pNMRsim

This document describes a simulation program for solid-state NMR that is being developed in the Durham NMR group. *Before thinking about using the program, do read the section at the end which tries to put you off.* V*YY.MM.DD* figures in the text indicate the "version" (labelled by date) when a particular feature / change was introduced (see also the `-news` output from pNMRsim). This document covers V15.08.13.

Introduction

The SIMPSON simulation package allows relative novices, or those who just don't care about the working of numerical simulations, to perform challenging simulations using a fairly straightforward, quickly edited, input file. This is much simpler and quicker than writing dedicated programs, even using specialist NMR libraries e.g. GAMMA, and (in most cases) this outweighs the performance penalties of using a general rather than bespoke program. That said, SIMPSON has some weaknesses, particularly in the area of large spin systems where SPINEVOLUTION is often used in preference.

pNMRsim has taken the SIMPSON philosophy and developed it to tackle large problems efficiently:

- The input file has the same basic syntax as the SIMPSON .in file. Some areas, however, have been significantly changed. For examples, pulse programs are represented at a relatively "high level", allowing complex pulse programs to be expressed efficiently, reducing the requirement for underlying language support, and increasing the opportunities for optimisation.
- The range of problems that can be handled efficiently is significantly extended (block diagonal Hamiltonians, spatially periodic Hamiltonians, phase transients and timing errors etc).
- Coupled quadrupoles are handled correctly, although this "generalised quadrupole" code has not been exhaustively tested.

Do note, however, that pNMRsim has been developed to tackle a particular range of systems and there are many areas, where SIMPSON and SPINEVOLUTION have particular functionality e.g. handling relaxation, which pNMRsim does not attempt to provide.

Input file format

Like SIMPSON, pNMRsim uses a plaintext input file which conventionally has the file extension `.in` (although it is not obliged to do so). Lines with # *as the first character* are treated as comments and are discarded. At the next step up from this level lies the overall structure of the input files, which consists of "blocks" of the form

```
<block name> {
<instructions>
}
```

The blocks, if present, must be in the order

`spinsys` Sets up the system Hamiltonian. This can be omitted when the FID/spectrum is being synthesised explicitly e.g. with `addsignals`.

`par` Sets up the remaining definitions used in later blocks: simulation parameters, pulse sequence elements etc.

`pulseq*` Optionally defines the nature of the pulse sequence e.g. pulses applied before detection[1]. If omitted, this defaults to `acq` i.e. acquire NMR signal. It cannot be present if a `spinsys` block was not included.

`initialproc*` Optionally defines any processing to be applied to the raw NMR signal, before summation of individual transients[2]. *(Removed in V11.06.14 to enable better parallelisation; restored in V12.08.02).*

`proc*` Defines the processing applied to the accumulated NMR signal. If omitted, the FID/spectrum is saved with a filename derived from the `.in` file.

`optimise` (V12.08.02) Additional optimisation instructions.

`finalise*` Optional instructions to be executed when pNMRsim terminates.

If an unexpected block name is encountered, it is assumed to be an "`include`" definition (see below).

Blocks marked * can be used multiple times. This allows different spectra and different processing to be applied to different rows of a complete data set, which is particularly useful when fitting multiple data sets simultaneously.

Although the format of the input file is closely related to SIMPSON, there a major difference; SIMPSON in effect provides a custom programming language for expressing NMR experiments (via the underlying Tcl). In contrast the pNMRsim input format provides a "description language" in which the problem is fully defined by the time the simulation begins. Hence there is a (fairly) strict "unique definition" rule: the value/values of a quantity are determined when first defined and cannot be changed subsequently. This significantly limits the degree of "programmability", but makes it much easier for pNMRsim to "understand" the structure of the problem.

Each of the standard blocks is considered in turn below. The syntax and usage of most commands closely follows SIMPSON and are only briefly outlined below. Refer to the SIMPSON paper for fuller descriptions. Instructions or additional arguments that are only present or work very differently in pNMRsim are shown in **`bold`**.

`spinsys`

This block defines the nuclear spin system and the "spectrometer" (e.g. RF channels). After parsing this block, pNMRsim has all the information required to construct the spin Hamiltonian and understand its block structure.

The initial lines should contain

`proton_frequency` <*freq*>`*` optional specification of ^{1}H Larmor frequency e.g. `proton_frequency 500e6` for a 500 MHz spectrometer[3].

`usernucleus` <*name*> <*name*>|(<*I*> (<*freq*>|<*gamma*> `-gamma`)) defines a new nucleus type with the quantum number and magnetogyric ratio specified either using an existing nucleus name e.g. `13C`, or with an explicit quantum number and Larmor frequency / gamma value (in T^{-1} s^{-1}). The proton frequency must have previously been defined in the former case. Because the interactions between nucleus with different types are always treated as heteronuclear, defining a new nucleus type in terms of an existing one is useful for grouping spins that can be considered as weakly coupled from other groups of spins and can be selectively irradiated e.g. `usernucleus CO 13C` would allow spins of type CO (e.g. carbonyl carbons) to be addressed with a separate RF channel.

`spinsys` <*nucleus*>`+` (+ denotes "1 or more of") e.g. `spinsys 1H 13C` defines a two spin system with spin 1 being a ^{1}H and spin 2 a ^{13}C. Spins are numbered from 1. The nucleus identity is primarily used to distinguish nuclei types and to determine Larmor frequencies. In the case of half-integer quadrupolar nuclei, the tag "`:c`" can be used to restrict the states[4] considered to the central transition only (V09.01.03), e.g. `11B:c`. This greatly speeds up the calculation, at the expense of including the satellite transitions.

One particular feature of pNMRsim is the ability to handle systems with spatial periodicity. This is specified using

`cells` *n*+ specifies a periodic geometry in terms of the number of "unit cells" along 1 to 3 spatial dimensions e.g. `cells 5` corresponds to a one-dimensional geometry of 5 cells, `cells 2 3` declares a system of 2 by 3 cells etc. Unless disabled by `-disable:periodic` (see below) or the support is missing in your version of pNMRsim, the presence of cells will allow the Hamiltonian to be further block diagonalised, greatly improving the efficiency of calculation.

One the spin system has been specified, the various NMR interactions can be added:

`shift` *n* <*iso*> [<*CSA*> <η> [α β γ]] specifies the chemical shift of spin *n* in terms of the isotropic shift, anisotropy and asymmetry of CSA and the Euler angles defining the orientation of the CSA with respect to the crystal orientation (these default to 0 0 0 if omitted). Angles are always expressed in degrees, while the units of frequencies, such as the isotropic shift and CSA, are Hz. Note that *n* must be within the range of the "unit cell" as it is impossible for chemical shifts to differ between cells. Alternatively shifts may be specified in ppm by including a trailing `p` e.g. `shift 1 1p` would specify an isotropic shift of 1 ppm. The NMR frequency must have been previously specified using `proton_frequency`.

`dipole` *n m* <*coupling*> [<η>|α β γ] sets a dipolar coupling between spins *n* and *m* accompanied by an effective asymmetry (for liquid crystal NMR) or a set of Euler angles defining the orientation of the internuclear vector with respect to the crystal frame of reference. Inter-cell couplings are specified when one (but not both) of the spin indices lies outside the unit cell. So if there are *m* spins in the unit cell `dipole 1` *m*`+1` … sets the coupling between spin 1 of the base unit cell and spin 1 of its next nearest neighbour. Alternatively the syntax *n,cell* (e.g. `1,1`) can be used. The coupling network must be properly periodic for the calculation to be valid, and an error is generated showing which couplings do not match if this is not the case.

A warning is generated if the dipolar coupling appears to have the wrong sign (based on the signs of the magnetogyric ratios). This check is not always appropriate e.g. when dealing with motionally averaged couplings and can be disabled with `-nochecks`.

`jcoupling` *n m* <*iso*> [<*aniso*> <η> [α β γ]] sets a *J* coupling between spins *n* and *m*. The anisotropic component of *J* (which is normally negligible in comparison to dipolar couplings) can be omitted.

`quadrupole` *n* <*order*> <*aniso*> <η> [α β γ] sets the quadrupole coupling for spin *n*. The order can be 0, 1 or 2: 1 corresponds to a first order calculation, 2 uses second-order perturbation theory while 0 does an "exact" calculation which copes with arbitrarily large quadrupoles. The coupling to quadrupolar nuclei should be handled correctly, ***but non-first order quadrupoles have not been fully tested***. Calculations with non-secular Hamiltonians are also much slower and should always be avoided if possible.

By default, 2nd order effects are implemented "classically" and are restricted to isolated (single) quadrupolar spins. When dealing with multiple spins and second-order quadrupoles, one of two treatments must be explicitly enabled: `-enable:generalisedQ` to enable the experimental general treatment or `-enable:classicQ` to force the "classic" treatment which does not handled transferred quadrupole effects (and should return the same results as SIMPSON). Whichever algorithm is used, subtle features (such as Berry's phase effects) will be missed when pushed e.g. quadrupole couplings that are comparable to the Larmor frequency or spinning rate, and the use of RF with anything other than first-order quadtupoles also raises warnings.

The default nucleus properties (compile-time option in V15.08.13) are taken, like SIMPSON, from the 2001 IUPAC values published by Harris et al. Previous versions used a slightly smaller subset of older values as used by the GAMMA libraries. `pNMRsim -version` will

now show `HarrisIUPAC` for "Nuclear spin properties". This item is missing (corresponding to the original values) in previous versions.

The asymmetry of tensor interactions is specified by default in terms of the η parameter. Alternatively the syntax <*aniso*> `-xy` <*xx–yy*> can be used to specify it in terms of the difference between *xx* and *yy* tensor components (V08.03.01). Care is needed when fitting asymmetry parameters that are close to the limits of zero and one, since convergence at a constrained limit can be very slow. Fitting in terms of the unconstrained *xx–yy* parameter can be useful here, with the risk that the solution found may fall outside the asymmetry parameter limits i.e. the *xx*, *yy*, *zz* components are incorrect.

Note that the "Haeberlen-Mehring-Spiess" convention for the ordering of tensor principle components is used by default for all interactions. This follows the practice of programs such as SIMPSON and SPINEVOLUTION, but other software may use different conventions, which will typically affect the interpretation of the Euler angles. This can be over-ridden using:

`tensorordering` <*interaction*>`+` [`-Haeberlen|-NQR`] (V15.08.13) which sets the ordering convention for subsequent interactions of a given type e.g. `tensorordering quadrupole -NQR` would mean that all following quadrupole definitions used the component ordering commonly used in NQR. If the final argument is omitted, the current setting for the specified interactions is displayed. Conventions can be mixed by a given type, but are fixed for a given interaction when it is created.

Whole tensor definition (V08.12.18): defining tensors in terms of separate components is cumbersome if it is necessary to perform computations on the complete tensor. Alternatively tensors may be specified using an ordered six component list containing the isotropic value, anisotropy, asymmetry and the three Euler angles. Hence dipole 1 2 [0,1000,0,0,0,0] would specify a dipolar coupling of 1000 Hz between spins 1 and 2. The `-xy` flag can be added to denote that the asymmetry is being defined in terms of *xx–yy*. However, values must be specified in Hz and not ppm. This "whole tensor" formulation is particularly useful if using an external program to perform tensor calculations e.g. motional averaging e.g. `dipole 1 2 ` `` `myprogram <arguments>` ``.

User defined interactions can be added using **`usershift`** `<name>` and **`usercoupling`** `<name>` respectively, allowing additional shifts and couplings to be specified for a given spin / spin pair. This is useful in solid systems where interactions such as the CSA and ABMS at a site will in general have different orientations and so cannot be merged into a single anisotropic shift. User-defined shifts and couplings are input in the same format as `shift` and `jcoupling` respectively e.g.

```
usershift ABMS
ABMS 1 0.1p 1p 0 0 20 0
```

`truncate` <*interaction*>`+` indicates that the listed coupling interactions should be treated as "weak" i.e. non-secular components ignored even for homonuclear coupling e.g. `truncate jcoupling`. Although the dipole interaction can be truncated, this would only be appropriate in very weakly ordered solutions.

The following instructions are used to define the "spectrometer":

`channels` <*nucleus*>`+` lists the active RF channels e.g. `channels 1H`. It can be omitted if no RF is active (difference from SIMPSON). This should be used sparingly as the presence of an RF channel indicates prevents block-diagonalisation for that nucleus type, slowing the calculation. A dummy channels directive can be useful to explicitly indicate which nuclei should not be subject to blocking in cases where pNMRsim has not been able to determine the block structure.

transients automatic|manual <amp1> … enables simulation of the effect of RF phase transients. The effect of out-of-phase transients is simulated using a pair of counter-rotating ideal tip pulses either side of any (soft) RF period (ideal pulses are, by definition, not subject to phase transient effects). If, as is commonly assumed, transient effects are "linear", then in-phase ("amplitude") phase transients can be neglected to a good approximation. The amplitude parameter specifies the tip angle (degrees) per kHz of RF nutation frequency e.g. 0.1 would give a 5° phase transient with 50 kHz RF cf. A. J. Vega, *J. Magn. Reson.* **170**, 22 (2004). These are specified individually for each of the channels (in order). In the automatic mode, the phase transients are applied automatically to each (soft) pulse unless the transient amplitude is fixed at zero for the channel. In manual mode, the presence of transients in specified on individual pulses using the −transients flag in the pulse definition.

This model is rather crude and a better, if much slower, alternative is to use the expandtrans functions in the fulltrans.inc include file to model the actual transients for phase-modulated sequences.

par

The par block sets up all the information required for the simulation apart from the actual pulse sequence (defined in pulseq). The allowed instructions are summarised below (see SIMPSON documentation for fuller descriptions / examples).

autoopt <*stop*> [powderquality|maxdt|maxjumpdt|gamma_angles|chebyshev_iterations]* [-reset]

allows optimal values for parameters such as the number of crystallite orientations required for effective powder averaging to be determined empirically. One or more parameters are optimised by "incrementing" the parameter (increasing "quality" and time required) until the results of successive calculations are indistinguishable within the <*stop*> parameter. This is specified as the norm of the deviation divided by the norm of the "signal" i.e. a value of 0.001 corresponds to one part in a thousand. Good convergence is important when fitting, since gradient methods in particular may become unstable if the calculations are too rough and ready. The starting point of each optimisation is the value specified in the par block. By default the optimised value of the associated parameter is used in the following calculations unless the −reset flag is specified in which case it is reset to the initial value e.g. autoopt 1e-4 powderquality gamma_angles −reset will find optimal values for the powder quality and gamma angles independently. More than one autoopt directive can be given e.g.

```
autoopt 1e-4 gamma_angles
autoopt 1e-5 powderquality
```

would first optimise the number of gamma angles and then the powder quality (using the previously determined optimal gamma angles). autoopt runs can be performed in advance of any type of calculation.

cache_limit [<limit>] sets a limit (in M) on the memory that can be used to store propagators[5] (default is 50 M). Without an argument, the current limit is displayed. Certain algorithms may work more efficiently if this limit is raised, but raising it too far will result in heavy "swapping" of memory and a rapid degradation of performance. The global −nocache option disables cacheing and so is essentially equivalent to cache_limit 0.

chebyshev_iterations <*n*> sets the number of iterations to use in Chebyshev propagation (MAS problems only, default 9). This parameter will interact with maxdt in the sense that more iterations will be required with coarser time steps. The time taken for

propagation calculation will be approximately proportional to *n* (and inversely proportional to the `maxdt` step).

