## Spinach package – kernel functions

The code base of Spinach has two parts: the kernel, containing very general functions applicable to any spin system, and the user-land – a collection of case-specific functions, experiment models and pulse sequences.

At the start of the simulation, the spin system information is either supplied directly to the kernel, using the syntax documented in the *Input Preparation* section, or read in from a third-party data file using one of the import filters in the user-land (*Gaussian03* logs and *Simpson* \*.in files are currently supported). The kernel oversees the utilization of this information in the most efficient way possible and supplies the user-land with directly usable objects, such as superoperator matrices (Hamiltonian, relaxation, kinetics, *etc.*), state vectors and propagators.

The kernel guarantees that the matrices and vectors it provides may be used directly, meaning that any existing Liouville-space spin dynamics simulation code may be ported to *Spinach* (and considerably accelerated for large spin systems) by simply using the superoperators and state vectors supplied by the *Spinach* kernel instead of the usual direct product matrices.

The kernel code is very well commented and written with clarity and portability in mind. It is always a good idea to have a look inside the function you are going to use before using it – the header would often contain specific detail about the call syntax and output.

1. **Kernel data structure**

The kernel keeps all simulation information in a single structured array called *spin\_system*, which is generated by *create* and updated by *basis* constructor functions. Its subfields are listed in the *developer\_notes* directory. The user-land can use this structure to retrieve information about any aspect of the simulation. The *spin\_system* structure is also used throughout the kernel as a source of information used to generate various infrastructure objects (*e.g.* superoperators) and perform infrastructure operations (*e.g.* propagation) in the most efficient way possible.

1. **Spin system creation and basis specification**

These functions must be the first kernel call, the details of their use are explained in the *Input Preparation* and *Basis Selection* sections. They control the parameters of the simulation, the structure of the spin system, tolerances on all approximations, output level and algorithm selection. After all parameters are processed, these functions create the *spin\_system* data structure. Call syntax:

spin\_system=create(sys,inter);

spin\_system=basis(spin\_system,bas);

where the *sys* and *inter* structures are described in the *developer\_notes* directory. It is often convenient to generate *sys* and *inter* structures using one of the import filters available in the user-land (see the *Input Preparation* section). Both functions produce extensive diagnostic output about the structure and interactions found within the spin system. All approximations and assumptions are also explicitly printed.

*Spinach* supports complete, restricted and connectivity-adaptive basis sets. The greater the coupling density in the spin system and the longer the simulation, the bigger basis is required. Liquid state NMR simulations (pulse-acquire, DQF-COSY, HSQC, *etc.*) are accurately simulated with IK-1 and IK-2 basis sets; reducing the basis causes a gradual loss of multiplicity detail. This basis size dependence is similar to the one observed in the electronic structure theory – the bigger the basis, the more accurate the result. The basis selection considerations are detailed in the *Basis Set Selection* section.

Every approximation in *Spinach* has an associated set of tolerances, which may be altered by setting sys.tols.\* subfields (the complete list is given in the *Internal Tolerances* section). The defaults are very conservative and guarantee accurate results in a large variety of simulations. Relaxing these tolerances from their default values would in many cases significantly accelerate the simulation.

1. **Product superoperators, commutation superoperators and state vectors**

Product, commutation and anti-commutation superoperators are available by using the following syntax:

L=c\_superop(spin\_system,opspec);

L=a\_superop(spin\_system,opspec);

L=p\_superop(spin\_system,opspec,side);

where L is the superoperator in question, side can be *‘left’* or *‘right’* and opspec is the *Spinach* operator specification – a compact internal notation used to represent spin operators and states. An opspec is a string of integers giving the states of each individual spin in the order of their appearance in the sys.isotopes variable. The state numbering lists irreducible spherical tensors by ascending ranks and within ranks by descending projection:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | index |
| 0 | 0 |  |  |  | 0 |
| 1 | 1 |  |  |  | 1 |
| 1 | 0 |  |  |  | 2 |
| 1 | –1 |  |  |  | 3 |
| 2 | 2 |  |  |  | 4 |
| 2 | 1 |  |  |  | 5 |
| 2 | 0 |  |  |  | 6 |
| 2 | –1 |  |  |  | 7 |
| 2 | –2 |  |  |  | 8 |

