.

name: The residue name.

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')
```

1.3.58 select.reverse

Synopsis

Function for the reversal of the residue selection.

Default arguments

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()
```

1.3.59 sequence.add

Synopsis

Function for adding a residue onto the sequence.

Default arguments

sequence.add(self, run=None, res_num=None, res_name=None, select=1)

Keyword Arguments

run: The name of the run.

res_name: The name of the residue.

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')
```

1.3.60 sequence.copy

Synopsis

Function for copying the sequence from run1 to run2.

Default arguments

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

Keyword Arguments

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

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')
```

1.3.61 sequence.delete

Synopsis

Function for deleting the sequence.

Default arguments

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.

1.3.62 sequence.display

Synopsis

Function for displaying the sequence.

Default arguments

sequence.display(self, run=None)

Keyword Arguments

run: The name of the run.

1.3.63 sequence.read

Synopsis

Function for reading sequence data.

Default arguments

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.

dir: The directory where the file is located.

name_col: The residue name column (the default is 1).

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=',')
```

1.3.64 sequence.sort

Synopsis

Function for numerically sorting the sequence by residue number.

Default arguments

sequence.sort(self, run=None)

Keyword Arguments

run: The name of the run.

1.3.65 sequence.write

Synopsis

Function for writing the sequence to a file.

Default arguments

sequence.write(self, run=None, file=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

dir: The directory name.

Description

If no directory name is given, the file will be placed in the current working directory.

1.3.66 state.load

Synopsis

Function for loading a saved program state.

Default arguments

state.load(self, file=None, dir=None)

Keyword Arguments

file: The file name, which must be a string, of a saved program state.

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')
```

1.3.67 state.save

Synopsis

Function for saving the program state.

Default arguments

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.

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)
```

1.3.68 system

Synopsis

Function which executes the user supplied shell command.

Default arguments

system(command)

1.3.69 thread.read

Synopsis

Function for reading a file containing entries for each computer to run calculations on.

Default arguments

thread.read(self, file='hosts', dir='~/.relax')

Keyword Arguments

file: The name of the file containing the host entries.

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"
```

1.3.70 unselect.all

Synopsis

Function for unselecting all residues.

Default arguments

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')
```

1.3.71 unselect.read

Synopsis

Function for unselecting the residues contained in a file.

Default arguments

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.

dir: The directory where the file is located.

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')
```

1.3.72 unselect.res

Synopsis

Function for unselecting specific residues.

Default arguments

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.

name: The residue name.

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')
```

1.3.73 unselect.reverse

Synopsis

Function for the reversal of the residue selection.

Default arguments

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()
```

1.3.74 value.copy

Synopsis

Function for copying residue specific data values from run1 to run2.

Default arguments

value.copy(self, run1=None, run2=None, data_type=None)

Keyword Arguments

run1: The name of the run to copy from.

data_type: The data type.

Description

Only one data type may be selected, therefore the data type 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'` .

ˆ - 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$’ or ‘[Bb]ond[-_][Ll]ength’
CSA	‘csa’	‘^[Cc][Ss][Aa]$’

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.

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
J(0)	`'j0'`	`'^[Jj]0$'` or `'[Jj](0)'`
J(wX)	`'jwx'`	`'^[Jj]w[Xx]$'` or `'[Jj](w[Xx])'`
J(wH)	`'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 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.

1.3.75 value.display

Synopsis

Function for displaying residue specific data values.

Default arguments

value.display(self, run=None, data_type=None)

Keyword Arguments

run: The name of the run.

Description

Only one data type may be selected, therefore the data type 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'` .

ˆ - Match the start of the string.

$ - Match the end of the string. For example, `'ˆ[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$' or '[Jj](0)'
J(wX)	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
J(wH)	'jwh'	'^[Jj]w[Hh]$' or '[Jj](w[Hh])'
Bond length	'r'	'^r$' or '[Bb]ond[-_][Ll]ength'
CSA	'csa'	'^[Cc][Ss][Aa]$'

1.3.76 value.read

Synopsis

Function for reading residue specific data values from a file.

Default arguments

value.read(self, run=None, data_type=None, file=None, num_col=0, name_col=1, data_col=2, error_col=3, sep=None)

Keyword Arguments

run: The name of the run.

frq: The spectrometer frequency in Hz.

num_col: The residue number column (the default is 0, ie the first column).

data_col: The relaxation data column (the default is 2).

sep: The column separator (the default is white space).

Description

Only one data type may be selected, therefore the data type 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', data_type='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'` .

ˆ - 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$'` or `'[Bb]ond[ -_][Ll]ength'`
CSA	`'csa'`	`'^[Cc][Ss][Aa]$'`

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.

Reduced spectral density mapping data type string matching patterns

Data type	Object name	Patterns
J(0)	'j0'	'^[Jj]0$' or '[Jj](0)'
J(wX)	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
J(wH)	'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 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.

1.3.77 value.set

Synopsis

Function for setting residue specific data values.

Default arguments

value.set(self, run=None, value=None, data_type=None, res_num=None, res_name=None)

Keyword arguments

run: The run to assign the values to.

data_type: The data type(s).

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 data type 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 Type column refers to the `'data_type'` 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	Type	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 data type matching the string will be set to the default value.
1	1	The data type matching the string will be set to the supplied number.
n	1	Invalid combination.
None	n	Each data type matching the strings will be set to the default values.
1	n	Each data type matching the strings will be set to the supplied number.
n	n	Each data type 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.

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', data_type='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, data_type='csa')
```

To set the NH bond length of all residues in the model-free run `'m5'` to 1.02 Angstroms, type:

```
relax> value.set('m5', 1.02 * 1e-10, 'bond_length')

relax> value.set('m5', value=1.02 * 1e-10, data_type='r')
```

To set both the bond length and the CSA value for the run `'new'` to the default values, type:

```
relax> value.set('new', data_type=['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, data_type=['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], data_type=['S2', 'te'], res_num=10)

relax> value.set(run='m4', value=[0.56, 13e-12], data_type=['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'` .

ˆ - 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$'` or `'[Bb]ond[ -_][Ll]ength'`
CSA	`'csa'`	`'^[Cc][Ss][Aa]$'`

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 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 data type string matching patterns

Data type	Object name	Patterns
J(0)	'j0'	'^[Jj]0$' or '[Jj](0)'
J(wX)	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
J(wH)	'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 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 default values

Data type	Object name	Value
Bond length	'r'	1.02 * 1e-10
CSA	'csa'	-170 * 1e-6

1.3.78 value.write

Synopsis

Function for writing residue specific data values to a file.

Default arguments

value.write(self, run=None, data_type=None, file=None, dir=None, force=0)

Keyword Arguments

run: The name of the run.

file: The name of the file.

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 data type 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', data_type='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', data_type='noe', file='noe.out')
relax> value.write(run='noe', data_type='noe', file='noe.out')
relax> value.write(run='noe', data_type='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' .

ˆ - 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$' or '[Jj](0)'
J(wX)	'jwx'	'^[Jj]w[Xx]$' or '[Jj](w[Xx])'
J(wH)	'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]$'

1.3.79 vmd.view

Synopsis

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

Default arguments

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')
```