MolSpin User Manual

August 2019

Claus Nielsen
University of Southern Denmark

Table of Contents

1 Installation **4**

1.1 Linux . . . 4

1.2 Windows . . . 4

1.3 Testing your MolSpin installation . . . 4

2 Getting Started **5**

2.1 Input file basics . . . 5

2.1.1 Overall structure: Objects and Object Groups . . . 5

2.1.2 Comments . . . 6

2.1.3 Case sensitivity and strings . . . 6

2.1.4 Input file example . . . 6

2.2 Running MolSpin . . . 10

2.2.1 Commandline parameters . . . 10

3 Specifying the spin system **12**

3.1 Spins . . . 12

3.2 Interactions . . . 13

3.3 Transitions . . . 13

3.4 Operators . . . 14

3.5 States . . . 15

3.6 Properties . . . 16

3.7 Specifying Tensors . . . 16

3.8 Specifying Trajectories . . . 18

3.8.1 Interaction trajectories . . . 18

3.8.2 Tensor trajectories . . . 19

3.8.3 Transition trajectories . . . 19

3.8.4 Spin trajectories . . . 19

3.9 ActionTargets: ActionVectors and ActionScalars . . . 20

4 Specifying the settings object group **21**

4.1 The Settings object . . . 21

4.2 Actions . . . 21

4.3 Output requests . . . 22

5 The Run section and task classes **24**

5.1 Overview of task classes . . . 24

5.2 General parameters . . . 24

5.3 Task class: DynamicHS-TimeEvolution . . . 25

5.4 Task class: PeriodicHS-TimeEvolution . . . 26

5.5 Task class: MultiDynamicHS-TimeEvolution . . . 28

5.6 Task class: StaticSS-TimeEvolution . . . 28

5.7 Task class: PeriodicSS-TimeEvolution . . . 29

5.8 Task class: MultiStaticSS-TimeEvolution . . . 30

5.9 Task class: StaticSS . . . 31

5.10 Task class: StaticHS-SymmetricDecay . . . 32

5.11 Task class: RP-SymmetricUncoupled . . . 33

5.12 Task class: Gamma-Compute . . . 34

5.13 Task class: HamiltonianEigenvalues . . . 34

6 Preprocessor directives **36**
6.1 #include <filename> . . . 36
6.2 #define <name> <value (optional)> . . . 36
6.3 #undef <name> . . . 36
6.4 #ifdef <name> . . . 36
6.5 #ifndef <name> . . . 36
6.6 #else . . . 37
6.7 #endif . . . 37

1 Installation

1.1 Linux

This software package relies on the Armadillo C++ Linear Algebra Library, which needs to be installed on the system prior to compilation of the MolSpin application. Furthermore, it is suggested that OpenBLAS and LAPACK are also installed on the system, and you should make sure that they are recognized by Armadillo when installing that package (when you install Armadillo it will tell you whether it has detected those packages).

You can use other high-speed math libraries like Intel MKL instead of OpenBLAS, but you may need to modify the makefile in that case. Note also if you are *not* using OpenBLAS that you need to change *main.cpp*: import and usage of the function “openblas_set_num_threads” should be removed (it is just 2 lines that should be removed).

To compile the MolSpin application, navigate to the folder containing *makefile* and *main.cpp* in a terminal, and type **make**. If everything works, you will then have the executable file called “molspin” in the folder. If it does not work, you may need to change the makefile or configure your Armadillo installation.

Note that MolSpin was developed using the GCC 5.4.0 compiler and requires a C++14-compatible compiler. The Armadillo version should be 8.5 or newer.

Note also that installation (or rather compilation) of MolSpin does not generate any files outside the MolSpin folder containing the source code.

1.2 Windows

In Windows it would be easiest to use the Windows Subsystem for Linux (WSL) or Cygwin for compiling and running MolSpin. I have not tried compiling MolSpin on Windows outside WSL/Cygwin.

1.3 Testing your MolSpin installation

Once MolSpin is installed, you should check that everything works by running **make test**.

2 Getting Started

2.1 Input file basics

To run MolSpin you must first prepare an input file which describes the spin system(s) you want to investigate, settings for your calculations, and the type of calculations you want to perform. In order to make this as easy an intuitive as possible, MolSpin uses a very elaborate format for input files, which in some ways look similar to the C programming language - but of course much simpler. Employing such an elaborate input format has the effect that input files may become rather large, and therefore you should know that it is possible to split the input into multiple files using the #include preprocessor directive described in section 6.1[1].

The following subsections walk you through the main elements of the input file formats.

2.1.1 Overall structure: Objects and Object Groups

All input in MolSpin is organized in *objects*, and each object must belong to an *object group*. The same syntax is used for defining objects and object groups: first a keyword and a name is specified, separated by a space, and then the contents of the object or object group is contained between { and }. The keyword denotes the type of object or object group, and the name is used to refer to it from other parts of the input file, or from the output or error messages from MolSpin. Here is an example where a *SpinSystem* object group is created, containing a *Spin* object.

```
SpinSystem MySystem
{
  Spin SomeParticle
  {
    spin = 1/2;
    type = electron;
  }
}
```

Indentation (spaces and tabs) and line breaks are ignored by MolSpin, so the same definitions could be written as:

```
SpinSystem MySystem{Spin SomeParticle{spin=1/2;type=electron;}}
```

Although this way of writing the input is valid, it is not very easy to see what is going on, so it is recommended to format your input files like the first example above!

There are only three types of object groups: **SpinSystem**, **Settings** and **Run**, and note that the **Run** group, which is also referred to as the **run section** is special since it cannot be assigned a name[2]. There is no limit to the number of objects that can be assigned to an object group, and it is possible to split an object group between multiple files by simply creating object groups of the same type (e.g. SpinSystem) using the same name.

Different object groups contain different types of objects: SpinSystem groups contain objects of the types Spin, Interaction, Transition, State and a single Properties object. Settings groups contains a Settings object and any number of Action and Output objects, and the Run section contains the Task objects that describe what kinds of calculations to perform. Sections 3, 4 and 5 describes all the details about how these objects work, and you should be aware that Action objects are crucial for defining the behaviour of multi-step calculations, and is the main source of the great flexibility that MolSpin offers.

[1]Note also that you can use #define directive to affect parameters defined in included files.
[2]All objects and both of the other object groups must be given a name; the run section is the only exception.

2.1.2 Comments

The MolSpin input format uses the same conventions for comments as in the C programming language: two slashes (//) are used for single-line comments, such that the rest of the line after // is considered a comment, and /* */ for multi-line or in-line comments. Here is example:

```
This is not a comment // But this IS a comment!
Not a comment /* this is a comment */ Not a comment again!
/*
Here is a comment
that span multiple lines!
*/
```

2.1.3 Case sensitivity and strings

The input format is **not case senstive** for normal input, but there is one notable **exception**: **strings** are case sensitive.

Strings are defined between quotation marks, and have the special property that they preserve spaces. Thus whenever you need to use capital letters, e.g. in a file name, or when you need to preserve spaces, e.g. when defining a vector or matrix, you need to use strings. Here is an example where the magnetic field vector of the Zeeman interaction is defined using a string:

```
Interaction Zeeman
{
  type = Zeeman;
  field = "0 0 1e-5";
  spins = electron1, electron2;
}
```

One should in general be very careful when using strings due to the case sensitivity they imply. Especially when writing names; the names of objects or object groups can become case sensitive by using a string:

```
Interaction "Zeeman"  // The name is now case sensitive - only strings can refer to it!
{ /* contents... */ }
```

2.1.4 Input file example

As an example we will setup a calculation of the singlet yield of a radical pair for different external magnetic fields. Let us first create a SpinSystem with the two unpaired electronic spins of the radical pair:

```
SpinSystem RPSystem
{
  Spin electron1
  {
    spin = 1/2;
    type = electron;
    tensor = isotropic(2);
  }

  Spin electron2
  {
    spin = 1/2;
    type = electron;
    tensor = isotropic(2);
  }

  // Other objects will be added here
}
```

This defines two spin-1/2 particles and sets the Landé g-factor to 2 (i.e. the isotropic g-tensors). Let us assume that each radical has a single magnetic nucleus, so we add two more spins to the SpinSystem:

```
  Spin nucleus1
  {
    spin = 1/2; // This is a spin-1/2 like a proton
    type = nucleus;
    tensor = isotropic("1.0");
  }

  Spin nucleus2
  {
    spin = 1; // This is a spin-1 nucleus like nitrogen
    type = nucleus;
    tensor = anisotropic("0.5 0.5 0");
  }
```

We will use the tensors of the nuclei as hyperfine tensors, and we want the specified tensors to be in mT units. Since the units by default are T, we can set the "prefactor" parameter of the hyperfine interaction objects to 10^{-3}. Thus the hyperfine interaction objects are:

```
  Interaction Hyperfine1
  {
    type = Hyperfine;
    group1 = electron1;
    group2 = nucleus1;
    prefactor = 1e-3;
  }

  Interaction Hyperfine2
  {
    type = Hyperfine;
    group1 = electron2;
    group2 = nucleus2;
    prefactor = 1e-3;
  }
```

Next we will add the external magnetic field as a Zeeman interaction, but only with the electrons; the interaction between the magnetic field and the nuclei is so small that it will be neglected here[3]:

```
  Interaction Zeeman
  {
    type = Zeeman;
    field = "0 0 5e-5";
    spins = electron1, electron2;
  }
```

[3]The difference is in the gyromagnetic ratio of the nuclei. If you want to add a Zeeman interaction with the nuclei you should do it as a separate Interaction object with "CommonPrefactor" set to false and by setting the prefactor of the Interaction object to the gyromagnetic ratio.