`crystal_file <name> [-sphere|-hemisphere|-octant] [-start|-middle|-end|-both]` sets up the powder averaging (defaults to single orientation if unspecified). Allowed values for `<name>` are: `zcw`*N* where *N* is the number of orientations (like SIMPSON). Alternatively `zcw:`*S* can be used to specify a set number *S*. `3zcw`*N* or `3zcw:`*S* specifies a 3-angle integration set determined using the ZCW algorithm[6], in terms of either a number of orientations or a set number respectively. Valid numbers for the ZCW sets are given in an error message if *N* is not a recognised value. `beta`*N* corresponds to *N* regularly spaced β angles (α fixed at 0). `alphabeta`*Na*`,`*Nb* gives *Na* regular steps in the α and *Nb* regular steps in β. If `<name>` is in `{}` brackets e.g. `{zcw:8}`, then the powder orientation becomes an arrayed variable i.e. a 2D array will be produced with rows corresponding to different orientations[7].

The flags modify the ranges and details of the sampling. The range qualifiers `sphere` (default), `hemisphere`, `octant`, allowing the range of spherical angles to be restricted. This allows for more efficient powder averaging (especially for `octant`), but only possible if the Hamiltonian has sufficiently high spatial symmetry). Unless checks are disabled, Hamiltonian will be tested for the required symmetry before the calculation starts. The other flags control the exact range of the β Euler angle. If `-both` is specified, the initial β value is 0 (pole) and the final value is π (other pole) or π/2 (equator); `-start` begins at 0, but does not include the maximal value; –end uses the same step but includes the maximal value and not the minimal value; –middle (default) offsets the angle by half step size to avoid both limits. Note that the angular step for `-both` is larger than for the other options. SIMPSON seems to use the equivalent of `-start` while the default pNMRsim is `-middle` i.e. avoiding the pole whose weighted contribution to the signal is zero (for most sampling schemes). The α angle always starts at zero and does not include 2π (like `-start`) unless the `-octant` range is used in which case the same method is for both α and β.

`single` `<alpha> <beta> <gamma>` specifies a single crystal orientation. The NMR response can be sampled at different specified orientations using an `{}` array e.g. `single {0:10:90}:1 {0:10:90}:2 0` would be equivalent to `{alphabeta10,10} -octant -both`.

If `name` is not recognised, an attempt is first made to read `name` (from the current directory) as a simple ASCII file. This should contain 3 or 4 columns depending whether just α, β angles are being specified, or all three (the final column is the weighting factor). If `name` is not found, the SIMPSON-format "crystal orientation" file `name.cry` is attempted (α, β averaging only). No range qualifiers can be applied in this case (or for `single`).

`detect_operator <`*product operator expr*`>*` determines which coherences are measured when the signal is detected e.g. `detect_operator Fx` to detect *x* magnetisation. See below for more information on product operator expressions. If the spin system is homonuclear, `detect_operator` defaults to `Fp` (i.e. +1 coherence). By specifying an array of operator expressions, the detection operator can be changed from row to row e.g. `detect_operator {Fx,Fy}` or `detect_operator cos($phase)*Fx+sin($phase)*Fy`. *Such non-fixed operators limit the scope for optimisation and should only be used when absolutely necessary*.

`echo` `<output>` outputs the rest of the line[8] to the display or `log_file` (if defined), after substituting any $ variables (use `\$` to include a $ character in the output). This is useful for outputting information about the state of the program, or commenting other output e.g. from `putmatrix`. An empty line is printed if `<output>` is empty.

eigenvalue <n> restricts the calculation to eigenvalue *n*. This is only significant for systems with translational symmetry in which *n* refers to the *k* eigenvalue from 0 to *N*–1 where *N* is the number of unit cells.

~~**filter** <*name*> <*coherence lists*> is a shortcut for `matrix set` <*name*> `coherence` <*coherence lists*>.~~

`gamma_angles` *n** number of integration steps for γ powder angle. This only relevant to simulations involving sample spinning. If `gamma_angles` is unset, a single value set by `gamma_zero` is used (a warning will be generated if neither `gamma_zero` or `gamma_angles` has been set explicitly). Obviously `gamma_angles` should not be set if the powder averaging already includes γ. *Note that in cases where gamma angle integration must be done explicitly (when the "γ-COMPUTE" algorithm cannot be used, it is generally better to use a 3 angle integration set rather than set* `gamma_angles`.

`gamma_zero` γ* "zero" value for γ powder angle (defaults to 0). This effectively sets the rotor phase at $t = 0$ for spinning experiments. `gamma_zero` is used as an offset on any explicit γ angle integration.

histogram is used to set the parameters for frequency-domain based acquisition. Multiple histogram commands can be used to set different parameters of the spectral accumulation:

`histogram [`<threshold>`] [-fold]` sets the histogram mode and an optional folding flag. The threshold, if specified, sets the minimum intensity to be included in the histogram. This is particularly useful in the more computationally expensive lineshape mode (see below) for discarded transitions of trivial intensity. If `-fold` is specified then frequencies that fall outside the spectral width are folded back into spectrum (otherwise they are discarded). (N.B. Calculations that proceed via the calculation of propagators intrinsically fold frequencies into a spectral width defined by the sampling rate, so this distinct behaviour of time *vs*. frequency based propagation will only be visible for static Hamiltonians in the absence of RF).

`histogram log_file` <*filename*>`|-` `[`<*threshold*>`] [-double|-binary|-real|-append]` causes the transition amplitudes and frequencies to be streamed to a specified log file (or the standard output is used if the filename is specified as `-`). Output can be limited to transitions with non-trivial intensities by specifying a *threshold* for the transition amplitude (this has no effect on the spectral histogram). The format is ASCII by default, but can be set to single precision or double precision binary using the `-double` and `-binary` flags. This file is intended for temporary output and the binary format will not be portable. The `-real` flag discards the imaginary component on output. The `-append` option (V11.06.14) will append the data to an existing file rather than overwriting. The file may have been generated by `log_file` instructions earlier in the `par` block, but the data must have been written out and the file closed before the `histogram log_file` command.

`histogram range` <*min*> <*max*> restricts the frequency range to be accumulated using minimum and maximum frequencies to be included[9] (rather than the default of the full spectral width). Range restriction is incompatible with the `-fold` option. This range information (rather than the setting of `sw`) is used for the spectral width and frequency origin when creating the data set prior to processing (V11.06.14).

`histogram lineshape <linewidth> [<Guassian fraction> [<steps> [<cutoff>]]]` allows the histogram to be accumulated using finite width Lorentzian/Gaussian lineshapes of a specified width and L/G fraction. Unlike the normal histogram mode where frequencies must be rounded to the nearest bin, finite width lineshapes may be positioned more accurately i.e. the peak maximum can lie between histogram bin centres. *Note that spectral frequencies outside the histogram range are always discarded even if part of the lineshape would lie within range*. `<steps>` specifies the number of steps into which each frequency bin is effectively divided (5 by default).

Particularly as complete lineshapes are being used, there is little improvement on increasing the resolution beyond this–the effects will be negligible unless the histogram binning is extremely coarse. The `cutoff` (zero by default) specifies a cutoff intensity (as a fraction of the lineshape maximum) in the lineshape function. Using a high cutoff will lead to faster histogram accumulation, but will leave unphysical "steps" in the spectrum where lineshapes have been truncated.

`log_file` <*filename*> `[-ascii|-matlab|-double|-append]` causes further output from `echo` and `putmatrix` to be streamed to the specified file. The format is either Matlab (default) or plain ASCII text. In the case of Matlab output, item names are tagged with "*_n*" where *n* is an index incremented from 1, for 1D data sets. The tag "*_r_n*" is used if the output is 2D (i.e. has more than one row), where *r* is the row index (from 1). `log_file` is not compatible with multi-processors simulations, since it is difficult to merge the outputs into a useful single stream. The `double` flag forces full double precision output (single precision is used by default to save space). `log_file` closes any current output stream, and `log_file` without arguments closes the current log and further output is sent to screen. This allows more than one output file to be created, although only one file can be active at a time. The `-append` is used to add logging output to an existing file. This can always include a file created with `save` which is useful for adding comments / information to Matlab format output.

`matrix set` <*name*> `(coherenceorder (`*[*<*indices*>*]* <*coherence orders*>`)*|`<*product operator expr*>`|`**`general`** <*matrix elements*>`|`**`spinorder`** `([`<*indices*>`]` <*spin orders*>`)*)` is used to set up "coherence mask matrices" for subsequent use with `filter`, or to evaluate a product operator expression e.g. for subsequent output with `putmatrix`. <name> is not restricted to being a number (e.g. `matrix set filter90 coherenceorder [-1,1]`). ~~The definition of coherence masks differs from SIMPSON: `totalcoherence` is not supported (as this doesn't have a clear physical significance) and the `coherence` selection is specified with a list of coherences for each RF channel e.g. `coherence [1,-1] [2,0,-2]`.~~ *Note that [] brackets are used here and elsewhere for lists, not {} (as in SIMPSON).*

`general` <*matrix elements*>`*` (V09.03.10) sets up a general square real or complex matrix e.g. for use in `exchange`. Matrix elements are supplied in column order and, for complex matrices, as alternating real and imaginary components e.g. `[0,1,1,0]` and `[0,0,0,-1,0,1,0,0]` would create the following 2 x 2 matrices:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad and \quad \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

The contents of the matrix element vector do not have to be fixed, allowing the matrices to change over a simulation.

`spinorder` <*spin orders*> | <*indices*> <*spin orders*> … (V11.12.21) creates filter matrices as `coherence` but based on spin order rather than coherence, where spin order is defined as the number of spins that change state between the bra and ket of a coherence (note that spin order is currently only defined for pure spin-1/2 spin systems). The spin orders argument determines which spin orders are selected (as a [] list or a single value), while the optional indices argument selects which spins are involved (all spins if argument is omitted) e.g. `spinorder Indicesof('1H') [4,5]` would set up a filter for spin orders 4 and 5 involving coherences between the ^{1}H spins of a system. Further index and spin order lists (V12.03.27) can be used to specify heteronuclear spin orders e.g. `spinorder Indicesof('1H') 5 Indicesof('13C') 1` would select for a spin order of 5 on the ^{1}H spins and 1 on the ^{13}C. A warning is given if the sets of spins overlap as this is likely to be an error.

coherenceorder <*coherence orders*> | <*indices*> <*coherence orders*> … (V13.08.08) works as `spinorder` but selects coherence orders rather than spin orders (and is not restricted to spin-1/2 systems). It replaces the previous flawed definition mechanism using `coherence`. `totalcoherence` can be used, as in SIMPSON, as a synonym for the case when only total coherence orders are being specified.

`maxdt` Δt * sets the "integration time step" when working with time-dependent (i.e. spinning) Hamiltonians (defaults to 1 μs). Like all time quantities, the units are μs.

maxjumpdt τ_{jump} sets the maximum time step when propagating the Hamiltonian using a fast, but more approximate, algorithm for evaluating propagation under simultaneous RF and MAS that are not well synchronised (associated with the `phasemodulation` optimisation). This is disabled by default (i.e. 0) and only enabled by setting a non-zero jump time. τ_{jump} should never be larger than the appropriate Δt for a problem, and it is advisable to verify convergence of results with respect to this parameter. **Take care with this functionality–it can easily give strange results!**

`ni` *n* ***<skip>*** sets the number of t_1 increments in a 2D experiment. t_1 is only incremented every *skip* "scans" (e.g. 2 per hypercomplex acquisition), so the total number of points in the indirect dimension is the product of *n* and *skip*. Note that setting `ni` changes the way the arrays are interpreted (see below).

`np` *n** sets the number of points in the indirect dimension (frequency or time domain propagation). Calculation time will increase with `np` for time-domain calculations (dramatically so if the calculation is "asynchronous"), but not for frequency domain calculations where it simply sets the number of histogram bins.

n<*dimension*> sets the number of points in indirect dimension n (from 1). See later section for details on n-D data sets.

precision *n* [<*m*>] [`-complexpair|-complexi|complexcompact`] uses *n* significant figures in all subsequent floating point output. If the two argument form, the accuracy of general floating point output and of matrix output can be specified individually, with <*m*> setting the accuracy of matrix output. If *m* is zero, then matrices are printed in compressed form with `.` and `X` representing zero and non-zero elements respectively (useful for examining matrix structure in large problems). A value of –1 indicates that the current setting should be preserved. The option argument (V11.12.21) modifies the way complex numbers are displayed: (*R*,*I*) for `complexpair` (default), *R*+i*I* for `complexi` (*R* and *I* always displayed), *R*+i*I* but omitting *R* or *I* if zero for `complexcompact`.

The `NMRSIM_FORMAT` (V10.11.04) environment variable can be used to obtain finer control over the formatting used when converting numeric variables to formatted quantities e.g. when `$spin_rate` is used in a filename. This a C-style formatting string for floating point numbers (see the manual page for the `printf` function) without the leading `%` character e.g. `f` will used a fixed number of figures / decimal places, `g` is the "general" style. If a `*` is present in the string, the precision value is passed as an argument. For example, the default formatting if not overridden by `NMRSIM_FORMAT` is `.*g` which gives numbers in the "general" format to *precision* figures, while a setting of `.4f` would output numbers with fixed number of digits (4).

propagation_method `diagonalise | chebyshev` sets the method used to calculate propagators from time-dependent Hamiltonians between diagonalisation of the Hamiltonian (default), Chebyshev propagation or switching between methods depending on the sparsity of the Hamiltonian. In general, Chebyshev propagation is more efficient when the matrices are sparse, while diagonalisation is more effective for dense matrices.

pseudohistogram behaves like `histogram` in the sense that the NMR signal is calculated via transition amplitudes and frequencies, as opposed to normal "time domain" methods that explicitly propagate the density matrix via propagators, except that the transitions are accumulated into an FID rather than a frequency domain histogram. This mode is often faster than normal time domain propagation but less efficient than histogram. It avoids the rounding issues that can affect direct histogram accumulation, but shares the same issues with numerical stability, hence it is not default behaviour.