*et cetera* (Spinach supports all spin quantum numbers). The sequence of integers in opspec is mapped by the kernel into the sequence of operators in the direct product, for example:

[0 2 0 3 1 0 2 1 0 0] => 

The same syntax is used by the statevec.m function that returns the state vectors:

rho=statevec(spin\_system,opspec);

It is not always convenient to supply the superoperator and state specification in the form described above, and the user-land has wrapper functions providing a more eye-friendly syntax:

rho=state(spin\_system,oper,spins)

L=operator(spin\_system,oper,spins)

where spins may be set to *‘all’, ‘1H’, ‘13C’, etc.* and oper to *‘Lz’, ‘L+’* or *‘L-’*. These functions parse the user-friendly input, convert it into opspec and call statevec and c\_superop respectively. The precise details of all calls accepted by c\_superop, a\_superop, p\_superop, statevec and operator are given in the headers of the corresponding functions.

1. **Secularity assumptions**

Because the rotating frame / secularity assumptions differ between simulation types, the secularity.m function must be called to set these assumptions, for example:

spin\_system=secularity(spin\_system,’endor’);

The following secularity assumptions are supported at the time of writing:

|  |  |
| --- | --- |
| 'nmr', 'esr' | Rotating frame with respect to all spins. For Zeeman interactions, keep the offset ZZ term. For couplings, keep the secular terms for spins belonging to the same isotope and weak terms otherwise. |
| 'lowfield', 'keep\_all' | Lab frame. For Zeeman interactions, keep everything. For couplings, keep everything. |
| 'eseem', 'endor', 'solid\_effect' | Electron rotating frame. For electron Zeeman interactions, keep the offset ZZ term; for nuclear Zeeman interactions, keep everything. Couplings from electron to nucleus are electron-secular, couplings between electrons are secular, couplings between nuclei are strong. |

The responsibility of setting specific secularity assumptions rests with the user. See IK’s Spin Dynamics course for a general introduction to secular approximations.

1. **Isotropic part of the Hamiltonian and the rotational basis**

The kernel function returning the isotropic Hamiltonian commutation superoperator has no adjustable parameters:

H\_iso=h\_superop(spin\_system);

except for the option to ignore the secularity status of all interactions and return the full non-secular laboratory frame Hamiltonian superoperator:

H\_iso=h\_superop(spin\_system,’full’);

This second option is most commonly invoked by the relaxation superoperator module and may be useful in the rare cases where a laboratory frame simulation needs to be performed. A call with two output parameters

[H\_iso,Q]=h\_superop(spin\_system);

returns the full rotational basis , computed using Equation 11 in IK’s relaxation theory paper (<http://dx.doi.org/10.1016/j.jmr.2010.12.004>), for the anisotropic part of the Hamiltonian commutation superoperator.

The function prints summaries of all interactions it encounters, along with their secularity status and spherical tensor expansions. Note that the summaries correspond to the input orientation of all anisotropic couplings.

1. **Anisotropic part of the Hamiltonian**

The anisotropic part of the Hamiltonian commutation superoperator for a specific spin system orientation may be obtained by supplying the rotational basis (see the previous section), along with the three Euler angles, to the orientation function:

H\_aniso=orientation(Q,[alpha beta gamma]);

If a cell array of Euler angle vectors us supplied, a cell array of Hamiltonian superoperators is returned. This function is used, in particular, during the calculation of powder averages using pre-computed orientation sets corresponding to Lebedev spherical integration grid points. Powder averaging in general is a user-land problem – the kernel only provides the grids (see the exp/grids directory).