Next, the reaction processes and initial state of the radical pair should be specified, but in order to this you first need to define some State objects (also within the SpinSystem). In these calculations we are only concerned with the spin state of the two unpaired electrons, so we can just ignore the nuclear spins when defining the State objects. We will define a set of states that makes up a complete basis of the Hilbert space spanned by the two electronic spins:

$$|S\rangle = \frac{1}{\sqrt{2}}(\,|\alpha\beta\rangle - |\beta\alpha\rangle), \qquad |T_+\rangle = |\alpha\alpha\rangle,$$
$$|T_0\rangle = \frac{1}{\sqrt{2}}(\,|\alpha\beta\rangle + |\beta\alpha\rangle), \qquad |T_-\rangle = |\beta\beta\rangle,$$

where α and β are the two spin states of an electronic spin. In MolSpin these states are defined as:

```
State Singlet
{
  spins(electron1,electron2) = |1/2,-1/2> - |-1/2,1/2>;
}

State T0
{
  spins(electron1,electron2) = |1/2,-1/2> + |-1/2,1/2>;
}

State Tp
{
  spin(electron1) = |1/2>;
  spin(electron2) = |1/2>;
}

State Tm
{
  spin(electron1) = |-1/2>;
  spin(electron2) = |-1/2>;
}

State Identity
{
}
```

The normalization constants of $\frac{1}{\sqrt{2}}$ are not necessary as MolSpin will normalize the states for you. Note that we have defined an additional state object without any contents, the *Identity* state, which will serve a special purpose: now that we are ready to define a decay process for the radical pair, the Identity state - which does not specify any particular state for any spin - can be used to specify spin-independent processes. Let us therefore just define a single, spin-independent decay process with a rate constant of 1 μs^{-1}:

```
Transition spinindependent_decay
{
  rate = 1e-3;         // The rate constant
  source = Identity;   // Specify the state here
}
```

Note that if you wanted to have triplet decay from your radical pair, you would need to create three Transition objects: one for each of the states $|T_0\rangle$, $|T_+\rangle$ and $|T_-\rangle$ defined above; it is not possible to define a single generic triplet state. Now we only need to specify the initial state, which is done using a special *Properties* object:

```
Properties properties
{
  initialstate = Singlet;
}
```

Currently no other properties can be specified for a spin system. If you instead want your system to be spawned not in a particular state like $|T_0\rangle$, $|T_+\rangle$ or $|T_-\rangle$, but a mixed ensemble containing an equal amount of all three of these, you can specify a comma-separated list of states to be represented initially:

```
Properties properties
{
  initialstate = T0, Tp, Tm;  // Mixed triplet ensemble
}
```

Note that some calculation methods may not describe ensembles, but the evolution of a specific state, in which case the above initial state specification will create a linear combination of $|T_0\rangle$, $|T_+\rangle$ or $|T_-\rangle$ as the initial state (i.e. just a single state!). No such calculation methods are implemented at the moment, but it may be the case for some custom methods created by other users.

That completes our description of the SpinSystem - the physical system is now described! Next we will create the so-called *Run section* of the input file which is just a collection of *Tasks*, i.e. the actual calculations to be carried out. We only want to perform a simple quantum yield calculation, so we will use the following Run section:

```
Run
{
  Task CalculateQuantumYield
  {
    type = RP-SymmetricUncoupled;
    logfile = "logfile.txt";  // The log will be written to this file
    datafile = "result.dat";  // Data will be written to this file
  }
}
```

The Run section is not assigned a name, and we define a single task of the type *RP-SymmetricUncoupled* which is a very special task type that can only be used under special circumstances - but which is very fast. It is recommended to read more about the main task types: **StaticSS**, **DynamicHS-TimeEvolution**, **StaticHS-SymmetricDecay**, and **RP-SymmetricUncoupled**. All of these can calculate quantum yields, while **StaticSS-TimeEvolution** and **DynamicHS-TimeEvolution** are the main task types for calculating the time evolution of a spin system. If you have any time-dependent interactions you should normally go with DynamicHS-TimeEvolution, but this task type is very slow.

Now we just need to specify the settings for the calculation, in particular we need to specify Actions, Outputs and a Settings object, and these are specified in the Settings object group:

```
Settings
{
  Settings general
  {
    steps = 500;
  }

  Action scan
  {
    value = 0.001;
    direction = "0 0 1";
    type = addvector;
    vector = RPSystem.Zeeman.field;
  }

  Output fieldstrength
  {
    type = length;
    vector = RPSystem.Zeeman.field;
  }
}
```

We specify that we want to perform 500 steps, that is run all the tasks in the run section 500 times. Since we do not just want to repeat the exact same calculation 500 times, we also define an Action, which adds a vector along the *z*-axis with a length of 1 mT to the external magnetic field between every step (i.e. the action is executed every time all of the tasks in the Run section has finished). Finally we also create an Output object, telling MolSpin that we would like to see the current field strength every time results are written. Note that use of such Output objects are not mandatory for the task classes, so not every task type may make use of it - but most task types will.

You are now ready to run MolSpin using your input file:

```
molspin my_input.msd
```

The **.msd** file extension is recommended for MolSpin input files. You can read more about running MolSpin in the next section.

2.2 Running MolSpin

MolSpin is run from the commandline using the following syntax:

```
<path to molspin executable> <parameters> <path to input file>
```

If the folder containing the MolSpin executable file is in your PATH environment variable you can simply write:

```
molspin <parameters> <path to input file>
```

If your input file is called “my_input.msd” (the **.msd** extension is recommended for MolSpin input files, but not required), then you would write:

```
molspin <parameters> my_input.msd
```

This assumes that you are currently in the folder containing the input file – otherwise you would need to specify the path to the file. Note also that you can specify some commandline parameters, and that these *must be given before the input file.*

2.2.1 Commandline parameters

The available commandline parameters are listed below:

Parameter	Example	Description
-a --append	molspin -a myfile.msd	File output is appended to existing files (instead of overwriting).
-c --checkpoint	molspin -c mytask myfile.msd	Skip all tasks in the runsection until the specified checkpoint is found. This only happens for the first step.
-d --defines	molspin -d myfile.msd molspin -d def 5 myfile.msd	Show all defined directives and their values, and included files. Can also be used to create defines on the commandline by specifying name and value to define (beware of case sensitivity).
-h --help	molspin -h	Shows help message if no input file is specified.
-n --steps	molspin -n 5 myfile.msd	Specify how many steps should be calculated before you are notified.
-o --objects	molspin -o myfile.msd	Show the objects that were read from the input files.
-os --objects-states	molspin -os myfile.msd	Similar to --objects, but with extra State object information.
-p --threads	molspin -p 24 myfile.msd	Specify the number of threads/processor cores to use. Works only if you are using OpenBLAS.
-r --first-step	molspin -r 5 myfile.msd	Specify which step to start from (if you don't want to start from step 0).
-l --step-limit	molspin -l 5 myfile.msd	Limits the number of steps to run.
-s --silent	molspin -s myfile.msd	Minimize the text output.
-t --action-targets	molspin -t myfile.msd	Prints all the ActionTargets that can be used.
-z --no-calc	molspin -z myfile.msd	Skip calculations.

Note that you can restart a stopped calculation using:

```
molspin -r <step number> -c <task name> -a myfile.msd
```

Here you can specify the last unfinished step number, and the task name of the first unfinished task that appears in the run section. If all tasks were completed for the last step number that was run (or if you have only a single task in the run section), and you merely wish to extend the calculation with more steps, you can omit the checkpoint specification. Make sure that you are using the append-option though, as you will otherwise overwrite the existing files!

3 Specifying the spin system

A *SpinSystem* defines a physical system. It is an object group that contains Spins, Interactions, Transitions, States and a Properties object, and it is possible to divide a SpinSystem definition between multiple files using the #include directive. SpinSystems are defined as follows:

```
SpinSystem my_system
{
  // Define objects (Spins, Interactions, etc.) here...
}
```

It is possible to define multiple spin systems, but note that not all task types supports interactions between spin systems.