`pulse`, `pulseid` are described in the "Defining pulse sequences" section.

`putmatrix` <*name*> outputs a matrix / Hamiltonian. <*name*> can be `start`, `detect`, `hamiltonian`, `density`[10], or the name of a matrix previously created with `set`. If a `log_file` has been defined, the matrix is streamed to the specified file, tagged with the name of matrix (Matlab format only).

`rotor_angle` <*angle*> sets the angle of the rotor axis with respect to the magnetic field (defaults to the magic angle).

sideband <n> restricts the calculation in MAS problems to sideband *n* (where 0 is centreband). This can *often* be used to pick out a specific sideband, but obviously has little physical significance. Note that the distinction between sidebands is not necessarily clear cut if the spin rate is not sufficiently high.

`spin_rate` ν_r * sets the sample spin rate.

`start_operator` <*product operator expr*>* sets the initial density matrix, σ_0 e.g. `start_operator Fx`. If the spin system is homonuclear, `start_operator` can be omitted and defaults to `Fz` (i.e. *z* magnetisation) if any pulses are used in `pulseq` or `Fx` (*x* magnetisation) otherwise. Like `detect_operator`, the product operator expression need not be fixed.

steps_per_rotation <*n*> specifies the timestep used when integration the evolution for time-dependent problems in terms of the number of integration steps per rotor period i.e. the integration timestep is no longer than the rotor period divided by *n*. This is an alternative (for MAS problems only) to setting the integration time step explicitly using `maxdt`.

`sw` <*width*>* sets the direct dimension spectral width. In the case of time domain propagation (the default), the dwell time should be "synchronised" with the other time-dependencies such as RF pulses and sample spinning for maximum efficiency. This restriction does not apply to frequency domain propagation. Note that for time domain propagation, frequencies outside this spectral width are unavoidably aliased (folded) back into the spectrum (whereas they would be removed by filtration on the spectrometer). See `histogram` for information on how out-of-range frequencies are handled in frequency domain propagation.

`sw`<*n*> <*width*> ~~[<*sfrq*>]~~ sets the spectral width for indirect dimensions, `sw1` for the first indirect dimension, `sw2` for the next etc.. The evolution in indirect dimensions is always calculated using time domain propagation of the density matrix. Hence synchronisation is helpful, although the efficiency gains will tend to be less noticeable in comparison to the direct dimension. *The optional sfrq parameter was removed in V11.06.14 in favour of the* `set` *directive in* `proc`.

tolerance <*dt*> sets the timing "tolerance" (default 0.005 i.e. 5 ns). This is in effect the time difference over which two times are taken to be different in pulse sequences. Similarly transient effects "older" than this period are assumed not to have an effect on subsequent evolution. Obviously it is then important not to have pulse sequence elements shorter than (or equal to) this time period, and event times should generally be much larger. This can be used to overcome synchronisation problems due to rounding errors.

Inserting a delay greater than <*dt*> is also a useful way to tidy up any lingering phase transients.

`variable` is used to define new variables (see below).

`verbose` **`-optimise -general -parse -powder -profile`** Aspects to be reported on are specified by flags referring to: fitting/optimisation, general operation, parsing (only useful to debugging), powder orientation. If no flags are given, all verbose output is enabled. `-profile` enables basic profiling. At the end of the calculation, the time spent in the different steps of `pulseq` and processing are displayed together with memory usage information from different levels.

The following SIMPSON functions are either redundant or not implemented in pNMRsim: `method`, `pulse_sequence`, `offset`, `select`, `turnoff`, `turnon`, `getinteractions`.

Defining pulse sequences

In pNMRsim, RF pulses and delay periods are assembled in the `par` block into named "sequence fragments"[11]. These can then be "applied" in `pulseq` e.g. as a preparation sequence (prior to acquisition) or during acquisition (e.g. for decoupling). The density matrix is only defined at the beginning and end of a fragment i.e. `filter` elements cannot be applied as part of a sequence fragment. "Asynchronous" programming of the RF channels is supported, using the `channel` directive to limit subsequent `pulse` and `delay` commands to the specified channel.

The following `par` block functions are used to define sequence fragments

`channel` *n* future `pulse` and `delay` commands are restricted to RF channel *n* (numbered from 1, in the order in which they were declared in `channels`). `channel` cannot be used part way through defining a synchronous fragment, since asynchronous and synchronous elements cannot be mixed.

`delay` <*dt*>* inserts a delay of *dt* μs into the sequence.

`pulse, pulseid` <*pw*>* (<*amp*>* <*phase*>* [<*offset*>* `[-coherent]`] [`-transients`])+ add RF pulses. *pw* is the nominal duration, *amp* is the RF amplitude (expressed as a nutation rate), *phase* is the RF phase and *offset* is the offset from the transmitter frequency. An *amp, phase, offset* triple must be supplied for each active RF channel in synchronous mode (order as defined in `channels`). Offsets must be either supplied or omitted for all channels, and non-zero offsets are only meaningful for soft pulses. As intuitively expected, setting the offset to match the shift of a peak will put it on-resonance[12] (V11.06.14). The `-coherent` flag indicates that the phase change *following* an off-resonance pulse should be coherent with respect to the final phase, e.g. as in Frequency-Switched Lee-Goldberg decoupling[13]. The `-transients` flag (soft pulses only) specifies that phase-transients should be applied (according to the transient amplitude set in `transients`).

`pulseid` is used for hard (ideal) pulses. In this case *pw* is only used to determine the effective tip angle, and the effective duration of pulse appears to be zero. The "synchronisation" of `pulseid`'s can be flagged using the optional sync parameter (`pulseid` *[<sync>]* <*pw*>…), where `` `+' `` ("rising event") indicates that it belongs to a new sequence fragment, `` `-' `` ("falling event") that it finishes a fragment, and `` `|' `` (default) gives no sychronisation hint. These hints are important if propagator evaluation starts or finishes on a hard pulse.

Unlike most other commands, the pulse commands accept list quantities, which are expanded to a series of pulses e.g. `pulse 2 50e3 [0,90,180,270]` is equivalent to

```
pulse 2 50e3 0
pulse 2 50e3 90
pulse 2 50e3 180
pulse 2 50e3 270
```

This makes writing shaped / ramped pulses more straightforward. The lists must be the same size, and the duration may be also be specified as a list of V11.09.29.

Although it is usually more efficient to use sequence fragments multiple times in the `acq` block (using `prop` <*seq*> <*n*>), it is sometimes necessary to construct a complete new sequence fragment e.g. for use as an acquisition decoupling sequence. As an alternative to using lists and a single `pulse`, this can also be done by defining the repeated fragment as a block and then using `include`[14] e.g.

```
XiX {
 pulse $pwdec 100e3 0
 pulse $pwdec 100e3 180
}
…
include XiX [1:3]
```

will include XiX three times (the [1:3] creates a dummy arrayed argument). Note that the final evaluation of pulse sequences is independent of the mechanism used to create them.

`store` (<*fragname*>*[*<*sync time*>*[*<*sync hint*>*]]*|<***listname***> <***fraglist***>) stores the current sequence fragment under the supplied name (not restricted to a number). A `store` finishes the current sequence[15] and resets the mode to synchronous. The total duration on the (active) independent channels must be the same before an asynchronous fragment can be stored (use `delay` to pad sequences to the same length if necessary). The optional synchronisation time is useful in MAS problems to establish the period over which the MAS period and RF period can be matched when this is not obvious. This can greatly speed up calculations, especially for phase modulated RF. The `sync_ratio` function (see below) can be used to find a suitable synchronisation.

A zero synchronisation time corresponds to no suggested synchronisation. The optional "synchronisation hint" (V08.03.08) can be useful for problems involving magic-angle spinning. It specifies a "helpful" division of the rotor period for "caching" of propagators. If, for instance, the value 5 is given, then 5 "cache slots" over the rotor period will be created and subsequent requests for propagators at intervals of 72° will not require recalculation.

`store` <*listname*> <*fraglist*> associates a name with a cyclic list of sequence fragments (see `prop` in the `pulseq` block for more details). Such lists are not interchangeable with sequence fragments, but the names must be unique to avoid confusion.

pulseq

This block defines how the pulse sequence fragments defined in `par` are to be used. In effect, the "actions" in this block are applied successively to the initial density matrix to generate the NMR signal for a given orientation. Valid commands in this block are

`acq` [<*phase*>*] [(<*seqname*>|-)[<*synctime*>]] [`-putmatrix`] acquires the FID with a given receiver phase shift (defaults to 0) and optional "acquisition sequence" i.e. RF applied during signal acquisition[16]. The phase parameter is omitted for indirect dimensions (see below). `acq` can be used after an `acqn` (direct dimension only), in which case it fills the remaining points in the data set (V09.03.10).

The optional "synchronisation time" can be used to help pNMRsim find a synchronisation interval when this can't be automatically deduced (i.e. the different time periods are not simple multiples of each other). For instance, if the rotor period is 30 μs and the dwell time is 20 μs, passing a synchronisation time of 60 μs would allow pNMRsim to calculate the evolution over just 60 μs rather than accumulating the signal "point by point"[17]. (see the discussion of synchronisation of pulse sequences).

The synchronisation interval can also be specified when RF is not active (i.e. when the dwell time and rotation period are not obviously synchronised) using – to denote no acquisition RF. The supplied valued overrides any synchronisation period (for the MAS and RF) defined with the sequence, which is otherwise used by default.

Acquisition sequences must normally have non-zero duration, however, sequences consisting purely of hard pulses are also accepted provided the relevant `sw` parameter is unset. In this case, data points are acquired after successive applications of the sequence (and the spectral width is meaningless).

The optional `putmatrix` flag (indirect dimensions only) outputs the propagator(s) used to the current output stream. (There is otherwise no mechanism to output propagators since these do not have an independent existence in pNMRsim). The propagator can be used to determine an "effective Hamiltonian" for the propagation via the matrix logarithm (but the result is not unique).

`acqn` <*points*> [<*phase*>*] [(<*seqname*>|-)[<*synctime*>]] [`-putmatrix`] (V09.03.10) behaves like `acq`, but acquires a fixed number of data points rather than filling the data row. Multiple `acqn`'s can be used for a single data row, but it is only valid for directly acquired dimensions. It should only be used if a data acquisition consists of distinct unrelated segments; most cases of "point by point" acquisition can be more efficiently expressed by acquisition in the presence of an RF sequence. The total number of data points acquired cannot exceed `np`. Unfilled data points will be zero.

`acqpoint` <*phase*> (V09.03.10) acquires a single data point *in the directly acquired dimension* using the current density matrix and detection operator. There is no propagation of the density matrix and the time is not incremented. Again, for reasons of efficiency, this should only be used if it is not possible to use the normal `acq`.

`echo` <*text*> outputs text (see above).

`exchange` <*operators*>+ <*matrix name*> (V09.03.10) allows components of the density matrix to be arbitrarily shuffled. The projections of the density matrix onto a set of operators are calculated and the named transformation matrix (created by `matrix set` <*name*> `general` in `par`) is applied to these coefficients and these new coefficients as the weighting factors for the operators in the new density matrix e.g.

```
matrix set swap general [0,1,1,0]
…
exchange I1x I2x swap
```

would exchange the coefficients of I1x and I2x in the density matrix i.e. exchange *x* magnetisation on spins 1 and 2. `transfer` can be regarded is a special case of `exchange` with the matrix `[0,0,1,0]` i.e. the component of the first operator is fully transferred to the second operator. Note that the exchange matrix may be complex, allowing phases to be altered.

`do` <*n*> … **`end do`** (V12.03.27) repeats *n* times the actions contained between the `do` and `end do` lines. The loop is skipped if *n* is zero. For example

```
do $np
   acqpoint
   prop myprop
   filter myfilter
```

```
end do
```

would effectively acquire a FID in which a filter was applied to the density matrix after each propagation step. *Note that this will be much slower than obtaining the FID with the normal* `acq` *directive, and such loops should be avoided if possible as they considerably reduce the scope for optimisation.*

`filter` <*mask*> applies the previously-defined coherence mask to the density matrix.

`get` <*variable name*> <*operator*> *[*<*source*>*]* creates a new variable whose value is the trace of a matrix (by default the density matrix, `density`) with the specified operator previously defined in `par` (real component only). If for instance, `matrix set detectH 1H:x` has been used in `par` to create the operator matrix `detectH`, then `get signal detectH` can be used to determine the current ^{1}H *x* magnetisation for subsequent output via `echo`. Note that the variable name must be a new one since you are not allowed to change the definitions of quantities during a simulation[18].

`prop` <*pseq*> [*n**] [`-putmatrix|-reset`] applies a sequence fragment or list of fragments sequence *n* times to the density matrix (or does nothing if *n* is zero). The optional `putmatrix` flag outputs the overall propagator to the current output stream. <*pseq*> refers to a previously-defined pulse sequence fragment, plus an optional phase shift, using the syntax <*fragment*>+<*shift*>. The phase shift is applied to all RF channels once the fragment propagator has been calculated, which is often simpler and more efficient than phase-shifting the individual elements of the sequence fragment. If these "phased sequence" fragments are included inside an {} array, then the elements of the array are stepped through in parallel with other arrayed quantities e.g. `{pulse90+0,pulse90+180}`. In principle the pulse sequence fragments in a list need not have the same duration, although a warning is given if this is not the case since it probably indicates an error (V11.03.06). `[]` denotes of list of fragments, whose members are used in turn each time a <*pseq*> is used. The pointers are reset whenever the sequence is changed. `store`'ing a shared fragment list as a named sequence allows the value of the list pointer to be preserved between different parts of a sequence e.g.

```
[in par]
store REDORxy8 [REDORxy,REDORyx,REDORXY,REDORYX]
[in pulseq]
prop REDORxy8 2
prop REDORmiddle
prop REDORxy8 2
```

will apply `REDORxy,REDORyx,REDORmiddle,REDORXY,REDORYX`. The `-reset` flag (V11.03.06) resets the list pointer before propagation i.e.

```
prop REDORxy8 2
prop REDORmiddle
prop REDORxy8 2 -reset
```

would apply `REDORxy,REDORyx,REDORmiddle,REDORxy,REDORyx`.