1. **Relaxation superoperator**

The function itself has no adjustable parameters:

R=r\_superop(spin\_system);

and uses the information (theory, correlation time, *etc.*) that the user has supplied to the create function (see the *Input Preparation* section of this manual).

*Spinach* implements a very general case of Bloch-Redfield-Wangsness relaxation theory, which includes all “difficult” contributions, such as interaction rhombicities, cross-correlations and dynamic frequency shifts. Contributions from all couplings present in the system (including quadrupolar and ZFS) are included. So long as the validity conditions of Redfield theory are satisfied, *Spinach* is not restricted to high-field systems – the role of  can be played, for example, by a large isotropic hyperfine coupling. The relaxation superoperator evaluation procedure is specifically designed to allow very large matrix dimensions to be processed.

Relaxation to thermal equilibrium, if requested by setting inter.equilibrium to *‘thermal’*, proceeds using the method outlined by Levitt and Di Bari, whereby the relaxation to the equilibrium state vector 



is introduced as an extra row and column in the Liouvillian:



Because the first element of any *Spinach* basis is the unit operator, this change of dimension is not required and the procedure amounts to introducing a one-way cross-term to the unit operator. The time propagation functions (evolution.m and step.m) are aware of this option and will automatically relax the system towards the equilibrium prescribed. Note that the population of the unit state will be forced to 1 if relaxation to thermal equilibrium is requested.

1. **Chemical kinetics superoperator**

The chemical kinetics superoperator is requested by:

K=k\_superop(spin\_system);

and uses the information from inter.chem variable that the user supplied to create.m function (see the *Input Preparation* section of this manual). *Spinach* takes advantage of the fact that bosons and fermions of the same type are fundamentally identical and assumes that chemical exchange processes amount to the *transport and flux of magnetization* within a topologically fixed spin system rather than to moving spins. For chemical exchange, networks of first-order chemical processes are supported:



where  is the concentration of spin  and  is the rate of magnetization transport from spin  to spin . An important condition on the elements of the kinetics matrix  is that the diagonal elements must balance out the off-diagonal elements so that the law of the conservation of matter is observed.

From the simulation perspective, when magnetization is pumped over from site  to site  in a spin system, the state populations are pumped across between the following states:



where  is an index running over the elements of the spin state space. The task of constructing a superoperator that would perform this action amounts to subtracting a slice of the population from  and forwarding it to :

,

and the superoperator in brackets shifts the population from one state to another [45].

The policy on chemical transport of multi-spin orders is case-dependent. In principle, the coherence with any observer spins must be preserved during chemical exchange, that is:



and, so long as we are treating a single system (or an ensemble of systems, which have chemical exchange happening inside them), this should be taken into account. In a typical solution, however, this would lead to the emergence of inter-molecular coherences between identical molecules – a situation that density matrix formalism is ill-equipped to accommodate. Because such coherences are non-observable and the spin is very unlikely to jump back to the exact molecule that it originally came from, the coherences in question are counted as lost:



and the corresponding contribution to the kinetics superoperator effectively causes relaxation.

For reactive radical pair simulations of the kind often encountered in Spin Chemistry, the singlet and triplet recombination superoperators may be included into  by setting the inter.chem.rp variables (see the *Input Preparation* section of this manual). The theories available at the time of writing are Haberkorn and Jones-Hore:





where  are singlet and triplet radical pair recombination rate constants,  denotes an anticommutator, and  are singlet and triplet projection operators.

1. **Exponential propagator**

For a given Liouvillian superoperator  and a given time step, the exponential propagator  can be requested as:

P=propagator(spin\_system,L,delta\_t);

The seemingly unsophisticated choice of matrix exponentiation algorithm in *Spinach* – Taylor series with scaling and squaring – is a consequence of the need to handle very large sparse matrices, which must stay sparse and for which only approximate norms are known. All other methods either involve dense matrices (*e.g.* the division operation in the Padé method and the diagonalization in the eigenvalue method) or place strict constraints on the matrix eigenvalues (Chebyshev exponentiation). In practice the scaled Taylor series converges to machine precision in 10-15 iterations. The use of Chebyshev approximation can be requested by setting the sys.tols.exponentiation switch (see the *Internal Tolerances* section) to ‘chebyshev’.