3.1 Spins

Spins are defined by the keyword *spin* followed by a name:

```
Spin Nitrogen_nucleus
{
  spin = 1; // 14N is a spin-1 particle
  tensor = isotropic(1) + anisotropic("0 0 -2.0");
  type = nucleus;
}
Spin electron
{
  spin = 1/2;
  tensor = isotropic(2);  // The anomalous magnetic moment of the electron (without QED)
  type = electron;
}
```

The following parameters can be defined for a spin:

Parameter	**Values**	**Default**	**Description**
Spin	<half-integer>	1/2	The spin quantum number in units of $\hbar$.
Tensor	<tensor>	isotropic(2)	The g-tensor.
Type	*electron*, *nucleus*, *unspecified*	*unspecified*	What type of spin is it (i.e. electron, atomic nucleus or something else?).
QuantizationAxis1	<vector>	"1 0 0"	First spin quantization axis.
QuantizationAxis2	<vector>	"0 1 0"	Second spin quantization axis.
QuantizationAxis3	<vector>	"0 0 1"	Third spin quantization axis.

Note that the spin quantum number is given as a "half-integer": You must specify either an integer or an integer with the suffix "/2". The allowed values of the spin quantum number are therefore: 1/2, 1, 3/2, 2, 5/2, 3, and so on. Note that you can also write e.g. 2/2 instead of 1, but decimal numbers such as 0.5 are not allowed.

The spin quantization axes determine the spin operators in the lab frame, i.e. $\mathbf{S}'_x = a_1\mathbf{S}_x + a_2\mathbf{S}_y + a_3\mathbf{S}_z$ where a_1, a_2 and a_3 are the components of the first quantization axis. Note that the lab frame spin operators are, furthermore, affected by the tensor as well, and in principle the quantization axis specification is redundant as the same results could be obtained using a rotation matrix for the tensor - but sometimes it is just easier to set the quantization axes than calculating the needed transformation matrix.

3.2 Interactions

All interactions come in one of the following two forms:

$$\mathbf{H}_{\mathrm{SS}}(i) = \mu_B\,\alpha\,\mathbf{g}_i\mathbf{S}_i\cdot\mathbf{A}\cdot\mathbf{F}, \qquad \mathbf{H}_{\mathrm{DS}}(i,j) = \mu_B\,\alpha\,\mathbf{g}_i\mathbf{S}_i\cdot\mathbf{A}\cdot\mathbf{g}_j\mathbf{S}_j, \tag{1}$$

where i and j are indices referring to two spins. μ_B the Bohr magneton, $\mathbf{S}_k$ are spin operators, and $\mathbf{g}_k$ are the tensors specified in the spin objects. $\mathbf{A}$, $\mathbf{F}$ and α is a tensor, a vector and a constant factor, respectively, specified for the Interaction object. The Hamiltonian $\mathbf{H}_{\mathrm{SS}}$ involves only a single spin (this could be the Zeeman interaction where $\mathbf{F}$ would be a magnetic field), while $\mathbf{H}_{\mathrm{DS}}$ will be referred to as the double-spin Hamiltonian (e.g. hyperfine or exchange interaction).

The following parameters can be specified for an Interaction:

Parameter	**Values**	**Default**	**Description**
Type	*singlespin*, *doublespin*	-	One of the two types of interaction in Eq. (1). Must always be specified.
Tensor	<tensor>	isotropic(1)	The tensor $\mathbf{A}$ used in Eq. (1).
Field	<vector>	zero-vector	The vector $\mathbf{F}$ used in Eq. (1). Note: Only used in single-spin interactions!
Prefactor	<float>	1.0	Optional prefactor for the interaction object. α in Eq. (1).
CommonPrefactor	<bool>	true	If set to false, the Bohr magneton μ_B will not be included in Eq. (1) for the interaction object.
IgnoreTensors	<bool>	false	If set to true, the tensors of the spin objects $\mathbf{g}_i$ and $\mathbf{g}_j$ will not be included in Eq. (1) for the interaction object.
FieldType TimeDependence	*LinearPolarization*, *CircularPolarization*, *Static*	*Static*	Allows the Field vector of the interaction object to become time-dependent. Note: Not all task types support time-dependent Hamiltonians!
Frequency	<float>	1.0	Frequency of field oscillations. Requires a FieldType other than *Static*.
Phase	<float>	0.0	Initial phase of field oscillations. Requires a FieldType other than *Static*.
Axis	<vector>	"0 0 1"	Axis that the field will rotate about. Requires FieldType set to *CircularPolarization*.
Perpendicular-Oscillations	<bool>	false	If set to true, the rotating field vector will be projected onto the plane perpendicular to the rotation axis. Requires FieldType set to *CircularPolarization*.

Note that there are some synonyms for the *Type* parameter: you can write *onespin* or *Zeeman* instead of *singlespin*, and you can use *twospin*, *hyperfine*, *dipole* or *exchange* instead of *doublespin*.

3.3 Transitions

A transition object describes a process that can change the total number of spin systems, i.e. it represents a source or a sink. Transition objects are commonly used to model decay or recombination reactions (sinks) or processes that replenishes the spin system. For transitions of the sink type it is also possible to specify a target spin system together with a target state, meaning that the transition/process transforms the spin system into another spin system - an example of such a process is an electron transfer from a radical, that generates another radical (since both radicals are spin systems, but with different molecular environment). Note that target spin systems are not taken into account by most task types; see the individual task types for more information.

Transition objects have the following parameters:

Keyword	**Values**	**Description**
Type	*source*, *sink*	Determines whether the process is a sink (default) or a source.
Source State SourceState	<State>	The state in the SpinSystem which is diminished or augmented.
Target TargetSystem	<SpinSystem>	The new SpinSystem that the transition leads to. Requires specification of a target state, and transition type must be *sink*.
TargetState	<State>	A state in the target SpinSystem.
ReactionOperators ReactionOperatorType	*Haberkorn*, *Lindblad*	Requests a specific type of reaction operator for this transition (Haberkorn is default).
Rate	<float>	The rate constant for the transition in ns^{-1}.
Lifetime	<float>	The inverse of the rate constant for the transition in ns. Only used if rate is not specified (either directly or in a trajectory).

Note that the reaction operator type is not taken into account by all task classes.

3.4 Operators

Operators are may be used for a variety of purposes by custom task classes, but their original intended purpose is to describe relaxation operators. Operator objects are a recent addition to the SpinAPI, and their uses in the implemented task classes is limited.

Operator objects have the following parameters:

Keyword	**Values**	**Description**
Type	*RelaxationLindbla* *relaxationdephasin* *unspecified*	The type of operator.
Rate1	<float>	A value that may be associated with the operator.
Rate2	<float>	A value that may be associated with the operator.
Rate3	<float>	A value that may be associated with the operator.
Rates	<float>	Sets all three rates to this value.
Spins SpinList	<Spins>	Spin objects that should be affected by the operator.

Note that the SpinAPI::SpinSpace class only supports the Operator type *RelaxationLindblad*, and only for Liouville-space task classes such as *StaticSS*. The produced relaxation is defined by:

$$\mathbf{R}(\rho) = \sum_i \left[k_x \left(\mathbf{S}_{ix}\rho\mathbf{S}_{ix} - \frac{1}{2}(\mathbf{S}_{ix}\rho + \rho\mathbf{S}_{ix}) \right) + k_y \left(\mathbf{S}_{iy}\rho\mathbf{S}_{iy} - \frac{1}{2}(\mathbf{S}_{iy}\rho + \rho\mathbf{S}_{iy}) \right) + k_z \left(\mathbf{S}_{iz}\rho\mathbf{S}_{iz} - \frac{1}{2}(\mathbf{S}_{iz}\rho + \rho\mathbf{S}_{iz}) \right) \right],$$

where the sum is over all spins in the specified *spinlist*, and k_x, k_y and k_z are the three rates.

3.5 States

State objects define the quantum state of a spin system, although they often describe only the quantum state of a subsystem, for example the singlet state of a radical pair. There are only two types of input in a State object:

Keyword	Values	Description
spin(<spin>)	<state ket>	Specify the the state of a single spin.
spins(<spinlist>)	<state ket>	Specify the the state of entangled spins.