`[]` lists can be nested inside {} lists (changed in V11.03.06). Note that || (sum) arrays cannot be used (changed in V11.03.06) since phase cycling etc. is generally best performed using the phase shift parameter. `prop` has some associated optimisations (see `combinepropagators` and `smartprop`). These generally can only be applied in MAS simulations when overall duration of the fragment set (i.e. the time over which it repeats) is synchronised with the spinning period.

`propfor` <*time*> <*pseq*> [`-putmatrix|-reset`] works likes `prop`, but rather than applying a sequence fragment a fixed number of times, it applies it for a specified time (in μs). This is useful, for example, when the first point of an acquisition is at a non-zero time; `propfor` can be used to fill the gap up to the first data point (which is always acquired at time zero). `propfor` is designed to be used with a single, "continuously

running" sequence and so cannot be used with `[]` lists. Rather a repeat use of `propfor` with the same sequence will pick up where the previous finished, unless the `-reset` flag is used to force propagation to restart from the beginning of the pulse sequence. For example:

```
propfor 50 tppm
pulse180 13C x
propfor 50 tppm [-reset]
```

might correspond to a spin-echo experiment on C with TPPM decoupling on H and a delay periods of 50 μs. Without the `-reset`, the decoupling would, in effect, run continuously either side of the echo pulse. With the `-reset`, the TPPM would start from the beginning after the echo pulse. The former is more likely to match the experimental implementation, but the latter formulation increases the likelihood that the propagator from the first period can be re-used for the second. The current implementation of `propfor` (V11.03.06) applies the `smartprop` and `combinepropagators` optimisations where possible, and so should be equally as efficient as the corresponding `prop`.

`pulse180` *[<nucleus>]* *<phase>** (V09.12.02) applies a hard (perfect) inversion pulse of given phase to the selected nucleus (`1H`, `13C` etc.). The nucleus may be omitted in homonuclear systems. Unlike other RF pulses in general, a perfect inversion pulse does not disrupt the block structure of the free precession Hamiltonian. Hence if RF is only required on a given nucleus for inversion pulses (e.g. a simple spin echo) it is then possible to use `pulse180` without including the associated nucleus in the RF channels declaration, with a considerable gain of calculation efficiency.

`putmatrix` *<name>* [`-full|-eigenbasis|-structure|-statistics` [*<indices>* [*<filter matrix>*]] [`-once`] displays the current contents of a matrix (see above). If the optional `-full` flag (V09.12.02) is specified then the block structure of operator matrices (only) is expanded (to facilitate comparison between simulations with different block structure).

Additional output types (V13.06.13): if `-structure` is specified, the dimensions of the blocks in the Hamiltonian / matrix are output. `-eigenbasis` outputs information on the eigenbasis. This takes the form of an ordered list of the states in each diagonal block of the eigenbasis, and, for operator matrices, this lists the index of the eigenbasis block (numbered from 1) for the row (bra) and column (ket) states of each block. In other words, combining the eigenbasis information from an operator with the Hamiltonian allows the states associated with each element of the operator matrix to be established. `-statistics` outputs statistics on operator matrices (most usefully for the density matrix). This currently breaks down by both coherence order and spin order the total number of matrix elements involved, the number of non-zero elements, and the (complex) sum and norm (root-sum-square) of the elements. The numerical threshold for deciding whether a matrix element is non-zero can be overridden with the environment variable NMRSIM_COHERENCE_TOLERANCE. In the absence of a log file, output is displayed in human-readable form. Log file output (ASCII or Matlab) contains the same information but in a form more suited for machine parsing / additional processing. (EXPERIMENTAL: subject to change) The optional arguments for `statistics` can be used to restrict the spins for which the statistics are calculated, by specifying the indices of the spins of interest e.g. `[1,2]` or `Indicesof('1H')`, and, as a further option, selecting a subset of coherences using a previously filter matrix (V13.08.08) e.g. `matrix set Cplus1 Indicesof('13C') 1` [in `par`] `putmatrix density -statistics Indicesof() Cplus1` [in `pulseq`].

If the `-once` flag (V13.06.13) is specified, the information will be output on the first run through only (e.g. the first orientation of a powder average).

rotor_angle <*angle*>* allows the rotor angle to be changed during a pulse sequence (it is reset to the default supplied in `par` at the start of `pulseq`).

scale <*factor*>* *[<filter matrix>]* (extended V11.12.11) scales the density matrix by the supplied factor. This allows different contributions to a summed spectrum to have a different weight (`scale` in the `proc` is only applied to the summed spectrum and so can't be used for this purpose). If the optional filter matrix is supplied the only the density matrix elements corresponding to the non-zero elements of the filter are scaled. This is typically used to damp terms by spin order and a warning is given if `scale` is used with a coherence-based filter.

timeadjust <*time*>* `[-absolute]` (V09.03.10) adjusts the current time counter (which starts at zero at the beginning of each run through the `pulseq` block). The time adjustment (in microseconds) is relative to the current time, unless the `-absolute` flag is given. Note that there is no effect on the density matrix; this simply allows time adjustments if actions such as `transfer` or `exchange` have been used to "artificially" manipulate the density matrix. In the context of MAS experiments, it may be necessary to adjust the time in this way to maintain rotor synchronisation. A warning is given if `timeadjust` is used in a static simulation since it has strictly no effect.

transfer <*from*> <*to*> Takes the trace of the density matrix with the operator <*from*> and replaces the density matrices with operator <*to*> scaled by this trace. The operators can either be defined in the par block (essential if they are not constant expressions) or specified directly as operator expressions (V09.01.03). Hence

```
matrix set 1Hx 1H:x
…
transfer 1Hx 13C:x
```

can be used as an ideal cross-polarisation i.e. converting the ^{1}H *x* magnetisation (operator defined in `par`) into ^{13}C *x* magnetisation (given as operator expression).

If {} arrays are being used to create data sets with multiple (independent) rows, then multiple `pulseq` blocks can be specified, one per row of the data set. This allows different pulse sequences to be applied to different rows. Alternatively, but less flexibly, a single `pulseq` block can be used, but the parameters varied using {} arrays.

proc

The `proc` block defines the processing that is applied to calculated FID or spectrum after any summation / powder averaging[19]. The individual "actions" are applied successively to the complete data set (which may be two-dimensional). A limited number of commands only apply e.g. `ft2d` (or function differently) for "true" 2D data sets i.e. `sw1` has been defined.

Processing can be applied to individual transients before any summation (over the powder and/or || summation arrays) by specifying processing commands in an `initialproc` block placed before `proc`. This is obviously significantly less efficient than processing a single summed signal and is only useful if different processing e.g. line-broadening needs to be applied to different signals (differing signal intensities can be handled in other ways e.g. `scale` in `pulseq`). By default (and if the `mergeprocessing` optimisation has not been disabled) the contents of `initialproc` will be merged into the front of `proc` in the interests of efficiency (V12.08.02).

As above, multiple processing blocks can be specified for data sets containing independent rows.

Note that with the exception of `addsignals`, most processing commands are specified in terms of raw data points rather than frequencies. Hence the "reference frequency" is not

relevant. The only other time that the reference frequency is used is with the `-scale` option of `save`. Internally data is stored in the "natural" order used by SIMPSON. Time domain data starting from $t = 0$, with zero frequency corresponding to no time dependence. Frequency domain data is stored in order of increasing frequency, with zero frequency assumed to correspond to the midpoint (by default the data is shifted by half and spectral width as part of Fourier transformation). Note that the "reference frequency", which sets the frequency of the spectral midpoint, is rarely required and is only directly visible if a frequency scale is constructed using the `-scale` option of `save`. By default data is also saved in this order (which is specified by the SIMPSON file format). Some processing programs, however, store data in "display order", which is in the opposite direction by convention for frequency domain data. The `-reversefrequency` flags (V15.08.13) can be used when reading or writing data to account for this. This is generally less confusing than using `rev` to fix up such issues.

Most processing parameters are fixed when the command is created, but others (marked with *) can be variable i.e. can be arrayed or used as fitting parameters.

`addlb` <*LW*>* [<*r*>* [<*LW1*>* [<*r1*>*]]] a line-broadening equivalent to <*LW*> Hz is applied to the FID. *r* is a Gaussian/Lorentzian factor (1 for pure Gaussian, 0 for pure Lorentzian–default). The optional parameters give the line-broadening and *r* factor for the indirect dimension. `addlb` can also be applied in the frequency domain (1D variant only), but note that the definition of the Gaussian/Lorentzian lineshape is subtly different[20]–lineshapes of the same width <*LW*> are added but in proportion specified by <*r*>.

`addnoise` <*stddev*>* adds Gaussian noise of given standard deviation to the data set.

`addsignals` <*frequencies*>* <*amplitudes*>* [<*phases*>*] adds frequencies directly to a time domain or frequency domain signal. Multiple signals can be specified using `[]` lists e.g. `addsignals [-1e3,1e3] [0.2,0.4] 45` adds two signals, of phase 45 degrees: intensity 0.2 at –1000 Hz and intensity 0.4 at 1000 Hz. Note that when used in the frequency domain, `addsignals` is one of the few commands to refer to any reference frequency. (Clarified in V15.08.13.)

`apply` <*function*> `[`<*arg number*> <*arguments*>`+] [-real|-imag|-complex|-complexpair]` (V09.03.10) applies a function to the data set. If the function takes more than one argument, the remaining arguments must be supplied in order along with an argument number indicating which argument the data set corresponds to. If the data set is multi-dimensional (`ni` set), the data is passed as a single data vector, otherwise each row is processed independently. By default (`-complex`) the function is applied to both real and imaginary components independently. Alternatively, the function can be applied to either the real or imaginary components; in this case, the function cannot change the number of data points. If `-complexpair` is specified, then the function is applied to a vector of "complex pairs" i.e. alternating real and imaginary components. This potentially allows complex arithmetic to be applied. Since the function may can external routines (via ``), potentially arbitrary transformations of the data are permitted.

Examples:

`apply sin -real` would replace the real component with the sin of each point.

`apply head 1 10` would delete all but the first 10 data points

`conjugate` Takes the complex conjugate of the data set. This can be applied to either FID or spectrum, although it is most useful for the former (having the effect of reversing the spectrum after Fourier transformation). This is applied automatically if the detection nucleus has a negative gyromagnetic ratio (but can be reversed by applying an additional `conjugate` step prior to `ft`).

`extract` <*indices*> *[<row indices>]* reduces the data set to a subset of the original rows and columns based on the sets of indices supplied e.g. `extract [30:40]` selects points

(or whole columns of a 2D data set) 30 to 40 (inclusive), `extract [30:2:32,40:2:42]` selects points 30, 32, 40 and 42 etc.[21] For 2D data, the first set of indices refers to the direct dimension (columns) and the second set to the rows. NB. the spectral width will generally be meaningless after an `extract`. An empty set of indices, `[]`, refers to the complete row / column (V09.11.10) e.g. `extract [] [1:3]` returns the first three rows of the data set.

`fill` *<value> [<indices> [<row indices>]]* `[-real|-imag|-complex]` sets all or part of the data set to a defined value. The indices (and optional row indices for nD data sets) are used to define a subset of the data. The `-real`, `-imag` or `-complex` (default) flags specifies whether the real component, the imaginary component or both are set e.g. `fill 0 -imag` sets the imaginary component of the data to zero, `fill 0 1` sets the first point of the data set to zero etc.

`ft [-inv|-noscalefirst|-noshift]` 1D Fourier transform (or inverse transform). By default the first point is scaled by 0.5 to ensure a zero baseline for a normal FID and the output is shifted so that the zero frequency is in the middle of the spectrum. These steps are not strictly part of the FT, but follow SIMPSON usage. The `-noscalefirst` option turns this off, and the `scalefirst` command can be used for more control of the first point scaling. Similarly `-noshift` turns off the data shift. The FT is most efficient if the number of points is a power of 2 (so the fast FT can be used)–a warning is given if this is not the case.

`ft2d` `[-noscalefirst|-noshift]` *[<rp> <lp> <rp1> <lp1>]* 2D Fourier transform. Note the different name to distinguish 1D and 2D transforms. A hypercomplex transformation is applied if the "skip" for the indirect dimension is 2 i.e. alternating "sine" and "cosine" FIDs, otherwise a "phase-modulated" FT is applied (which will normally lead to phase-twist lineshapes). The phase parameters can be omitted to perform the transformation without phase correction. Note, however, that a phase correction in t_1 cannot be applied as a separate step as the necessary hypercomplex data has been discarded. `ft2d` cannot be applied to data sets already in the frequency domain. `-noscalefirst` and `-noshift` turn off the first point scaling and data shifting respectively that are applied by default.

`normalise` `[`*<norm>*`] [-integral|-minmax|-abs|-area]` normalises the data set to a specified value (1.0 by default) using either the integral (unscaled sum) of the real component (default), the maximum or minimum of the real component, the maximum absolute value of the complex data points (V09.01.03), or the "area" (V10.11.14) of signals (sum multiplied by spectral width). The data set is considered as a whole for nD data (i.e. `ni` set), otherwise rows are normalised independently.

`offset` *<value>*`|`*<list>* `[-real|-imag|-complex]` adds a constant value to the real (default), imaginary or both real and imaginary components of the data set. Alternatively a list of the same size as the data set can be added.

`phase` *<rp>* <lp>** `[`*<pivot>*`]` Phase adjust spectrum (in one dimension, cf. `ft2d` for 2D phasing). The *<pivot>* value specifies the origin for the first-order order ("right") phase as a fraction of the spectrum i.e. 0 corresponds to one end, while a pivot of 0.5 sets the origin to the middle of the spectrum. Normally the spectrum has been shifted post Fourier transform so that zero is in the middle (and so a pivot of 0.5 is most appropriate). Phase can be applied to time-domain data (useful for "shearing" data sets), and here the pivot should be 0. Defaults of 0 (time domain) and 0.5 (frequency domain) are used if this parameter is omitted.