The propagator function monitors the matrix density and switches over to dense algebra if it cannot avoid exceeding the matrix density threshold, which is set at 25% by default. For small time steps, the number of non-zeros in the propagator is similar to that of the Liouvillian. It increases rapidly as soon as the time step crosses the  threshold.

Derivatives of the exponential propagator with respect to an arbitrary linear parameter  of an arbitrary operator  occurring in  are available by the use of the following syntax

P=propagator(spin\_system,L,delta\_t,derivatives);

where derivatives is a cell array of matrices corresponding to the  operators for all the necessary parameters .

1. **Thermal equilibrium**

Has to be updated.

Because longitudinal spin states in Liouville space correspond to *polarizations* rather than *populations*, the state vector returned by equilibrium would in some cases contain small numbers (*e.g.* for 15N at room temperature). It is therefore advisable, when running with accurate thermal equilibria at high temperatures, to inspect the trajectory-level state space reduction tolerances and make sure that important states are not dropped automatically because of their low occupancies. The default tolerances are in most cases tight enough.

It should also be noted that setting inter.temperature variable to be *identically* equal to zero does *not* collapse the system into the lowest possible collective energy level, but causes equilibrium.m to return the simplified equilibrium state that is often used in basic NMR and ESR simulations: . This is the default.

1. **Trajectory-level state space reduction**

Even the reduced basis sets, such as IK-2 and ESR-1 (Table S2 in the Supplementary Information), often turn out to be excessive and contain unpopulated dimensions [28], which are specific to the experiments being simulated. They can be pruned using the Zero Track Elimination procedure (Section II.4) by calling

P=zte(spin\_system,L,rho);

where L is the Liouvillian superoperator, rho is the initial state vector and P is a matrix projecting the system from the current to the reduced basis set and back:



The reverse transformation amounts to placing zero tracks back to their original locations. The detailed analysis of the ZTE technique is given in our recent paper [28] and summarized in Section II.4 above.

While diagonalization is in most cases unfeasible even for reduced Liouvillians because of its  cost in time and memory, a sparsity-preserving block-diagonalization can be achieved at  cost using the connectivity tracing procedure proposed in our recent paper [26] and summarized in Section II.9. Projectors into the subspaces that are disconnected in the current basis may be requested by calling

P=path\_trace(spin\_system,L);

where P is a cell array of projectors into disconnected subspaces. The projectors are applied in the same way as the ZTE projector in Equation . The efficiency of the path tracing procedure depends on the choice of the basis set. For the IST basis sets used in *Spinach*, there are always at least two (*e.g.* mary\_test\_1.m), and sometimes over a hundred (*e.g.* hsqc\_test\_1.m) independent subspaces, depending on the calculation type and spin interactions present.

ZTE, path tracing and symmetry factorization can be applied sequentially (symmetry, then ZTE, then path tracing) using a wrapper function that returns the projectors into the resulting set of disconnected minimal subspaces

P=reduce(spin\_system,L,rho);

where P is a cell array of projectors that may be used as prescribed by Equation . Individual algorithms may be disabled by setting the sys.disable switch before the call to create (Table S1 in the Supplementary Information). Unless they are specifically disabled, trajectory-level pruning algorithms are applied automatically and transparently every time a call is made to the time evolution function.

1. **Time evolution**

Two functions are available for moving the state vector forward in time under a given Liouvillian. A single propagation step  under a Liouvillian  (*e.g.* a hard pulse with  and ) is best performed with Krylov algorithm using

rho=step(spin\_system,L,rho,delta\_t);

where rho is the state vector. For a single propagation step, the Krylov algorithm is faster than matrix exponentiation because it computes  directly from  and  using only matrix-vector multiplications (details are given in Section II.5). The step function is also optimal in the case when the Liouvillian is time-dependent (shaped pulses, *etc.*) and many steps need to be taken with different values of .