The first keyword, *spin*, is used to specify the spin state of a single Spin object inside the spin system. The spin state should be written as a state ket, i.e. |1/2> or |-1/2> for a spin-1/2 particle, or |+1>, |0> or |-1> for a spin-1 particle. Note that the name of the Spin object is given between (and) as part of the keyword, and you can thus set the state for each spin individually as in the following example:

```
// Define some spin objects
Spin s1 { spin = 1/2; }
Spin s2 { spin = 3/2; }

// Define a quantum state
State MyState
{
  spin(s1) = |1/2>;
  spin(s2) = |-3/2>;
}
```

You do not need to specify a state for every spin in the spin system. Whenever the state object is used to generate a projection operator onto a quantum state (as is the case in almost all task classes), the resulting state projection operator will be an identity operator in the subspace spanned by the spins which have not been specified in the State object. As a special result of this, the projection operator corresponding to an empty State object is just the Identity operator for the spin system, and using such a State object as the source state in a Transition will lead to a spin-independent transition:

```
// Empty State object
State IdentityOperator { }
```

All task types that rely on a density operator formalism, for example, only use State objects to create projection operators (the initial ensemble described by a density operator can be written as $\rho(0) = \sum_i \mathbf{P}_i / \operatorname{Tr}(\sum_i \mathbf{P}_i)$ where $\mathbf{P}_i$ are projection operators). You need to be more careful with task types that use state vectors (e.g. when solving the Schrödinger equation) since state vectors need to represent a specific state - in such cases it is not possible to use a mixed ensemble of states. Such task types are not currently implemented in MolSpin though.

Specifying the state of each spin individually is not always enough, since spins can be entangled. Examples are the singlet and T_0 states of a radical pair. Such entanglements can be specified using the keyword *spins* by providing a comma-separated list of Spin object names between (and) as part of the keyword. The value is again written as a state ket, where the magnetic quantum number of each spin is given as a comma-sparated list. Also, in order to properly specify entanglements, a linear combination of state kets can be specified:

```
// Define some spin objects
Spin s1 { spin = 1/2; }
Spin s2 { spin = 3/2; }
Spin s3 { spin = 1/2; }

// Define a quantum state
State MyEntangledState
{
  spins(s1,s2) = 2|+1/2,-3/2> - 3i|+1/2,-1/2> + 0.1 |-1/2,3/2>;
  spin(s3) = |1/2>;
}
```

Note that MolSpin does not expect the specified state to be normalized; it is the responsibility of the task classes to normalize any state vectors or spin state ensembles.

3.6 Properties

A spin system can have a single Properties object which defines some general properties for the system. The following properties are available:

Keyword	**Values**	**Description**
InitialState	<State list>	A comma-separated list of states describing the initial state of the spin system.

If multiple initial states are specified, note that the exact meaning of having multiple initial states depends on the used task types (especially if custom task types are used). All of the standard task types described in this manual treats a spin system as an ensemble which initially contains an even mixture of all the specified states, unless otherwise specified in the task class description.

3.7 Specifying Tensors

Tensor objects can be specified for Interaction and Spin objects, and play the role of hyperfine and g-tensors. Since tensors are complex entities, a special syntax must be used to specify them, and this syntax consists of a combination of the following keywords:

Keyword	**Values**	**Description**
Isotropic	<float>	The isotropic value of the tensor.
Anisotropic	<vector>	The three components of the anisotropic part.
Matrix	<matrix>	A full matrix representation of the tensor. The matrix is expected to be symmetric.
axis1 axis2 axis3	<vector>	The principal axes of the tensor.
ChangeBasis	<matrix>	A similarity transformation will be applied to the lab frame matrix representation of the tensor, i.e. $\mathbf{A}' = \mathbf{S}^\dagger \mathbf{A} \mathbf{S}$ where $\mathbf{A}$ is the current lab frame representation, $\mathbf{A}'$ is the lab frame representation after the transformation, and $\mathbf{S}$ is the transformation matrix specified by this keyword.
Trajectory	<filename>	Load a trajectory for the tensor, and use it to set the tensor parameters.

Every keyword is followed by parentheses containing the parameters, and multiple keywords are added by a plus sign. For example:

```
tensor = isotropic("1.0") + anisotropic("1e-4 1e-4 1e-3") + trajectory("motions.mst");
tensor = matrix("1.0 0.0 0.0;0.0 1.0 0.0;0.0 0.0 1.0");
```

The trajectory in the first example could for example contain columns with time and the three principal axes, e.g. describing rotational diffusion of a radical while assuming constant principal values (isotropic and anisotropic values) of the tensor. Note that the matrix in the second example above is an identity matrix, so the same definition could be made using **isotropic("1.0")** instead. The examples above also demonstrate **how vectors and matrices are defined**: vectors consists of three numbers separated by **spaces**, and must be written as **strings**[4], and matrices are defined in a similar way, with **semicolon** to separate rows in the matrix.

Important note on tensor definitions: The order in which components of a tensor are added together matters! The keywords for a tensor are processed in the order in which they appear; in the first example above, the isotropic part is first set, then the anisotropic part, and then finally a trajectory is assigned. There is no conflict in any of these examples, but consider instead the following two cases which yield very different results:

```
tensor = axis1("1 1 0") + axis2("1 -1 0") + matrix("1 0.1 -0.3;0.1 1 0.7;-0.3 0.7 -2");
tensor = matrix("1 0.1 -0.3;0.1 1 0.7;-0.3 0.7 -2") + axis1("1 1 0") + axis2("1 -1 0");
```

The **matrix** keyword diagonalizes the given matrix and sets the three principal axes to the eigenvectors of the matrix, regardless of what the axes were assigned to previously. Hence the first line above is *wrong*: the axis specifications are **overwritten** by the *matrix* keyword! Note also that any transformation performed by the *changebasis* keyword will be overwritten by the axis keywords. Thus **it is recommended** that you *use the* axis *keywords after the* matrix *keyword (as in the second line above), and the* changebasis *keyword after the* axis *keywords.*

[4]In order to preserve the spaces that would otherwise be ignored by the input file parser!

3.8 Specifying Trajectories

Trajectories is a special feature in MolSpin that allows you to specify complex time-dependences of parameters. For some task types they may also be used to provide sets of parameters, without specifying any time - such uncommon uses will be mentioned explitcly in the documentation for the individual task types. A trajectories are specified in separate files which contain only a trajectory, and it is recommended to use the **.mst** file extension to easily identify trajectory files (**M**ol**S**pin **T**rajectory).

A trajectory file can be perceived as a table consisting of some rows and columns. The first line of the trajectory file defines the column headers, and each following line defines a row with data. Each column is separated by one or more space or tab characters; this holds for both the header row and all the data rows.

Below is an example of a trajectory file for an Interaction object:

```
time prefactor field.x field.y field.z
100 1 0 0 5e-4
300 1 5e-4 0 0
400 1 0 5e-4 0
600 5 0 0 5e-4
800 0.1 0 0 0.001
```

The **time** column has a special meaning and is required for task types that includes time-dependent parameters: when the parameters in the trajectory are requested at a specific time (here the prefactor and field of an Interaction object), *the first entry in the trajectory* is used for all times *smaller* than the first specified time. When the requested time is *larger* than any entries in the trajectory it is instead *the final entry in the trajectory* that is used. For all requested times between that of the first and the final entry in the trajectory, the first entry with a time larger than the requested time is found, and a **linear interpolation** between this and the previous row in the trajectory is used for all the parameters. It is currently not possible to change the interpolation type, but since there is no limit on the number of rows in the trajectory you still have fine-grained control[5]. Note that MolSpin expects the rows to be **ordered by time**.

3.8.1 Interaction trajectories

Interaction objects recognize the following column headers:

time, **prefactor**, **field.x**, **field.y**, **field.z**.

Note that if any of the field components (field.x/y/z) are used, it is required that the remaining components are specified as well. Otherwise the specified field components are ignored (i.e. you must either specify both field.x, field.y and field.z, or none of the three).

Trajectories for the tensor of an interaction is specified separately by adding the *trajectory* keyword to the tensor specification, and can be a separate file. You can use the same trajectory file for both the Interaction object and its tensor since Interaction and Tensor objects use different column header names (except time) and each object type will just ignore the columns that it does not use. Note that this approach still means that the trajectory file will be loaded into memory twice, and that there is a certain memory overhead (waste) with this approach - for very long trajectories it is therefore recommended to split the Interaction trajectory (prefactor and field.x/y/z) and its Tensor trajectory into two separate files.

[5]Memory limitations still apply; every trajectory is stored in the memory while MolSpin is running.

3.8.2 Tensor trajectories

Tensor objects recognize the following column headers:

> **time**, **isotropic**, **anisotropic.x**, **anisotropic.y**, **anisotropic.z**, **axis1.x**, **axis1.y**, **axis1.z**, **axis2.x**, **axis2.y**, **axis2.z**, **axis3.x**, **axis3.y**, **axis3.z**

Note that if *one* component of a vector is specified, *all* components of the same vector must be specified (as with the field.x/y/z for Interaction objects). Thus if you want to specify the anisotropic part you must specify both anisotropic.x, anisotropic.y and anisotropic.z. Similar with e.g. axis2.x/y/z. But you do not need to specify all axes if you specify one of them; you can for example specify axis1 and axis3 without specifying axis2, as long as you just include both the x, y and z components of both axis1 and axis3.