`resample` *<newnp>* `[-fold|`*<sw>* `[`*<offset>*`]]` Resamples the data set (direct dimension only) to contain *newnp* data points (using cubic interpolation). If the `-fold` option is specified then the data assumed to "fold round" and peaks that have been split across the ends of the spectrum will be treated correctly. In contrast, if the spectrum is treated in a simple linear fashion then the frequency range can also be changed by

specifying a new spectral width and optionally a signed offset giving the centre of the new spectrum with respect to the centre of the current spectrum.

`resample` is particularly useful for adjusting the frequency to match experimental data which is unlikely to have been sampled with the same synchronisation conditions that are convenient for efficient simulation. Note that resampling a time domain data set is likely to give poor results and is not permitted.

`rev` The order of data points in the spectrum is reversed. This is applied automatically to the results of frequency-domain calculations when the detection nucleus has a negative gyromagnetic ratio.

`save` <*fname*> `[`<*variables*>`]` `[`**`-simpson|-simplot|-matlab|-ascii`**`]` `[-nodata|-projection|-sum|-scale|-statistics|-parameters|-source]` saves the data set in SIMPSON, Matlab or a raw ASCII format. SIMPLOT doesn't read the SIMPSON 2D format, and so 2D/arrayed data sets can written out as a series of rows using `-simplot`. If *fname* is of the form `<base>.XXX` then the filenames are `<base>00.XXX`, `<base>01.XXX` etc., otherwise the series begins `<fname>00`. The number of digits varies between 1 and 3 depending on the maximum number of rows in the data set (V09.07.02). These can then be loaded and overlaid in SIMPLOT e.g. `simplot <base>??.spe`. A sum spectrum and variable array list is also calculated and output. In MATLAB format, these data sets are incorporated in the output file, otherwise they are written as separate <*fname*>`_sum` and <*fname*>`_scale` sets (`-ascii` only). 2D data with `np = 1` are treated as 1D data sets when saving in `simplot` and `ascii` formats. NB. `save` should not normally be used in `proc` when `[]` arrays are being used or when fitting/optimising (see below).

Optionally the contents of (numeric) $ variables can also be saved e.g. `save output final_chisquared` would store the value of the variable containing the final value of χ^2 from a fitting. This is most useful for Matlab format output as the values can be stored in a single file rather than creating a set of individual files for each output. Note that the variable *name* is used rather than its *value* (`$final_chisquared`). Note that (unless `-nodata` is specified), the list of spectral widths, `sw` (frequency domain data) or dwell times, `dt` (time domain) is automatically saved in the Matlab format.

The remaining flags control what outputs are saved. In addition to the main data set, additional data can be saved e.g. sum spectrum, projections etc. These sets are stored in the MATLAB file if this format is being used, with the appropriate name (e.g. `projection`), otherwise a separate file is created with the identity of the data set appended to the filename e.g. `mysim_projection`. Real data sets (e.g. axis scales) cannot be saved in the SIMPSON format; use ASCII instead. `-nodata` suppresses saving of the data, so different formats can be used for data and "supplementary information" e.g.

```
save myfile -simpson
save myfile -ascii -scale -nodata
```

The options are

`-scale`	Time or frequency domain scales in the direct dimension (`scale`) plus indirect dimensions (`scale`<*n*>) if present.
`-sum`	Sum (`sum`) of rows (regular nD data only)
`-projection`	1D (sum) `projection` across columns (2D data only)
`-statistics`	Fitting statistics (`covariance` for the covariance matrix and `residuals` for the fitting residuals)
`-parameters`	Values (`parameters`) of arrayed variables in each row. If

	the save occurs after fitting statistics have been calculated (i.e. within the `finalise` block), then the fitted parameters + errors are saved instead. The variable/parameter names are either listed in the parameters file when fitting parameters are being saved (V12.08.02) or in a separate "file" (`parameternames`) when arrayed variables are being saved or the MATLAB format is being used.
`-source`	Incorporate (where possible) the contents of the input file as comment lines in the output file together with the command line arguments.
`-reversefrequency`	(V15.08.13) Reverse order of output for frequency domain data (see above). Will also reverse frequency scale if `-scale` used in same directive.
`-original`	(V15.08.13) Save original data masked by any fitting mask.

`scale` <*factor*>* The data is multiplied by the given factor.

`scalefirst` <*factor*>* [<*factor1*>*] scales the first point (of the FID) by the given factor (usually 0.5). The optional second parameter applies a scaling in t_1.

`shifthalf` shifts the data set by half its length, swapping zero frequency between the centre and the edge of the spectrum. The shift is performed in both dimensions for true 2D data.

`set` [`-sw|-sw1|-ref|-ref1|-sfrq|-sfrq1` <*value*>*] (V11.06.14) allows data set parameters, such as the spectral width, frequency origin, and spectrometer frequency in direct or indirect dimensions to be set / overridden. The `ref` values specify the frequency of the middle of the spectrum (by default zero), while the `sfrq` values provide the notional spectrometer frequency for the dimension (this can be deduced automatically from `detect_operator` for direct dimensions). The ppm notation can be used to evaluate NMR frequencies if `proton_frequency` has been set e.g. `1e6p1` would return the NMR frequency (in Hz) of nucleus 1. The `ref` and `sfrq` parameters are saved with some file formats (SIMPSON, Matlab) but are not relevant to the simulation itself and other processing with a few exceptions for `ref`. `setdomain` should be used rather than the SIMPSON usage of `set -type`.

`setdomain` [<*dimension*>] [`-time|-frequency|-switch|-States|-noStates`] (V09.08.26) allows information to be changed explicitly for a specified dimension (numbered from 1 up to the acquisition dimension). The acquisition dimensioned is the default. `-time` and `-frequency` set the domain to time or frequency respectively, while `-switch` swaps between the two. These are useful if using a user-defined or external processing function that does not update dimension information. `-States` flags that an indirect dimension consists of alternating cos and sin FIDs, rather a simple series of (phase modulated) FIDs. `-noStates` flags the reverse. This information is important for several 2D processing commands. These flags are only valid for time domain data in indirect dimensions. If no flags are specified, the current dimension information is output for the specified dimension or all dimensions if none is specified.

`transpose` (V11.06.14) swaps direct and (first) indirect dimensions. `transpose` can only be applied to nD data sets or ones in which each row has the same number of data points and other processing characteristics. It is useful for converting a series of single-point data sets into a one-dimensional data row.

`zerofill` <*points*> [<*points1*>] fills the FID up to the specified number of points with zeroes. If <*points*> is less than the current number of points, this is taken as a zero-filling

factor i.e. the length of the FID is multiplied by the given factor, increased, if required up to the nearest power of 2 (V09.05.01) and padded with zeroes (although excessive filling factors generate an error). The optional parameter specifies the zero-filling in t_1.

`finalise`

The optional `finalise` block contains instructions to be executed before pNMRsim terminates. Only a limited number of instructions are valid here: `echo`, `log_file` and `save`, `fit statistics` (V08.03.01). This block is most useful for optimisation problems for instructions to save the optimised results, parameters etc. (instructions placed in `proc` will be executed multiple times). Note that `fit statistics` updates the internal variables associated with the `-parameters` and `-covariance` options for `save`. It can be used whenever fitting parameters have been defined, even if no fitting was actually performed.

Product operator expressions

Instructions such as `start_operator` take product operator expressions as arguments. These consist of weighted sums of individual product operator terms[22], which themselves consist of products of individual spin operators:

`I`<*number*>`<`*op*`>` for operator <*op*>, one of `x`, `y`, `p` (or +), `m` (or −), `c`, of spin <*number*> e.g. `I1x`, or `F`<*op*> for the sum operator <*op*>. `c` to refers to the central transition of half-integer quadrupoles

`In`<*op*> is the SIMPSON equivalent to `F`<*op*>

`I`<*number*>`:`*m*`,`*n*, `F`*m*`,`*n* for the single transition operator between *m*–*n* for an individual spin or a sum operator respectively e.g. `I2:2,1`. The indices refer to the Zeeman levels indexed from 1 (up to 2*I*+1).

<*nucleus*>*:*<*op*>, <*nucleus*>:*m*,*n* are sum operators or sum single transition operators for a given nucleus type e.g. `13C:x`, `2H:1,2`

`C`<*channel*>*:*<*op*>, `C`<*channel*>:*m*,*n* are sum operators or sum single transition operators for a given RF channel e.g. `C1:y`. These are useful for specifying sum operators in heteronuclear systems without referring to specific nuclei.

Each operator can be proceeded by a real or imaginary scaling factor e.g. `2*` or `3*i*`. Note that pNMRsim only understands real numbers so it is not possible to use general complex expressions.

Examples

Simple expressions: `I1x`, `Fy`, `C1p`, `13C:x`

Product operators: `2*I1x*I2x`, `0.82*I1p*I2+*I3-` (note that only `I` terms can be used when multiple operators are present)

Full product operator expressions: `cos(45)*Fx + sin(45)*i*Fy`, `I1p*I2m + I1m*I2p`

N.B. Prefer simple expressions where possible; optimisations involving the block structure of the matrix representations will not be applied if the block structure of the operators cannot be readily deduced.

2D and arrays

pNMRsim has been designed to make producing multi-dimensional spectra or series of 1D spectra as painless as possible. Setting `ni` (or `n1`) in the `par` block flags to pNMRsim that

the output is a two-dimensional spectrum, and it will then expect two `acq` commands in `pulseq` i.e. in general

`[prop|filter]*`: *preparation*

`acq` [<*seq*>]: *evolution*

`[prop|filter]*`: *mixing*

`acq` [<*phase*>] [<*prop*>]: *acquisition (this last* `acq` *can be omitted)*

Note that the indirect dimension `acq` does not take a receiver phase shift parameter.

The indirect dimension dwell time is determined from `sw1` and t_1 is incremented from 0 in steps of the dwell time, every *skip* runs through the acquisition.

Phase cycling in the indirect dimension e.g. for phase sensitive detection, is implemented by passing a set of sequence fragments in the `prop`'s: e.g. using `prop {prepCos,prepSin}`, the density matrix would be propagated by `prepCos` and `prepSin` on alternate t_1 increments. Alternatively, propagator phase shifts can be used e.g. `prop {prep,90+prep}`. This formulation keeps the structure clear and aids optimisation e.g. the propagator `prep` would only be calculated once.

Multi-dimensional data sets are created by setting the variables `n1` (1st indirect dimensional; equivalent to `ni`), `n2` (2nd indirect dimension) etc.[23] e.g. `n1 10`, `n2 20` creates a 3D data set with 10 increments in the first indirect dimension and 20 in the second. 2 or 3 `acq` commands would be expected (depending on whether the direct dimension `acq` is explicit).

Arraying variables is another way of creating nD data sets. In most cases, any quantity can be "arrayed" by writing its value as a list e.g. `delay {5,10}` (see below for more details). The calculation would be repeated twice with the 2 rows of the final data set corresponding to simulations with delay of 5, and 10 μs. Any number of quantities can be arrayed in this way. Multiple lists are always stepped through in synchrony e.g. combined with `pulse 5 {0,90,180,270}`, the output would have 4 rows corresponding to

delay 5 μs, phase 0
delay 10 μs, phase 90
delay 5 μs, phase 180
delay 10 μs, phase 270

Different arrays must have "compatible" lengths i.e. be integers multiples of each other, with the longest array setting the number of rows in the output. As illustrated above, shorter arrays will be "cycled" through until the longest array is exhausted. There is no facility (at the moment) for nesting arrays to create multidimensional data sets.

If a data set is multi-dimensional (i.e. `ni`/`n1` has been set), then the calculated data set must be rectangular i.e. the number of points must be the same in each row. If `ni` is unset, however, then the parameters of different rows can be completely independent, *including the number of data points*, `np`. This is particularly useful for simultaneous fitting of multiple data sets. In this case, however, some data processing commands e.g. 2D Fourier transformation will be unavailable.

The two approaches can be combined, provided the number of sizes of the arrays and the `ni` are commensurate. The maximum array length must be one of:

The `ni` increment, in which case the array is stepped through with each 2D row (and restarted when for each step of the indirect dimension dwell time).

`ni`, in which case the array is only stepped when the indirect dimension dwell time is incremented (and is not cycled)

The total number of 2D rows i.e. each element is used once for the corresponding row.

For instance

```
variable N 32
transients {0:0.05/($N-1):0.05}
…
ni $N
```

would cause the phase transient parameter to be increased from zero to 0.05 in 32 steps over the course of a 2D simulation.

Arrayed variables can be nested by tagging with a virtual dimension; `:`<*m*> means that the variable will be varied over dimension *m* only. The dimension sizes are deduced automatically from the largest run (unless the data set is multi-dimensional, in which case the dimension sizes are set explicitly) e.g.

```
variable foo {1:4}:1
variable bar {2,3}:2
variable bar2 {3,4,5,6}:2
variable foobar {1:8}
```

Assuming indirect dimensions had not been specified, two virtual dimensions would be created: dimension 1 of size 4 (from `foo`), and dimension 2 of size 4 (set of `bar2`). The total number of rows would then be 16, and the variables would take on the values in the table:

foo	bar	bar2	foobar
1	2	3	1
2	2	3	2
3	2	3	3
4	2	3	4
1	3	4	5
2	3	4	6
3	3	4	7
4	3	4	8
1	2	5	1
2	2	5	2
3	2	5	3
4	2	5	4
1	3	6	5
2	3	6	6
3	3	6	7
4	3	6	8

Dimension 1 (`foo`) is varied more quickly than 2 (`bar` and `bar2`). As `foobar` is not qualified, it loops through its values continuously, independently of the dimension structure.

An independent set of virtual dimensions can also be created for summed || variables.

Multiple variables can be created in a single assignment from a list argument e.g.

```
variable m,n [0,10]
```

creates a variable `m` with the value 0 and `n` with the value 10. The quantities in the list are not required to be constants.

Whole lists can be assigned to multiple variables (V11.06.14) e.g.

```
variable x,y,z [0,1,2,3,4,5,6,7,8]
```

would assign `[0,1,2]` to `x`, `[2,3,4]` to `y` and `[6,7,8]` to `z`. The number of elements in the list must be a factor (including zero) of the number of variables. The alternative assignment order is selected using `;` rather than `,` to separate the variables e.g.

```
variable x;y;z [0,1,2,3,4,5,6,7]
```

would assign `[0,3,6]` to `x`, `[1,4,7]` to `y` and `[2,5]` to `z`, with the list members notionally allocated in order to `x`, `y` then `z` until the list is exhausted. Note how in this case the number of variables does not need to divide evenly into the list length.