For long trajectories under time-independent Liouvillians, the kernel provides a fairly sophisticated wrapper function, which automatically takes advantage of the trajectory-level state space reduction functions described in Section III.9.

● The final state vector after a period of evolution can be requested by:

rho=evolution(spin\_system,L,[],rho,timestep,nsteps,’final’);

Because only the final state is requested, the step size and the number of steps are adjusted internally (keeping the overall evolution time static) to ensure optimal performance. A horizontal stack of state vectors can be supplied, in which case every vector in the stack is taken forward by the same time interval.

● System trajectory for a given number of steps under a given Liouvillian can be requested by:

rho=evolution(spin\_system,L,[],rho,...

timestep,nsteps,’trajectory’);

The output variable rho contains the state vectors for every point in the trajectory concatenated horizontally (a total of nsteps+1 columns). The first point is the initial state. A specific situation often encountered in NMR and ESR spectroscopy is a 180-degree “refocusing” pulse in the middle of an incremented evolution period. The stack of final states for each increment in the duration of the evolution period can be requested using the *‘refocused\_trajectory’* option:

rho=evolution(spin\_system,L,[],rho,...

timestep,nsteps,’refocused\_trajectory’,R);

where R is the cell array of generators of the refocusing pulse (*e.g.*  in the case of a 180-degree pulse on the X-axis), which will be applied in the order of appearance in R.

● The dynamics of the observable corresponding to a given state vector can be requested by:

observable=evolution(spin\_system,L,coil,...

rho,timestep,nsteps,’observable’);

where coil (called so for obvious reasons) is the state vector corresponding to the detection state (*e.g.*  in the case of quadrature detection). If a horizontal stack of state vectors is supplied as the initial condition, a vertical stack of observable traces is returned, individual lines corresponding to the observable traces starting from each of the initial conditions supplied. Multi-coil detection can be requested by calling:

observable=evolution(spin\_system,L,coil,rho,...

timestep,nsteps,’multiple\_observables’);

and supplying a horizontal stack of state vectors in the coil variable – a multi-line answer will be returned with lines corresponding to individual columns of the coil variable.

The best illustration of the practical use of these options is the source code of the 2D NMR and DNP pulse sequences in the user-land (Section IV and exp directory of the Spinach distribution).

1. **Optimal control waveform optimization**

*Spinach* implements the GRAPE (gradient ascent pulse engineering) procedure [84-85] for optimal control based waveform optimization. The calling syntax is:

[objective,gradient]=grape(spin\_system,drift,controls,...

waveform,time\_step,nsteps,...

starting\_state,target\_state);

in which drift is the static “drift” Liouvillian, controls is a cell array of control superoperators, waveform is a row vector giving the control amplitudes at each step (for multiple control operators, the waveforms should be concatenated horizontally in the order in which the control operators are listed in controls), starting\_state is the initial condition state vector and target\_state is the destination state vector, which should be populated to the maximum possible extent by the optimized waveform. The objective output is the real part of the scalar product between the final state and the destination state, and the gradient output is the gradient of the objective function with respect to the waveform parameters, evaluated at the current value of waveform.

The utilization of the resulting objective function and gradient is a subject of active current research; from the kernel perspective, this is a user-land problem. Examples of using *Matlab*’s built-in L-BFGS [86] optimization module (a part of the *Optimization Toolbox*) are given in the examples directory. It should be noted that the first-order approximation to the GRAPE gradient is often unsuitable for BFGS runs [87] due to its limited accuracy, an exact gradient option is provided for this purpose. This option is chosen by default, unless the user manually specifies *‘approximate’* in the sys.tols.grape\_gradient subfield when calling create (Table S3 in the Supplementary Information).