Here is an example of a tensor trajectory:

```
time isotropic anisotropic.x  anisotropic.y  anisotropic.z
100 0 0.0 0.0 0.0
200 2 0.2 0.2 0.0
400 1 0.0 1.2 0.0
500 0 0.8 0.0 0.0
800 1 0.0 0.0 2.0
```

3.8.3 Transition trajectories

Transition objects can have trajectory that specifies the time-evolution of the rate constant, or alternatively the time-evolution can be specified for the lifetime instead of the rate constant (the lifetime is the inverse of the rate constant).

Here is an example of a Transition trajectory:

```
time rate lifetime
100 1e-2 100
200 1e-4 10000
400 0.0 1e+10
500 1e-3 1000
600 2e-3 500
```

Note: You should specify either a rate column or a lifetime column, since only one of them can be used (the lifetime also provides a rate). If both columns are present as in the example above, only the column with the "rate" header will be used.

3.8.4 Spin trajectories

For Spin objects their tensors can be assigned a trajectory, but the Spin object itself may also be assigned a trajectory with the spin quantization axes.

> **time**, **axis1.x**, **axis1.y**, **axis1.z**, **axis2.x**, **axis2.y**, **axis2.z**, **axis3.x**, **axis3.y**, **axis3.z**

Note that the same trajectory file can be assigned to Spin objects and their tensor, since the Spin object will just ignore the additional data columns used by the tensor.

3.9 ActionTargets: ActionVectors and ActionScalars

Many parameters can be changed between the executing of calculation steps using *Actions* (see section 4.2), which provides a great deal of flexibility. These parameters that can be affected by Actions are collectively known as ActionTargets, and may be divided into ActionVectors and ActionScalars. An example of an ActionVector is the field vector of a singlespin Interaction, whereas the rate constant for a Transition object, for example, is an ActionScalar. In addition to their use in Actions, ActionTargets can also be used by Output objects (see section 4.3).

The name of an ActionTarget will be of the form "<SpinSystem name>.<object name>.<parameter>", and you can use the "-t" commandline option to see a list of available ActionTargets for the spin system(s) you have defined:

```
molspin -t -z myfile.msd
```

Here the "-z" option is also used to prevent running any calculations, assuming you only wanted to see a list of ActionTargets.

Note: Make sure that all your objects within a spin system have unique names, and that your spin systems have unique names, such that each ActionTarget can be uniquely identified.

4 Specifying the settings object group

The Settings object group allows you to define *Actions* and *Output requests*, as well as some general options for your calculations.

4.1 The Settings object

The Settings object should contain a Settings object, which is used to specify general options for the calculations. The following settings are available:

Keyword	Values	Description
Steps	<integer>	The number of calculation steps to perform, i.e. number of times the Run section is being executed. Default: 1.
Time	<float>	Sets the time for any time-dependent interactions or trajectories. Only useful for task classes that do not support time-dependent interactions (in order to "freeze" time-dependent interactions at specific values for the calculations). Default: 0.
TrajectoryStep	<integer>	Sets the trajectory step for any parameter with a trajectory. For trajectories including a time column this is only useful for task classes that do not support time-dependent interactions. Default: 0.
TrajectoryStep-BeforeTime	<bool>	If true, the trajectory step will be set before the time, which means that trajectories with a time column will use time instead of trajectory step. Default: true, except when *TrajectoryStep* and not *Time* is specified.
Notifications	*quiet* *sparse* *normal* *details*	Specifies the notification level. Default: Normal.
IgnoreWarnings Nowarn	<bool>	Ignore warnings. Default: False.
IgnoreErrors Noerr	<bool>	Ignore error messages. Default: False.

Note that *TrajectoryStepBeforeTime* should be written without "-". Note also that some task classes do not distinguish between notification levels for log output.

4.2 Actions

The use of *actions* is a central feature of the MolSpin application, as they provide a very general framework for setting up calculations. When you set up your SpinSystems you are able to perform a single calculation, but if you want to see the effect of changing a parameter on e.g. the quantum yields, multiple calculations are needed. Hence you can specify the number of calculations you want to be performed as the *steps* parameter in the settings object. But without *Actions*, all of those calculations will be identical – it is the role of an Action to change the parameters between these individual calculations.

The following types of actions are available, with the Action-specific parameters shown together with each Action Type:

Action Type	Parameter	Description
RotateVector		Rotates an ActionVector about a specified axis. Rotation angle is given as the action value in degrees.
	Vector ActionVector	The ActionVector to act on.
	Axis	A vector specifying the axis to rotate the ActionVector about.
ScaleVector		Scales an ActionVector by multiplication of the action value.
	Vector ActionVector	The ActionVector to act on.
AddVector		Adds a constant vector to the ActionVector.
	Vector ActionVector	The ActionVector to act on.
	Direction	A vector specifying the direction of the vector to be added to the ActionVector. The length of the vector is set to the action value.
MultiplyScalar ScaleScalar		Multiplies an ActionScalar by the constant action value.
	Scalar ActionScalar	The ActionScalar to act on.
AddScalar		Adds the constant action value to an ActionScalar.
	Scalar ActionScalar	The ActionScalar to act on.

In addition to the Action-type-specific parameters, the following list of parameters are available for all Actions:

Parameter	Values	Description
Type	<type>	A type of Action object from the table above.
Value	<float>	This is the "action value" refered to by the specific actions, and its function depends on the action type (see the action type table above).
Period	<integer>	Specifies how frequently the action should be performed. A period of 1 (default) means that the action will be performed after every step, a period of 5 means it will be performed after every 5th step.
First	<integer>	The first step at which the action should be performed (default 1). After this step, the action will be performed with a frequency specified by the action period.
Last	<integer>	The last step at which the action may be performed. After reaching this step, the action will not be performed anymore.

4.3 Output requests

In addition to the standard output from the various task classes, such as quantum yields, you can request additional output using *Output* objects. Note that Output objects only requests output, but it is up to the individual task classes to decide whether when and where this requested output is printed, and task classes may choose to ignore it (but this is not the case for any of the currently implemented task classes).

Output requests are used to obtain information about ActionTargets: either by printing the actual ActionScalar value or ActionVector components, or by requesting information such as the length of an ActionVector. All the possible Output request types are listed in the table below:

Output Type	**Parameter**	**Description**
Angle VectorAngle		Outputs the angle (degrees) between two vectors.
	Vector	A pointer to the ActionVector used to calculate the angle.
	Reference	A constant vector used to calculate the angle.
Length VectorLength		Outputs the lengths of a vector.
	Vector	A pointer to the ActionVector whose length should be calculated.
Components XYZ Vector VectorXYZ		Outputs the Cartesian components of a vector.
	Vector	A pointer to the ActionVector whose components to write to the output file.
Dot VectorDot Projection		Calculates the projection (dot product) of the vector onto a reference vector.
	Vector	A pointer to the ActionVector used to calculate the projection.
	Reference	A constant vector the ActionVector is projected onto.
Scalar		Outputs a scalar
	Scalar	A pointer to the ActionScalar to write to the output file.

In addition, Output objects have the following properties that can be specified:

Keyword	**Values**	**Description**
Type	<type>	A type of output object from the table above.
Prefactor	<float>	Any value will be multiplied by this number before being printed.

5 The Run section and task classes

The Run section defines what happens at every calculation step: each task specified in the Run section are executed sequentially within a calculation step, and this execution is repeated for the number of calculation steps defined in a Settings objects (i.e. in the Settings object group, not the Run section which only defines what happens every step). The tasks in the Run section are executed in the order that they appear in the input file.

5.1 Overview of task classes

There is a wide variety of task classes available, as summarized in the table below. In general their names refer to Hilbert Space (HS) or Superoperator Space (SS) methods, where superoperator space (Liouville space) methods in general requires much more memory (i.e. scales much worse with number of spins). The prefixes *Static*, *Dynamic* and *Periodic* tells whether time-dependent interactions are supported. Note that some methods are specialized to specific spin systems; the RP-SymmetricUncoupled method is designed for radical pairs only.