Advanced topics: include files, expressions and variables

"Pre-parsing level"

The input file is parsed is at two distinct levels: an initial transformation of the input into a series of lines that are then passed to the "interpretation" level for parsing. This "pre-parsing" level is analogous to pre-processing of C/C++.

Each line is read from the current input source and processed independently. A trailing `\` marks a "continuation line"; the `\` is discarded and the line combined with the following line e.g.

```
echo my \
lo\
ng line
```

will be combined into the single input line `echo my long line`.

Then any "substitution variables" are replaced i.e. $*n* where *n* is an integer or $VARIABLE where an all caps variable is assumed to be an environment variable. The $*n* refer to additional command line arguments when pNMRsim was run (in the case of the main .in file) or arguments passed to `include` (see below). $*n can be used to refer to all the arguments from n onwards e.g. `$*2` would be replaced by `foo bar` if $2 were `foo` and $3 were `bar`. `()` can be used to distinguish the variable from surrounding text e.g. `$(HOME)othertext`. An error is generated if the variable is not defined unless a default argument has been supplied using the following syntax `$(`<*variable*>`?`<*if undefined*>`)`[24] e.g. `$(INCLUDEDIR?.)` will evaluate to `/home/user` if INCLUDEDIR is set to `/home/user` or `.` if it is not set. `$(`<*variable*>`?`<*if defined*>`:`<*if undefined*>`)` uses the second "argument" if the variable is defined, otherwise the expression is replaced by the third "argument" e.g. `$(DEBUG?echo Debugging:)` becomes "`echo Debugging`" if the environment variable DEBUG is set, otherwise no text is substituted. `$(`<*variable*>`!...)` (V12.08.02) works in the same way as `$?` except the test is whether the variable is defined *and non-empty* (as opposed to simply being defined).

Because arguments are passed by simple text substitution, care is required with expressions e.g. if $2 is `5+5`, the expression `2*$2` will be evaluated as 2*5+5 i.e. 15 rather than as 2*10. This can be avoided with parentheses i.e. `2*($2)` will be evaluated as 20 as intended.

Parsing of the input only takes place after this point. With the exception of the following instructions: `include`, `includeonce`, `variable`, `function`, `setenv` all other directives must be used inside the appropriate `{}` block[25].

Directives specific to the pre-parsing level:

`include` <*filename*> [<*argument*>*] includes the contents of <*filename*> at this point, allowing externally generated sections of code to be incorporate within the framework of a main `.in` file. The `includeonce` variant will only perform the include for a given

filename once and future uses of `includeonce` with the same filename will be ignored. Optional additional arguments are either evaluated (as real quantities) or passed on as is if quoted in single quotes and used as $ substitution variables ($1, $2 etc.) as the included file is parsed (see below), e.g. `include XiX.inc '$pw' $vrf` and the `XiX.inc` file:

```
pulse $1 $2 x
pulse $1 $2 -x
```

would result in the input lines:

```
pulse $pw <value of vrf> x
pulse $pw <value of vrf> -x
```

Although this resembles a "function call", macro substitution is a better analogy and care should be taken when passing quoted quantities (see below for the pitfalls of substitution variables).

A powerful feature of `include` is that evaluated (i.e. unquoted) arguments are "vectorised"; if an argument expands to a list, then a new include will be created for each element of the list. Multiple list arguments must have the same size and the lists are stepped synchronously e.g. `include XiX.inc '$pw' [0:30e3:60e3]` would result in the input lines:

```
pulse $pw 0 x
pulse $pw 0 -x
pulse $pw 30000 x
pulse $pw 30000 -x
pulse $pw 60000 x
pulse $pw 60000 -x
```

The example above also shows that sections for inclusion can be created within the input file using `{}` blocks. If a block name is not recognised, it is interpreted as defining a section of input to be used later by `include`. Hence, rather than create a separate `XiX.inc` file, it would be sufficient to include:

```
XiX {
  pulse $1 $2 x
  pulse $1 $2 -x
}
```

as a block at some point before it was required (e.g. in `pulseq`), and then use `include XiX $pw $vrf`. Reading an `XiX` file would only be attempted if an XiX "macro" had not been created. Such "internal includes" avoid the creation of multiple files and are marginally more efficient[26]. Note that the contents of such a block are only parsed when the block is "called", not when the file is scanned hence the $1 etc. will not be substituted at the point when the block is first encountered.

By default, input files (for `include`, `crystal_file` and `''` includes[27]) are looked for in the current working directory. The NMRSIM_PATH environment variable can be used to set a search path e.g. if the value is `/usr/local/shared/pNMRsim:.` then pNMRsim will first try to read the file from `/usr/local/shared/pNMRsim`, and if this fails from the current working directory. The path is not used if a relative filename is specified (i.e. it does not begin with `/`). This is useful for sharing common definitions.

`setenv` <*variable name*> <*value*> (V09.03.10) allows environment variables to be set, either to be passed to external function calls or as new/altered substitution variables.

The `-noexecute` command line flag allows the effects of this pre-parsing level to be examined without evaluation of the resulting directives[28].

~~Modules~~

~~pNMRsim can dynamically load "modules" that provide additionally functionality (or possibly modify default behaviour)[29]. These are loaded with~~

~~**`module`** <*filename*> where *filename* refers to a dynamic library (e.g. `extras.so`) which is installed in a suitable library directory (e.g. one listed in `LD_LIBRARY_PATH`). `module` on its own lists the modules that have been loaded. Modules must be loaded before the `spinsys` block i.e. functions cannot be added/modified part way through an input file.~~

Expressions and variables

Most arguments to pNMRsim instructions do not need to have fixed values. An individual quantity can be one of:

1. *A single value which may vary in a fitting / optimisation* e.g.:

`rotor_angle 54` sets the rotor angle to 54 degrees

`rotor_angle 54V` sets the rotor angle to 54 degrees, but indicates this parameter is a "variable" to be optimised in a fitting (the `V` is ignored if there is no fitting). `rotor_angle 54V1` sets the rotor angle to 54 degrees, flagging it as a fitting parameter, but also specifying its "uncertainty" at 1 degree. If unspecified, the uncertainty defaults to 10% of the parameter value[30]. If these values are too large, the fitting may have trouble converging on a solution; if too small, the fitting may progress slowly or easily get stuck in a local minimum.

2. *An "array" of fixed values* e.g.:

`rotor_angle {50,54,58}` steps the rotor angle through 50, 54 and 58 degrees. Empty items are ignored e.g. `{50,$1}` would be equivalent to `{50}` if $1 were empty, and `{50,52}` if it had the value `"52"`. The `V` suffix can also be used to denote a variable quantity e.g. `rotor_angle {50V,54V,58}` in the context of a fitting/optimisation would allow the rotor angle to vary (around its initial value) in rows 1 and 2 of a fitted data set, but fix it at 58 in the third.

`rotor_angle {54:4:58}` steps the rotor angle from 50 to 58 degrees in steps of 4 degrees. (A warning is given if the step size doesn't neatly divide into the difference between start and end values.) If omitted, the step defaults to 1 and the behaviour when the range is invalid is also changed;`[3:1:2]` gives an error, while `[3:2]` returns an empty list. The latter behaviour is useful in cases such as `[1:$n]` which returns an empty result if `$n` is less than 1.

`rotor_angle {"angles"}` uses the content of the text file `angles` for the array values. The file should contain a simple list (possibly empty) of numbers in text form. The different elements of the "array" specification can be combined e.g. `{"angles",54:56,"moreangles"}`.

Array elements enclosed in | | delimiters[31] (changed V11.03.06) denote a "sum array" e.g. `|50,54,58|`. In this case the calculation is repeated for each in the array elements and the results summed. "Normal" arrays can be nested inside sum arrays e.g. `|{50,54},{58,60}|` would create 2D data sets using the array `{50,54}` and `{58,60}` and sum the result. A post-fix `V` e.g. `|15,16|V`, is effectively shorthand for adding a 'V' qualifier to all the elements (V15.08.13).

Note that the array contents must be fixed when the variable is defined e.g. `{50,$myN}` is allowed if the contents of variable `$myN` are fixed. From V09.09.XX it is possible to include non-constant expressions (see below) in arrays with some restrictions: the

expression must return the same number of elements each time (i.e. the array cannot change size) and it is not valid to make an arrayed variable dependent on another arrayed variable. This is most useful for expressions involving fitting parameters e.g. to constrain amplitudes of two components to a fixed 1:2 ratio:

```
variable scalefactor 10V
…
scale |$scalefactor,2*$scalefactor|
```

3. *An expression.*

Expressions can be used to calculate results, which can include lists as well as single values. They can contain following operators

Numerical quantities: if a shift quantity is being defined, the suffix '`p`' can be used to denote a ppm quantity. `p` can be used without qualification in homonuclear problems or where the intended nucleus is obvious (V11.06.14) e.g. in `shift` commands. In other cases, the `p` must be followed by the nucleus index (from 1) e.g. if the `nuclei` were `13C 1H` then `1p1` would return the ^{13}C NMR frequency (in MHz).

`V` denotes a fitted variable and cannot be used *directly* in an expression e.g. `2*50V`, but fitted variables can be used in subsequent expressions.

Mathematical operators: `^`, `*`, `/`, `%`, `+` or `−`. `^` (power) takes precedence over `*`, `/` and `%` which take precedence over `+` and `−` i.e. `10*10+1` evaluates to 101 not 110, `10/4+1` to 3.5. `%` returns the remainder from a division e.g. `6.5 % 3` evaluates to 0.5.

`x`, `y`, `-x`, `-y` can be used as shorthand for the quadrature phases 0, 90, 180 and 270 degrees.

Functions are called using <*function name*>(<*arguments*>). The arguments can be list quantities, in which case the operations are "vectorised" e.g. `sin([0 45 90])` would return the list `[0.0, 0.7071, 1.0]`. Functions can be loaded dynamically or defined. The built in functions are:

Mathematical functions: `sin(`*x*`)`, `cos(`*x*`)`, `tan(`*x*`)`, `asin(`*x*`)`, `acos(`*x*`)`, `atan(`*x*`)`, `exp(`*x*`)`, `ln(`*x*`)`, `sqrt(`*x*`)`, `ceil(`*x*`)`, `floor(`*x*`)`, `round(`*x*`)`, `abs(`*x*`)` function identically to their counterparts in the C maths library with the exception that angles for the trigonometric functions are expressed in degrees (rather than radians). Note that "domain errors" (such as taking the square root of a negative number) will cause pNMRsim to abort with an error message. It is therefore important to ensure that invalid parameters are caught *before* expressions are evaluated in fitting/optimisation problems (e.g. with judicious use of `abs` or %). `ceil` and `floor` also have two argument versions e.g. `ceil(`*x*`,`*y*`)` returns the nearest multiple of *y* that is >= to *x*.

`<` `>` comparisons (V09.08.26) return 0 or 1 depending on whether the inequality is true (1) or not (0). There is no direct equality comparison, as this needs to be expressed carefully when dealing with floating point numbers. `if(x-y,..)` is fine for testing whether two integers `x` and `y` are the same, otherwise a test like `if(abs(x-y)>tolerance,...)` ought to be used.

`gcd(`*m*`,`*n*`)` and `lcm(`*m*`,`*n*`)` (V12.03.27) return the greatest common divisor and lowest common multiple respectively for a pair of integers *m* and *n*. This can be useful in determining synchronisation conditions.

`echo(`*x*`)` returns *x* but also prints it to the screen. This is primarily for debugging functions.

`extract(<input>, <selection>)` selects a subset of a supplied list, with the second argument supplying a list of indices (from 1) e.g. `extract(<input>,1)` would return

the first element of the list, `extract(<input>,[1:3])` would return the first 3 elements etc.

`head(<input>, <n>)` returns the first *n* elements of a list (`head`). *n* defaults to 1 if omitted.

`if`(<*expr*>,<*true*>,<*false*>) returns its second argument if <*expr*> evaluates to a non-zero quantity and the third argument otherwise e.g. `if($n,2,4)` will evaluate to 2 if `$n` is non-zero, 4 otherwise. Since expressions return floating point numbers, it is important to ensure that a negative result is truly zero; use the rounding functions to ensure this if necessary. There are no logical operators, but nested `if`s can be used instead e.g. `if(`<*cond1*>`,`<*cond2*>`,0)` will return a non-zero result (<*cond2*>) if both conditions expressions evaluate to non-zero quantities, and zero otherwise.

`Indicesof`(<*spin type*>) returns the index of spins of a given type within the spin system (V11.12.21) e.g. `Indicesof('1H'). Indicesof()` returns a list of all the spin indices i.e. from 1 to the total number of spins.

`last(`<*input*>, <*n*>`)` returns the last *n* elements of a list (`head`). *n* defaults to 1 if omitted

`noise(`*x*,*n*`)` returns *n* random numbers from a normal distribution of standard deviation *x* and zero mean. *n* defaults to 1 if omitted (V11.06.14).

`random(`*x*,*n*`)` returns *n* (floating point) random numbers within the internal 0 to *x*. *n* defaults to 1 if omitted (V11.06.14). Note the different ordering/use of arguments to the MATLAB `random` function.

`repeat(`<*input*>, <*n*>`)` creates a list with <*input*> repeated *n* times e.g. `repeat([2,3],2)` evaluates to `[2,3,2,3]`.

`replace(`<*initial list*>,<*indices*>,<*value(s)*>`)` (V09.08.26) selectively replaces elements of a list. For each index in the index list, the corresponding element in the initial list is replaced by the value from the values list (which must be of the same length as the index list). Alternatively if the values list has a single element, this value is used to replace all the selected elements.

`replace([1:5],[1,3],[3,1])` will return `[3,2,1,4,5]`

`replace([1:5],[1,3],0)` will return `[0,2,0,4,5]`

`rev`(*x*) returns the list *x* but with the elements in reversed order.

`sign`(*x*) returns –1, 0 or 1 depending on whether *x* is negative, zero or positive. Note that the comparison with zero must be exact, so this should be used carefully if *x* is non-integral.

`size`(*x*) returns the length of a list *x*.

`switch`/`switchdefault(`<*expr*>,<*if 1*>,<*if 2*>,*... [*<*default*>*]*`)` evaluates a different argument depending on whether the initial expression evaluates to 1, 2, 3 etc. An error is generated if the expression fails outside the range of arguments (or is non-integral). In the case of `switchdefault`, the final argument is evaluated if the expression is an integer, but fails out of range. Hence `switch(sign($x)+2,<if negative>,<if zero>,<if positive>)` will return one of three values depending on whether `x` is negative, positive or zero.

`sync_ratio(`<*target ratio*>, <*max*> *[,* <*limit*>*]*`)` takes the target ratio <*t1*>/<*t2*> and returns a pair of integers *m* and *n* whose ratio, *m/n*, approximates most closely to the target. The maximum *m* or *n* is set by <*max*>. The pair of numbers can be assigned directly to two separate variables, `variable m,n sync_ratio(0.8,20)`, or can be pulled out from the returned two-member list using `extract`, `head`, `tail` etc. The numbers can then be used to adjust timings / power levels so that two time-dependencies are properly synchronised and to determine appropriate ``synchronisation times''. If the optional <*limit*> argument is specified then the pair `[0,0]` will be returned if even the best ratio found deviates from its target by more than this value.

`tail(<input>, <n>)` returns the rest of a list after skipping the first *n* (default 1 if omitted) elements.

`Valueof('<variable name>')` and `Valuesof('<variable name>')` return the current value of a variable (equivalent to `$<variable name>`) or the full set of values that an arrayed variable can take (V08.03.01) e.g. for

```
variable x {10,20,30}
```

`Valuesof('x')` would return the (constant) value [10,20,30], while `Valueof('x')` would return 10, 20 or 30 depending on which row of the data set was being evaluated. This allows array variables to be "interrogated". Note, however, that `Valuesof` cannot be applied to expressions since their values are always context dependent.

`Errorof('<variable name>')` and `Errorsof('<variable name>')` behave as above but return the "error" on the parameter rather than its value. 0 is returned if the parameter has no associated error. The "errors" are initially the step values defined when the fitting parameter is created e.g. `20V2`. In the `finalise` block, the values reported are the errors determined in the fitting.

`Error('<error text>')` (V09.03.10) stops execution and reports the supplied error message. This is useful when using `if` to check for out-of-bound quantities e.g. negative delay periods e.g.

```
function verifypositive if(#1<0,Error('delay period must be
>=0'),#1)
…
variable tau verifypositive($tr-$t180)
```

will create the variable `tau` with the value of `$tr-$t180` if it is non-negative, otherwise an error will be reported and execution will stop.

`Warn(<message>)` and `WarnOnce(<message>)` (V11.09.29) print warning messages, with the `WarnOnce` form only printing the message the first time the warning is raised. For example

```
function warnnegative [#1,if(#1<0,Error('value is
negative')]
variable tau warnnegative($tr-$t180)
```

() can be used to control evaluation order[32] e.g. `(3+$pw)*5`.

`[]` denotes a list e.g. `[1,2]`. Unlike the static arrays considered above, the elements do not need to be fixed and can be expressions themselves e.g. `[sin($phase),cos($phase)]`. Nesting of lists is not supported, so `[[1,2],3]` would be expanded to `[1,2,3]`.

`"file"` creates a "static list" of numbers from reading in a text file (as above).

`` `<command>` `` creates a list based on executing a command through the Unix shell. The output of the command must be a simple list as for the `"` "static" list generator above. Since this is only useful when data needs to be generated "dynamically" as a function of the state of the calculation, <*command*> will normally contain at least one $variable; a warning is printed if this is not the case.

Expressions (and sub-expressions) that are constant (i.e. do not involve variable quantities) are evaluated when created. Non-constant expressions are re-evaluated at the start of a new "row" (in strict order of their definition), *after* any arrayed variables (`{}`) have been updated. For instance, in

```
variable pw 5
```

```
variable twopw 2*$pw
```

`twopw` will be fixed at 10, whereas in

```
variable pw {5,10}
variable twopw 2*$pw
```

the value of `twopw` is undefined until the start of the first row when `pw` is set to 5 and then `twopw` is evaluated to 10. `twopw` will have the value 20 in the next row. Note that `twopw` does not "have the value" {10,20} in the second example.

The parameters of several instructions must evaluate to a constant quantities e.g.

```
sw1 = $spin_rate*$gamma_angles
```

is invalid if `spin_rate` and/or `gamma_angles` are not constants (e.g. arrayed), since the spectral width in the indirect dimension must be constant across the entire simulation. There are more constraints on the variability of parameters in multi-dimensional calculations compared to those in which the calculation rows are independent e.g. `sw` must be constant in the former case, but may vary from row to row in the latter case.

In general, non-constant "expressions" are only valid within the "main loops" of `pulseq` and the `proc` blocks since they may depend on arrayed variables whose value depends on the data set row. Similarly sum-arrayed variables are not well defined within the `proc` and `finalise` blocks for the same fundamental reason. Warnings are generated if a variable appears to be used "out of scope" (V08.03.01).

In rare cases it may be useful to alter the default behaviour of evaluating constant expressions using the functions `const` and `mutable`. `const` forces the contained expression to be treated as constant and so immediately evaluated), while mutable causes the expression to be treated as non-constant and so not evaluated. Consider the expressions:

```
variable vec1 repeat(random(1.0),4)
variable vec2 repeat(random(mutable(1.0)),4)
variable vec3 const(repeat(random(mutable(1.0),4))
```

In the first case, `random(1.0)` is immediately evaluated and `vec1` is filled with 4 copies of the same random number (which may not have been the intention!). In the second case declaring the argument of `random` to be mutable means that the random subexpression becomes non-constant and so will be evaluated 4 times to fill `vec2`. However, as a side-effect, the complete expression `vec2` is then non-constant and so will be re-evaluated (to give a different set of 4 numbers) at the start of each row of a calculation. In the final case, the surrounding `const` forces the expression to be evaluated and then "frozen" with 4 random values. Alternatively the second argument of `random` (V11.06.14) can be used to avoid this issue entirely!

pNMRsim uses three distinct types of "variable" for storing quantities for subsequent recall with $<*variable*>:

System variables: these can be real quantities, integers, or string quantities (although all string quantities are "read only" and can't be used in expressions). Their usage is defined internally, and so they are principally used to display information about the current state e.g. `echo spinning speed = $spin_rate`. Currently defined system variables are

`alpha`, `beta`, `gamma`: current powder Euler angles[33]

`spin_rate`, `rotor_angle`

`sw`, `sw1`

`np`, `ni`: number of points in direct and indirect dimensions

`i_orientation`, `i_evaluation`: index variables giving the current index (from 0) into the powder orientation, and the number of times the calculation has been

evaluated (e.g. in optimisation/fitting). (Explicit user-defined variables can be created for other indices e.g. {} and || arrays).

`pi`: value of π.

`proton_frequency`: proton frequency (MHz) or 0 if undefined.

`time`: current time (zero at start of sequence). Only valid during `pulseq`.

`cputime`: elapsed CPU time (seconds) since pNMRsim started.

`rotor_phase`: current rotor phase. Only valid during during `pulseq`.

`name`: "base" filename (stripped of leading directories and `.in`)

`final_chisquared`, `final_optimisation`: store the final value of χ^2 or the value of the optimisation function for fitting and optimisation respectively. These two variables can only be used in the `finalise` block.

They can be used in expressions (if numerical) or where string arguments are used e.g. `save`, `echo`, (they are expanded to the current contents of the variable). In situations where the variable name is not isolated, `$(`<*variable*>`)` can be used e.g. `save $(name).fid`. The contents are undefined if a variable is accessed before it has been explicitly set (although this will usually be a default value).

Substitution variables: $*n* refers to argument *n* passed from the command line, or unquoted arguments to `include`. In addition, all caps variable names are assumed to be environment variables e.g. $HOME, and all occurrences of $<*number*> or environment variables are replaced as the first step in parsing each line of input (see "pre-parsing level" above).

User variables: these refer to floating-point variables created with `variable`. Their primary function is to connect together logically equivalent quantities into a single definition e.g. `variable pw90 5`.

Although most quantities in pNMRsim must be uniquely defined in a single place, a variable can be redefined provided that its value evaluates to a constant. Hence the following (rather silly example) will work

```
addtwo {
  variable a $a+2
}
…
  variable a 0
  include addtwo [1:5]
```

`addtwo` will be included five times and so the final value of `a` will be 10. But if `a` had been declared as an array e.g. `{0,10}` or a fitting parameter e.g. `5V`, then its definition cannot be changed. This is because variables that evaluate to constants are substituted immediately and so it is irrelevant if their values are changed at some other point. Non-constant variables in contrast must be defined in a single place since they are "live" throughout a calculation. Redefining a previously constant quantity as a non-const is, at least, poor practice, and will generate a warning (V11.06.14).

Variable names must consist of alphanumerical characters (letters + digits) or underscores (_) and must start with a letter. ALL CAPS variable names should only be used for environment (substitution) variables. Variables and expressions can be used when the result is an integer, but the evaluation is always performed with floating point numbers and an error is generated if the final result is not close to an integer.

User functions

New functions can be created using:

`function` <*name*> <*expression*> [`L|S`]*

Arguments to the function in the expression are denoted #*n* where *n* is the argument number (from 1). The optional second argument whether where each argument should be passed as a list (`L`) or as individual scalar values (i.e. vectorised, or "threaded"), `S`. For instance

with `function add #1+#2 SS`
`add([1:3],[2:4])` gives `[3,5,7]` (both arguments vectorised)

with `function add #1+#2 LL`
`add([1:3],[2:4])` also gives `[3,5,7]` since both arguments are passed as complete lists and the lists added.

but `function add #1+#2 LS`
`add([1:3],[2:4])` gives `[3,4,5,4,5,6,5,6,7]` since first scalar 2 is added to list [1,2,3], then 3, then 4.

If no flags are specified, all arguments are treated as scalar and the maximum value of *n* used in the expression defines the number of arguments[34].

The function definitions can be hard to read if multiple arguments are involved. From V15.08.13, it is possible to name arguments, and use the argument names rather than argument numbers e.g.

```
function divide(top,bottom) #top/#bottom
```

An empty argument name indicates that the argument will not be used.

Functions that are not in use[35] can be redefined although a warning is given. Note that functions with the same name but different numbers of arguments count as different functions. Previous definitions of functions are retained[36] but mapped to the original name plus an additional underscore e.g.

```
function sin sin_(2*#1)
```

means that sin(*x*) will now return sin(2*x*)!

Functions may be defined and used recursively e.g.

```
function factorial if(#1,#1*factorial(#1-1),1)
```

defines a factorial function, *N! = N*(N–1)…*1*. Such recursive definitions invariably involve an `if` statement to determine the end condition. In principle any loop can be written in such a recursive fashion, although it is easy to write functions which never terminate...

See `extrafunctions.inc` in the `extras` directory for examples of additional mathematical functions such as sinh, cosh etc.

Command line arguments

The full command syntax is
`pNMRsim [<flags>] <input file> [<arguments>]`

Valid flags are:

- `-abort` By default a warning message is printed if a non-fatal problem with numerical convergence is detected, but the calculation continues. If this flag is set, the calculation is forced to stop.
- `-debug` turns up the verbosity (if output is turned on with `verbose`).

`-enable:<option>` and `-disable:<option>` are used to explicitly enable or disable individual optimisations / optionals. A positive `-enable` will in some cases force an optimisation when pNMRsim does not expect it to be appropriate and so should be used with care. It also provides a quick way to check whether an optimisation is active; a warning is printed if a positively enabled optimisation could not be used. The optimisations are

`cache`	Cache propagators for reuse. Disabling caching may be useful if memory usage is a problem. The `simple` transient mode can cause problems with propagator caching so small differences may be observed with and without caching.
`classicQ`	`-enable:classicQ` will force "classic" treatment of second order quadrupoles even in multi-spin systems where results may be incorrect (for spins coupled to second order quadrupoles). Equivalent to SIMPSON.
`combinepropagators`	Use smart calculation of matrix powers when accumulating repeated identical propagators. (This will not be possible in MAS simulations if the propagation period is not a multiple of the rotor period).
`EDmatching`	Special case where "excitation" and "detection" operators are matched and the spectrum is purely real. Also a special case for heteronuclear decoupling.
`eigsymmetry`	Exploit the symmetry of the $\pm k$ eigenvalues (periodic problems only)
`forcepointbypoint`	Force each point of FID to be calculated by explicit propagation rather than re-using propagators. Enabling will make calculation very slow!
`gammacompute`*	Use γ-COMPUTE algorithm for γ angle integration under MAS. The algorithm is disabled quite conservatively and can be explicitly enabled if required.
`generalisedQ`	Explictly enable algorithms for "exact" treatment of quadrupoles and coupled quadrupoles.
`mergeprocessing`	(V12.08.02) Merge `initialproc` contents to the start of `proc` if independent of summation loop.
`minimalupdating`	Only update propagators etc. if they become invalid
`mzsymmetry`**	Exploit the symmetry of the m_z blocks
`mzblocking`	Exploit the m_z block structure for spins not subject to RF
`parallel`	Distribute power averaging in a parallel computation (MPI or threading, as enabled at compilation). MPI will be used if not explicitly disabled, while the experimental multi-core code (V11.06.14) is only enabled by default in NMRSIM_NUM_CORES has been specified.
`partitioning`	Exploit the m_z block structure for all spins (only relevant to Chebyshev propagation)
`periodic`	Exploit periodic symmetry (if `cells` specified)
`phasemodulation`	Use specialised code for sequences involving pure phase modulation. See `maxdt` for an additional optimisation associated with this algorithm. The code will be turned off,

	but can be specifically enabled in some cases where it is difficult to achieve synchronisation.
`preventoverwrite`	(V09.05.01) Prevent overwriting of existing files (by `save`). With the `abort` flag, execution will stop if a file would be overwritten. Not enabled by default in `pNMRsim` i.e. `-enable:preventoverwite` must be used explicitly.
`realhamiltonian`	Try to use a real (or diagonal) Hamiltonian rather than full complex
`smartprop`	(V11.03.06) Optimise cases where `prop` is used over regularly incremented durations by re-using previously calculated propagators (sub-optimisation of `combinepropagators`)
`synchronise`	Exploit synchronisation conditions between MAS, RF and observation

*Can be forced (but with care!)

**Not enabled by default–enable with care!

`-noshortcuts` is a quick way to disable all these "short cuts". There should be no discernable differences in the output with or without this option and this is only useful for debugging. Individual optimisations can be controlled with `enable` and `disable`. `-noshortcuts` can be combined with `enable` to enable specific optimisations.