Task class	**Type**	**TD**	**RT**	**MS**	**Memory**	**Speed**
DynamicHS-TimeEvolution	TE/QY	Yes	All	No	Medium	Very slow
PeriodicHS-TimeEvolution	TE/QY	Per.	All	No	Medium	Slow
MultiDynamicHS-TimeEvolution	TE	Yes	All	Yes	Medium	Very slow
StaticSS-TimeEvolution	TE	No	All	No	High	Medium
PeriodicSS-TimeEvolution	TE	Per.	All	No	High	Medium
MultiStaticSS-TimeEvolution	TE	No	All	Yes	High	Medium
StaticSS	QY	No	All	No	High	Medium
StaticHS-SymmetricDecay	QY	No	Indep.	No	Medium	Fast
RP-SymmetricUncoupled	QY	No	Indep.	No	Low	Very fast
Gamma-Compute	QY	Per.	Indep.	No	Medium	Fast
Eigenvalues	EL/RES	Yes	–	No	Medium	Fast

Type: Describes what can be calculated, which is either the time-evolution (TE), quantum yields (QY), energy levels (EL) or resonance frequencies (RES). **TD:** Whether time-dependences are taken into account, where *Per.* means only periodic time-dependences. **RT:** Supported reaction types, where *Indep.* means that only a spin-independent reaction is supported. **MS:** Supports use of multiple spin systems. Most task types without multi-system support can still handle each spin system individually, but ignores transitions between spin systems. **Memory and speed:** The memory and calculation time requirements of a task class, which for most task types scales exponentially with system size.

5.2 General parameters

There are a few parameters that are available to all task classes; these are shown in the table below.

Parameter	Values	Description
Logfile Log Output	<filename>	Location of logfile. If no logfile is specified, log output will be written to stdout.
Datafile Data	<filename>	Location of main data file. If no data file is specified, main data output will be written to stdout.
AppendLog	<bool>	If set to true, log output will be appended to the specified logfile instead of overwriting an existing file. Not enabled by default.
AppendData	<bool>	If set to true, data output will be appended to the specified main data file instead of overwriting an existing file. Not enabled by default.
Append	<bool>	Setting this option overrules both AppendLog and AppendData. Note that the command-line append option also overrules this option.
Notifications	*quiet* *sparse* *normal* *details*	Specifies the notification level. Default: Normal.
IgnoreWarnings Nowarn	<bool>	Ignore warnings. Default: False.
IgnoreErrors Noerr	<bool>	Ignore error messages. Default: False.

5.3 Task class: DynamicHS-TimeEvolution

Brief description

Brute-force numerical integration to obtain the time-evolution of the spin system ensemble. Can perform time integration of these results in order to output quantum yields instead.

Limitations, approximations and comments

- This is the most general method to compute time-evolution or quantum yields for a single SpinSystem; it can handle all types of input.
- MultiDynamicHS-TimeEvolution is the equivalent method to use when you need multiple SpinSystems.
- Make sure to use a small enough timestep to avoid significant numerical errors.
- When calculating quantum yields, make sure the totaltime parameter is long enough to let the SpinSystem decay completely.
- This is a very slow method; use it only when faster methods do not suffice.

Parameters

Parameter	Values	Default	Description
TimeStep	<float>	0.01	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
CalculateYields	<bool>	false	Whether to output time-evolution (false) or quantum yields (true).
TransitionYields	<bool>	false	Whether to calculate quantum yields per transition (true) or per State defined in the SpinSystem (false). Quantum yield calculations only.
OutputStride	<integer>	1	How often output should be written (time-evolution output only), i.e. write output every nth numerical integration timestep, where the parameter n is specified here.

Technical details

Solves the Liouville-von Neumann equation for the density operator ρ:

$$\frac{d\rho}{dt} = -\frac{i}{\hbar}[\mathbf{H}, \rho] + \mathcal{K}(\rho), \tag{2}$$

where $\mathbf{H}$ is the (time-dependent) Hamiltonian, and $\mathcal{K}$ is the superoperator representing reactions (either Haberkorn or Lindblad form).

The time-evolution output is the expectation value of state projection operators:

$$p_i(t) = \mathrm{Tr}(\mathbf{P}_i \rho(t)), \tag{3}$$

where $\mathbf{P}_i$ is the expectation value of state i. This is calculated for each State object defined in the SpinSystem (i.e. the index i refers to such a State object).

If quantum yields are requested instead, there are two posibilities; by default the yields are just calculated for each State object defined in the SpinSystem:

$$\Phi_i = \int_0^{\mathrm{TotalTime}} \mathrm{Tr}(\mathbf{P}_i \rho(t))\, dt\,. \tag{4}$$

Note that the upper integration limit is *TotalTime* rather than ∞, and that no rate constant is involved here; thus this is not a "proper yield". The other possibility is when the *TransitionYields* parameter is true:

$$\Phi_j = k_j \int_0^{\mathrm{TotalTime}} \mathrm{Tr}(\mathbf{P}_j \rho(t))\, dt\,. \tag{5}$$

Here the index j refers to the *SourceState* parameter of a transition object (which is also a State object, and hence $\mathbf{P}_j$ is the projection onto that state) and k_j the rate of the transition. This is calculated for each transition object in the SpinSystem.

The numerical integration of the Liouville-von Neuman equation is performed using an asynchronous leapfrog algorithm to obtain the time-evolution of the spin system, and the quantum yields are calculated using the trapezoidal rule on the time-evolution data.

5.4 Task class: PeriodicHS-TimeEvolution

Brief description

Same method as DynamicHS-TimeEvolution except that periodicity in the time-dependence of the Hamiltonian is exploited to provide a faster method.

Limitations, approximations and comments

- Use this method if you use one or more time-dependent interactions, where all of these time-dependences are periodic.
- The totaltime parameter should be at least a few periods long in order to get any advantage from this method.
- Requires more memory than DynamicHS-TimeEvolution.
- The method does not work with trajectories.
- Assumes the Hamiltonian to be piecewise constant for very short time intervals as defined by the StepsPerPeriod parameter. The accuracy of the results can generally be improved by increasing this parameter (at the cost of speed and increased memory usage).
- Make sure to use a small enough timestep to avoid significant numerical errors.
- When calculating quantum yields, make sure the totaltime parameter is long enough to let the SpinSystem decay completely.

Parameters

Parameter	**Values**	**Default**	**Description**
TimeStep	<float>	0.01	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
CalculateYields	<bool>	false	Whether to output time-evolution (false) or quantum yields (true).
TransitionYields	<bool>	false	Whether to calculate quantum yields per transition (true) or per State defined in the SpinSystem (false). Quantum yield calculations only.
OutputStride	<integer>	1	How often output should be written (time-evolution output only), i.e. write output every nth numerical integration timestep, where the parameter n is specified here.
Steps StepsPerPeriod	<integer>	50	Splits each period into this number of intervals. The Hamiltonian is assumed to be constant in each of these intervals.

Technical details

Same method as DynamicHS-TimeEvolution except that the Hamiltonian is calculated at different times before the numerical integration procedure is started, and these Hamiltonians are then cached such that they can be used repeatedly during the numerical integration; i.e. this method gains speed at the cost of increased memory usage (calculating Hamiltonians is time consuming). The period is inferred from the interactions in the spin system.

If the StepsPerPeriod parameter is denoted N and the period of the Hamiltonian by T, then the Hamiltonian is calculated at times $nT/N \ \forall n \in \{0, 1, ..., N\}$ and is assumed to be constant for the duration T/N.

5.5 Task class: MultiDynamicHS-TimeEvolution

Brief description

Multi-system version of the DynamicHS-TimeEvolution method, i.e. supports transitions between Spin-Systems.

Limitations, approximations and comments

- This is the most general method to compute time-evolution; it can handle all types of input.
- Use DynamicHS-TimeEvolution instead if you only have a single SpinSystem.
- This method can be used to calculate quantum yields as well: just create an extra spin system to hold the product states and create Transition objects with these product states as target (if no interactions are present in this other system the reaction products will just accumulate and so you get the accumulated quantum yield over time).
- Make sure to use a small enough timestep to avoid significant numerical errors.
- This is a very slow method; use it only when faster methods do not suffice.
- Alternative type name: *DynamicHS-MultiSystem.*

Parameters

Parameter	Values	Default	Description
TimeStep	<float>	0.01	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
OutputStride	<integer>	1	How often output should be written (time-evolution output only), i.e. write output every nth numerical integration timestep, where the parameter n is specified here.

Technical details

To be added...

5.6 Task class: StaticSS-TimeEvolution

Brief description

Superoperator space (Liouville space) method that calculates the time-evolution of spin systems with a time-independent Hamiltonian.

Limitations, approximations and comments

- Superoperator space methods scale much worse with system size than native Hilbert space methods (in terms of both memory and speed).
- Ignores time-dependences: If time-dependences are present, they are evaluated only at time 0.

Parameters

Parameter	Values	Default	Description
TimeStep	<float>	1.0	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
ReactionOperator	*Haberkorn*, *Lindblad*	*Haberkorn*	The default reaction operator type to use for this task. This is ignored for Transition objects where the reaction operator type is specified.

Technical details

Calculates the propagator in superoperator space (Liouville space):

$$\hat{\mathbf{U}} = \exp\left(-\frac{i}{\hbar}\hat{\mathbf{H}}\,\Delta t + \hat{\mathbf{K}}\,\Delta t\right) = \exp(\hat{\mathbf{A}}\,\Delta t), \tag{6}$$

where $\hat{\mathbf{H}}$ is the Hamiltonian superoperator, $\hat{\mathbf{K}}$ is the superoperator representing reactions (either Haberkorn or Lindblad form), and $\hat{\mathbf{A}}$ is the total Liouvillian superoperator. The propagator is then used to calculate the time-evolution:

$$\rho(t + \Delta t) = \hat{\mathbf{U}}\rho(t) \tag{7}$$

The time-evolution output is the expectation value of state projection operators:

$$p_i(t) = \mathrm{Tr}(\mathbf{P}_i\rho(t)), \tag{8}$$

where $\mathbf{P}_i$ is the expectation value of state i. This is calculated for each State object defined in the SpinSystem (i.e. the index i refers to such a State object).

5.7 Task class: PeriodicSS-TimeEvolution

Brief description

Superoperator space (Liouville space) method that calculates the time-evolution of spin systems with a periodic time-dependence of the Hamiltonian. Very similar to StaticSS-TimeEvolution.

Limitations, approximations and comments

- Use this method if you use one or more time-dependent interactions, where all of these time-dependences are periodic.
- The totaltime parameter should be at least a few periods long in order to get any advantage from this method.
- Requires more memory than StaticSS-TimeEvolution, but can handle periodic time-dependences.
- The method does not work with trajectories.
- Assumes the Hamiltonian to be piecewise constant for very short time intervals as defined by the StepsPerPeriod parameter. The accuracy of the results can generally be improved by increasing this parameter (at the cost of speed and increased memory usage).
- Superoperator space methods scale much worse with system size than native Hilbert space methods (in terms of both memory and speed).

Parameters

Parameter	Values	Default	Description
TimeStep	<float>	1.0	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
ReactionOperator	*Haberkorn*, *Lindblad*	*Haberkorn*	The default reaction operator type to use for this task. This is ignored for Transition objects where the reaction operator type is specified.
Steps StepsPerPeriod	<integer>	50	Splits each period into this number of intervals. The Hamiltonian is assumed to be constant in each of these intervals.

Technical details

Same propagator-based method as StaticSS-TimeEvolution, except that multiple propagators are calculated: A full period T of the periodic Hamiltonian is divided into N intervals (given by the StepsPerPeriod parameter), each of length T/N, and a propagator is calculated for each interval:

$$\hat{\mathbf{U}}(n\,T/N) = \exp\left(\left[-\frac{i}{\hbar}\hat{\mathbf{H}}(n\,T/N) + \hat{\mathbf{K}}\right]\Delta t\right) \quad \forall\, n \in \{0, 1, ..., N\}, \tag{9}$$

These propagators are then used repeatedly to propagate the state of the spin system.

Note that N propagators are calculated and stored in memory, in contrast to a single propagator for the StaticSS-TimeEvolution method. This results in a significantly poorer performance than StaticSS-TimeEvolution.

5.8 Task class: MultiStaticSS-TimeEvolution

Brief description

Superoperator space (Liouville space) method. Multi-system version of StaticSS-TimeEvolution, i.e. supports transitions between SpinSystems.

Limitations, approximations and comments

- Use a different method if you only have a single spin system.
- Both source and target state of Transition objects must be completely described (i.e. use state objects where the state of *every* spin is specified). Note that this limitation is not present in the MultiDynamicHS-TimeEvolution method.
- The ReactionOperator parameter only affects the used decay operator, not the creation operator (see technical details).
- This method can be used to calculate quantum yields as well: just create an extra spin system to hold the product states and create Transition objects with these product states as target (if no interactions are present in this other system the reaction products will just accumulate and so you get the accumulated quantum yield over time).
- Superoperator space methods scale much worse with system size than native Hilbert space methods (in terms of both memory and speed).
- Alternative type name: *StaticSS-MultiSystem*.

Parameters

Parameter	Values	Default	Description
TimeStep	<float>	1.0	Numerical integration timestep (in ns).
TotalTime	<float>	10000	Total integration time (in ns).
ReactionOperator	*Haberkorn*, *Lindblad*	*Haberkorn*	The default reaction operator type to use for this task. This is ignored for Transition objects where the reaction operator type is specified.

Technical details

To be added...

5.9 Task class: StaticSS

Brief description

Superoperator space (Liouville space) method that calculates the quantum yield. This is the most general quantum yield calculation method to use when there are no time-dependent interactions or time-dependent rate constants, since time-dependences are not taken into account.

Limitations, approximations and comments

- Superoperator space methods scale much worse with system size than native Hilbert space methods (in terms of both memory and speed).
- Ignores time-dependences: If time-dependences are present, they are evaluated only at time 0.
- Alternative type name: *StaticIVP*.

Parameters

Parameter	Values	Default	Description
TransitionYields	<bool>	false	Whether to calculate quantum yields per transition (true) or per State defined in the SpinSystem (false).
ReactionOperator	*Haberkorn*, *Lindblad*	*Haberkorn*	The default reaction operator type to use for this task. This is ignored for Transition objects where the reaction operator type is specified.

Technical details

Solves the Liouville-von Neumann equation for the density operator ρ, in superoperator space (Liouville space):

$$\frac{d\rho}{dt} = -\frac{i}{\hbar}\hat{\mathbf{H}}\rho + \hat{\mathbf{K}}\rho = \hat{\mathbf{A}}\rho, \tag{10}$$

where $\hat{\mathbf{H}}$ is the Hamiltonian superoperator, $\hat{\mathbf{K}}$ is the superoperator representing reactions (either Haberkorn or Lindblad form), and $\hat{\mathbf{A}}$ is the total Liouvillian superoperator.

There are two posibilities for calculation of the quantum yields; by default the yields are just calculated for each State object defined in the SpinSystem:

$$\Phi_i = \int_0^\infty \mathrm{Tr}(\mathbf{P}_i \rho(t))\, dt = \mathrm{Tr}(\mathbf{P}_i \hat{\mathbf{A}}^{-1} \rho(0))\,. \tag{11}$$

Note that there is no rate constant is involved here; thus this is not a "proper yield". The other possibility is when the *TransitionYields* parameter is true:

$$\Phi_j = k_j \int_0^\infty \mathrm{Tr}(\mathbf{P}_j \rho(t))\, dt = k_j\, \mathrm{Tr}(\mathbf{P}_j \hat{\mathbf{A}}^{-1} \rho(0))\,. \tag{12}$$

Here the index j refers to the *SourceState* parameter of a transition object (which is also a State object, and hence $\mathbf{P}_j$ is the projection onto that state) and k_j the rate of the transition. This is calculated for each transition object in the SpinSystem.

Here it is used that:

$$\int_0^\infty \rho(t)\, dt = -\hat{\mathbf{A}}^{-1} \rho(0), \tag{13}$$

as can be derived using the properties of the Laplace transformation.

5.10 Task class: StaticHS-SymmetricDecay

Brief description

Efficient quantum yield method relying on some assumptions.

Limitations, approximations and comments

- The interactions should be time-independent (otherwise they are just evaluated at time 0).
- Only a single spin-independent reaction is present (i.e. "symmetric decay" as all states decay with same rate).
- Only the task class RP-SymmetricUncoupled is faster and more efficient than this method.
- Alternative type name: *StaticHS*.

Parameters

Parameter	Values	Default	Description
DecayRate ReactionRate RateConstant	<float>	0.001	The spin-independent rate constant to use (in ns^{-1}). If the rate constant is not specified, the rate of the first defined Transition object in each spin system is used. If no Transition objects are defined in a spin system, the default value will be used.

Technical details

Calculates the quantum yield by solving the equation:

$$\Phi_i = \sum_{n,m} \langle n|\mathbf{P}_i|m\rangle \langle m|\rho(0)|n\rangle \frac{k^2}{k^2 + (\omega_n - \omega_m)^2}, \tag{14}$$

where $\rho(0)$ is the initial state density operator, ω_n is an eigenvalue of the Hamiltonian (upto a factor of $\hbar$), $|n\rangle$ and $|m\rangle$ eigenstates of the Hamiltonian, k is the spin-independent rate constant, and $\mathbf{P}_i$ is the projection operator onto state i. Quantum yields Φ_i are calculated for all State objects defined in the spin system.

5.11 Task class: RP-SymmetricUncoupled

Brief description

Very efficient quantum yield method subject to rather strict limitations.

Limitations, approximations and comments

- Works only for radical pairs.
- The interactions should be time-independent (otherwise they are just evaluated at time 0).
- Only a single spin-independent reaction is present (i.e. "symmetric decay" as all states decay with same rate).
- The two radical should be uncoupled (i.e. no double-spin interaction such as the exchange or magnetic dipole-dipole interaction between the two unpaired electrons).
- Which spins belong to each radical is inferred based on the "type" parameter of the spins: only two spins should be given the type "electron" (one per radical).
- This method calculates only the singlet quantum yield for the radical pair, and the triplet yield is then inferred as these two yields should add up to 1.
- Alternative type name: *RP-Uncoupled*.

Parameters

Parameter	Values	Default	Description
Tolerance	<float>	1e-10	Quantum yield contributions smaller than this tolerance level are neglected.
DecayRate ReactionRate RateConstant	<float>	0.001	The spin-independent rate constant to use (in ns^{-1}). If the rate constant is not specified, the rate of the first defined Transition object in each spin system is used. If no Transition objects are defined in a spin system, the default value will be used.

Technical details

The two radicals are treated independently and with their own single-radical spin Hamiltonians. The first radical is referred to as radical A, and the other radical as B. The method calculates the quantum yield by solving the equation:

$$\Phi_i = \frac{1}{4} + N \sum_{p,q} \sum_{i \neq j} \sum_{g \neq h} R^A_{pq} R^B_{pq} \frac{k^2}{k^2 + (\omega^A_i - \omega^A_j + \omega^B_g - \omega^B_h)^2}, \tag{15}$$

where ω^m_i is an eigenvalue of the single-radical Hamiltonian for radical m (upto a factor of $\hbar$), k is the spin-independent rate constant, and R^m_{pq} is defined by:

$$R^m_{pq} = \sum_{i,j} \mathrm{Tr}\left(\mathbf{S}_{m,p} |i\rangle\langle i| \mathbf{S}_{m,q} |j\rangle\langle j|\right), \tag{16}$$

where $|i\rangle$ are the eigenstates of the spin Hamiltonian for radical m, and $\mathbf{S}_{m,p}$ are the spin operators for the unpaired electronic spin of radical m with $p \in \{x, y, z\}$.

5.12 Task class: Gamma-Compute

Brief description

Implementation of the γ-COMPUTE algorithm. Similar to PeriodicHS-TimeEvolution except that an average of the phase of the periodic Hamiltonian is also performed, and this method only calculates quantum yields.

Limitations, approximations and comments

- *Work-in-progress...*

Parameters

Parameter	**Values**	**Default**	**Description**
Steps	<integer>	100	Splits each period into this number of intervals. The Hamiltonian is assumed to be constant in each of these intervals..
TotalTime Period	<float>	10000	Total integration time (in ns).

Technical details

To be added...

5.13 Task class: HamiltonianEigenvalues

Brief description

Calculates eigenvalues of the spin Hamiltonian. Can also calculate resonance frequencies and other properties of the Hamiltonian (see parameters).

Limitations, approximations and comments

- The resonance frequency analysis is a secondary result and is therefore printed in the logfile rather than the datafile.
- Alternative type name: *Eigenvalues.*

Parameters

Parameter	**Values**	**Default**	**Description**
Superspace UseSuperspace	<bool>	false	Whether to use the superspace Hamiltonian rather than the native Hilbert space ("normal") Hamiltonian for all calculations.
Eigenvectors PrintEigenvectors	<bool>	false	Whether eigenvectors should be printed.
Hamiltonian PrintHamiltonian	<bool>	false	Whether the Hamiltonian should be printed.
SeparateReal SeparateImaginary SeparateComplex	<bool>	false	Whether output should be written as complex numbers (false) or as separated real and imaginary parts (true).
ResonanceFrequencies Frequencies Resonances	<bool>	false	Whether to perform a resonance frequency calculation.
SpinList	<Spin list>	*empty*	A comma-separated list of spins. Transition matrix elements will be calculated for these spins if a resonance frequency analysis is performed.
InitialTime StartTime Begin	<float>	0	The initial time (in ns) used for all calculations.
TotalTime StopTime End	<float>	0	If this is larger than the initial time, all calculations will be performed at times from initial to total time (in ns), as defined by the timestep parameter.
Timestep	<float>	1	Timestep (in ns) to use if a totaltime larger than initialtime is specified.
ReferenceStates RefStates	<State list>	*empty*	Comma-separated list of states. The projection of each eigenstate onto these states is calculated.

Technical details

The eigenvalues and eigenstates are calculated by diagonalization of the Hamiltonian at a given time (time 0 by default, but the calculations can be done for different times using the initialtime, totaltime and timestep parameters).

Resonance frequencies are calculated as the difference between pairs of eigenvalues, and are calculated for each pair of eigenvalues. If a spinlist is specified, the following transition matrix elements are also calculated for each specified spin:

$$T_{i,x} = \langle v_p|\mathbf{S}_{i,x}|v_q\rangle, \qquad T_{i,y} = \langle v_p|\mathbf{S}_{i,y}|v_q\rangle, \qquad T_{i,z} = \langle v_p|\mathbf{S}_{i,z}|v_q\rangle, \tag{17}$$

where i is the spin index from the spinlist, $|v_p\rangle$ and $|v_q\rangle$ are eigenstates of the Hamiltonian, $\mathbf{S}_{i,r}, r \in \{x,y,z\}$ are the spin operators, and $T_{i,r}, r \in \{x,y,z\}$ are the transition matrix elements.

The reference states, if specified, provide the projections of eigenstate onto these references, i.e.:

$$R_{i,p} = \langle v_p|\mathbf{P}_i|v_p\rangle, \tag{18}$$

where $R_{i,p}$ is the projection of the pth eigenstate onto reference state i, with corresponding state projection operator $\mathbf{P}_i$ and eigenstate $|v_p\rangle$. This is calculated for all eigenstates.

6 Preprocessor directives

Similar to the C programming language, the MolSpin input format allows the use of a variety of preprocessor directives. Such directives are prefixed with the symbol #, but the directives are slightly more restricted than in C since they cannot be invoked inside the definition of objects (e.g. Spin or Transition objects), and the *include* directive cannot be used within an object group either (spinsystem, settings or run section). The main reason for this limitation is that the directives are read "on the fly" while parsing the input files, rather than an actual preprocessing (although the "preprocessor directive" term was adopted for its similarity with C).

6.1 #include <filename>

The *include* directive can be used to split your input into multiple files. You can, for example, define your spin system in one file, and use it for different purposes by defining settings and run sections in other files including your spin system file.

Note that MolSpin keeps track of the files that are included, such that repeated or recursive inclusion of the same files is prevented.

6.2 #define <name> <value (optional)>

If used without a value, the define directive can only be used by other directives such as *ifdef* or *ifndef*. This way it is possible to e.g. create so-called *guards*[6] or to easily switch several lines on/off by commenting in/out a define directive.

If a value was also supplied, any occurrence of the defined name in any object will be replaced by the value. *Note the limitation here:* Only the *contents of objects*[7] can be replaced by the defined value. If you need to use spaces in either the name or value of your definition you can write them as strings, i.e. between quotes ("this is a string"), but remember that strings are always treated as case sensitive in MolSpin!

6.3 #undef <name>

Removes a definition (regardless of whether the definition had an associated value). The definition is valid from the point in the input file where the define directive is made and until the *undef* directive.

6.4 #ifdef <name>

Short for "if defined". This directive does nothing if the specified name was defined previously by a define directive, but if the specified name was *not* defined previously (or if the definition was removed previously by *undef*), all the subsequent contents of the input file is ignored until an *endif* or *else* directive is encountered.

Should always have a corresponding *endif* directive.

6.5 #ifndef <name>

Short for "if not defined". Same as the *ifdef* directive except that the following contents are instead ignored if the specified name *was* defined previously. Thus this is the logical negation of the *ifdef* directive (i.e. opposite behaviour).

Should always have a corresponding *endif* directive.

[6] *Guards* is a standard technique in C or C++ programming to prevent inclusion of the same file twice by putting all the code between *ifndef* and *endif*, where the first line after *ifndef* makes the define directive that was just checked. This is used in all the header (.h) files of the MolSpin source code.

[7] Objects refer to Spins, Interactions, Transitions, etc. but not object groups (SpinSystems, Settings and Run Section), and the define directives are only valid for their *contents*, i.e. everything between { and }. Thus a define directive has no effect on e.g. the name of a Spin object.

6.6 #else

This directive can be used in between an *ifdef* or *ifndef* directive and its corresponding *endif* directive. Either the input between *ifdef*/*ifndef* and *else* or between *else* and *endif* will be ignored, i.e. the *else* directive “switches” between ignoring and not ignoring the lines between *ifdef*/*ifndef* and its corresponding *endif*.

6.7 #endif

Denotes the end of an *ifdef* or *ifndef* directive.