`-nochecks` disables various checks on the input spin system e.g. a test is performed by default if the powder averaging range has been restricted to verify that the Hamiltonian is indeed duplicated in the omitted regions of the integration sphere. Similarly, the periodicity of the coupling network is checked for periodic systems. If the input file is known to be valid from a prior run, this option can be used to disable these checks and (slightly) reduce the start up time. `-nochecks` also prevents the display of some information messages e.g. estimates of calculation time that are otherwise displayed.

`-news` displays the "NEWS" file which summarises the pNMRsim version history.

`-noexecute` instructs pNMRsim to only read in the input file and output it on the standard output after applying any text and macro substitution. This has two uses: checking that text/macro substitution is working as expected and creating a single input file that does not have any external dependencies (though `include`). It will not find syntax errors or other problems in the instructions themselves.

`-randomise` randomises the starting point for the random number generator. By default any random numbers e.g. for noise values will produced in exactly the same sequence each time the program is run e.g. results will be exactly reproducible. Randomising the generator ensures that different numbers will be generated in each run.

`-silent` disables normal message output e.g. fitting progress.

`-qualify` <*string*> appends <*string*> to the `$name` variable derived from the input file name. This is useful for giving distinct names to the output of pNMRsim runs with different input arguments.

`-verbose:`<*option*> is equivalent to adding `verbose` <*option*> at the start of the input file, which is a convenient way of increasing the verbosity of the output without changing the input file.

`-version` (V08.03.01) outputs a string which can be used to identify the version of pNMRsim (format: year.month.date). This is taken from the first entry of the NEWS file. It also shows list optional features which may have been enabled at compile time.

Data fitting and optimisation

Parameters are supplied to the fitting/optimisation routines via the `par` block instruction `fit` (data fitting) or `minimise`/`maximise` (optimisation)[37]. Not all instructions apply to all cases e.g. `noiselevel` only applies to data fitting, while the choice of optimisation method applies to both fitting and parameter optimisation.

Alternatively fitting / optimisation commands can be placed in an `optimise` block after any processing, but before any `finalise` block (V12.08.02). This allows reference to fitting variables created after the `par` block, and may be a more logical long-term siting for such instructions.

`fit exp` <"*experimental" data*>* supplies the data for fitting, in the form of the name of a 1D Simpson data file. A warning is given if the spectral width indicated in the input file doesn't match that of the calculation. Multiple data sets can be fitted simultaneously by supplying a series of filenames with the order corresponding to the rows of the calculated data set. Fitting is not enabled if no data sets are specified.

For each data set loaded with `exp` (or `add`), variables `sw_exp`<*n*>, `np_exp`<*n*> and `ni_exp`<*n*> are created containing the spectral width, number of (complex) data points and number of rows respectively from data set *n* (numbered from 1). The spectral width is zero if it could not be determined from the file contents.

`fit add` <"*experimental" data*> *[*<*row indices*>*[*<*column indices*>*]] [*`-time|-frequency|-reversefrequency`*]* adds a single data to the fitting, optionally including range parameters that define subset of the data for fitting e.g. `fit add mydata 1:5` will fit the first 5 data points of `mydata`. Multiple `fit add` lines can be used to build a multi-row data set. If one set of indices are supplied, these are assumed to correspond to columns of the data set. To select whole rows use `[]` for an empty column selection e.g. `fit add mydata 1:5 []`.

The optional `-time` or `-frequency` flags can be used to indicate whether the data is in the time or frequency domain (assumed default) where this cannot be deduced automatically (as is the case for SIMPSON format input). This is of direct significance where a subset of the data is used as the spectral width will be scaled accordingly if the data is in the frequency domain. The `-reversefrequency` flag (V15.08.13) is useful for data that has been stored in "display" rather than the expected frequency order.

`maximise|minimise external` <*function*> <*arguments*> uses an external function (supplied by a "module") for optimisation rather than a simple sum. The external function is supplied a data object containing the calculated spectrum and must return a simple number. One (but not both) of `external` or `sum` must be specified for optimisation to be activated. The additional programming required by `external` can be avoided if the optimisation metric can be calculated in `proc` using e.g.

`fill` `` `<external command>` ``

on a single point FID will execute an external command (via the shell) and fill the FID with this value. Optimisation on the sum of the FID is then equivalent to optimising on the value returned by the external function, and does not require new code.

`fit mask [<data set 1> <data set 2> … |<column mask> [<row mask>]] [-exclude|-reversefrequency]` (V09.08.26) restricts the fitting to a subset of the calculated / experimental data by selecting data points to include in the calculation of χ^2. Selections are made using indices to include e.g. `1:10` would fit the first 10 points. In the case of multiple data sets, a selection should be made for each

row in order. For 2D data, the selection is made on the column indices and optionally on both columns and rows (to select a "rectangular" 2D subset). The `-exclude` option inverts the selection i.e. the specified points are excluded. With no arguments, any masking is suppressed. Note that the experimental and simulated data sets must already be the same size. Use the subsetting option of `fit add` if necessary to trim the experimental data to the right size. The `-reversefrequency` flag (V15.08.13) reverses the mask order, and is useful if the points ranges have been determined in software which numbers data points in "display" order rather than increasing frequency.

`fit|minimise|maximise method [-gradient|-simplex] [-randomise] [-real|-imag|-complex] [-no_eta_constraints]` selects between a gradient descent fitting (default) and a simplex optimisation. Gradient descent methods are significantly more efficient, but do require reasonably good starting points. Simplex is slow, but robust and should be used if gradient descent methods become unstuck.

The `-randomise` option randomises the starting point by adding $\pm\varepsilon$ where ε is the uncertainty on the parameter i.e. this starts the optimisation at a plausible point in the parameter space and the fitting should always converge on the same solution. Further investigation will be necessary if it doesn't!

The `-real`, `-imag` or `-complex` options (`fit` only) determine whether the data fitting is based on the full complex data set or just the real or imaginary component. Fitting the complex data set (default) includes more data and so should, in principle, give a higher quality fit, but fitting the real component only is often simpler.

The `-no_eta_constraints` flag (V08.03.01) disables the automatic creation of a 0 to 1 value constraint on tensor η values. Note that constraints are not created for tensor asymmetries specified in terms of *xx–yy*.

`fit|minimise|maximise iterations` [<*max iterations*>] defines the maximum number of iterations (fitting, default 100) or approximate function evaluations (Minuit optimisation). The current setting is shown if a new value is omitted. A value of zero means that the trial function will be calculated without further fitting, which is useful for testing the starting parameters (V15.08.13).

`fit noiselevel` <*noise*>* specifies the uncertainty on individual data points (assumed to be constant across the data set) i.e. the standard deviation of the noise. This is required for meaningful values of χ^2 and for proper parameter error estimates. If the noise level is not supplied, it is estimated from the standard deviation of the fitting residuals (reasonable if the systematic errors are negligible, but not otherwise). When fitting multiple data sets, the noise level ought to be an arrayed variable giving the noise level in the different (one-dimensional) data sets e.g. `fit noiselevel {0.1:1}`.

`fit normalise [`<*norm*>`] [-integral|-minmax|-abs]` (V09.01.03) normalises "experimental" and simulated data sets to avoid the need for an intensity scaling factor. (The parameters are the same as the `normalise` processing directive). Note that the normalisation of the calculated data is enabled automatically and so any explicit `normalise` directive in the processing is redundant. Note that the normalisation mode cannot be changed interactively.

`minimise|maximise precision` <*value*> (V11.03.06) informs the optimiser (Minuit) that the internal precision of calculations is less than double precision e.g. `minimise precision 1e-8` might be appropriate if calculations were only accurate to single precision. Convergence might otherwise be difficult.

`maximise|minimise sum` *[<indices> [<row indices>]]* indicates that the optimisation is to be performed on the sum of the spectrum (real part only) or a selected part of the spectrum specified by a set of indices e.g. `sum 30:40` means the sum of

points 30 to 40 (see `extract` for more examples). For 2D data, the first set of indices refers to the direct dimension and the second set to the rows (the rows are summed together if this is omitted).

`fit|minimise|maximise tolerance [`<*tolerance*>`]` defines the stopping condition. For fitting, this is the fractional difference in χ^2 below which the optimisation is taken have converged[38] (default 10^{-4}). If the new value is omitted, the current convergence tolerance is shown. In the case of simplex fitting, the user has the option of restarting the optimisation from the new starting point, which is often an effective strategy for pushing simplex along, and so it is often useful to lower this limit for simplex optimisations.

At least one parameter of the simulation must be flagged as variable by including "V" in the parameter definition (see above). As well as an initial value, each parameter must have a defined "error estimate" which sets the initial scale. This defaults to 10% of the parameter value, but must be set explicitly if the initial value of the parameter is zero. The "error estimate" is specified as a number following the "V" e.g. "`0V100`" sets the initial value to zero with an "error bar" of 100. A trailing `p` indicates a ppm value and a `%` indicates a scaling factor e.g. "`100V1%`". The error estimate only needs to be of an appropriate order of magnitude, but an excessively small estimate may lead to slow convergence while an over-large estimate may lead the optimisation astray.

Unless normalisation is used, fittings generally require a variable scale parameter to correctly adjust the intensities of trial vs. experimental data sets. This can be done explicitly by including `scale` <*scale parameter*>`V` in `proc`. If this is not done, a suitable starting value will be suggested.

Asymmetry parameters are not fitted directly as anisotropic parameters are more effectively defined for fitting purposes in terms of the anisotropy and "*xx–yy*" (anisotropy multiplied by asymmetry).

Instructions to save the fitted data set or output parameters (e.g. via `log_file`) should be placed in the `finalise` block. By default, if no `finalise` block is supplied when fitting, the fitted spectrum is automatically saved. See `save` for options controlling what information is saved following a fit.

More advanced optimisation

Normally fitting / optimisation will involve a single pass set up using the directives above. In more difficult multi-parameter fits, it may be necessary to optimise in more than step or explore the problem interactively. This will typically involve "fixing" certain parameters and optimising a subset of the parameters before optimising a different set of parameters. Parameters can be referenced by index (increasing from 1 in the order of creation) or by a name which pNMRsim creates for each new fitting variable e.g. `shift_1_aniso` to refer to the anisotropy of the shift of spin 1. The output of fitting will show the names created for the different parameters. Multiple parameters with a common prefix can be selected (V12.08.02) using * to stand for any trailing characters e.g. `t1_*` would select all parameters starting with `t1_`. This is useful for applying e.g. the same constraint to multiple parameters created with an {} array.

The relevant directives, which can be used with fitting or optimisation, are

`fix` <*parameters*>`+` e.g. `fix 1 x` fixes the values of parameter 1 and x.

`release [`<*parameters*>`+]` allows the specified parameters to vary freely. With no parameters, all parameters are allowed to vary.

`constrain` <*parameter*> <*min*> `-minimum|`<*max*> `-maximum|`<*min*> <*max*> (V08.03.01) applies a constraint to the specified parameter, as a minimum (lower) bound, maximum (upper bound) or an allowed range[39]. The constraint will not be

applied if the current parameter value falls outside the constrain (use `set` to move it). Note that the current restraint is replaced by the new one e.g. setting a minimum after setting a maximum will just constrain the lower limit i.e. `constrain` is not cumulative.

`unconstrain` <*parameters*>`+` (V08.03.01) removes any constraints from the specified parameters.

`set [`<*parameter*> [<*value*>]] changes the value of a fitting parameter. If the value is not specified, the current value of the parameter is given. `set` on its own displays the values of all fitting parameters.

`error [`<*parameter*> [<*value*>]] changes (or displays) the error estimate on an optimisation parameter. The error can be specified as an absolute number or a percentage (e.g. `10`%). Percentages are expressed in terms of the value of the parameter as originally defined, not the current value of the parameter in a multi-step optimisation.

`run [-allowinteractive]` starts the optimisation, stopping when convergence is achieved or the maximum number of steps is exceeded. `-allowinteractive` flags that interaction with the user is allowed (relevant to simplex fitting). If no explicit `run` directive is found, then the equivalent of `run -allowinteractive` is used to start the optimisation. The new optimisation only replaces the starting point if it is successful.

`statistics [-release_constraints|-if_constrained] [-silent]` (re)calculates the error values and statistics (e.g. correlation matrix). The `-silent` option disables output (but still updates internal variables). `-release_constraints` performs the calculations with any constraints temporarily lifted. This will give more meaningful values for the errors, but may fail if parameters are at or very close to their limiting values. `-if_constrained` (V12.08.02) is equivalent to `-release_constraints` but the command is only executed if there are constraints to release.

`system` <*shell command*> (V09.09.XX) executes the rest of the line via the "system" command. This can be useful for starting display programs e.g.

```
system simplot experimental.spe fitted.spe &
```

`interactive` switches into an interactive mode in which optimisation directives are input and executed. The following additional directives are only applicable to this interactive mode:

`finish` indicates that the optimisation is complete. The "`finalise`" block (if any) will be run on the basis of the last successful set of parameters.

`abort` shuts down the optimisation and pNMRsim without running `finalise`.

`help` displays the allowed directives.

For example

```
fit fix 1
fit iterations 10
run
fit release
fit iterations 100
fit interactive
```

Fixes parameter 1 and optimises for up to 10 iterations before allowing all parameters to vary and increasing the maximum number of iterations to 100. The fitting then drops into interactive mode.

Parallel computation

Parallel computation is extremely useful in solid-state NMR simulations since the computation often needs to be summed over multiple orientations or other inhomogeneous factors such as RF inhomogeneity. The summations over the powder orientation and any || summation array are combined (V11.06.14), so that parallelisation will be effective when either or both summations are present. As of V11.06.14, two approaches can be used to split up a calculation over multiple processors (use `-version` to see which is enabled):

MPI: MPI should be used to exploit large computational cluster systems. The MPI code must have been enabled at computation and the resulting binary must be submitted through the MPI runtime system e.g. `mpirun` unless parallel computation has been disabled with `-disable:parallel`. One processor is used as a "master" to distribute chunks of works between "slave" processors, and so the maximum speed up is the number of processors requested from the MPI runtime minus one.

Threading (V11.06.14): In this approach, the process splits into multiple "threads" of computation which a multi-core processor should automatically distribute between the different cores. In this case the work is split into equally divided chunks between the threads, ensuring full use of all cores requested but also meaning that overall completion will be limited by the slowest core. By default all available cores are used. This can be overridden by setting the environment variable `NMRSIM_NUM_CORES` to the number of cores to be used. This is useful to retain some processor cores for other jobs.

Note that each execution thread runs independently. Particularly in the threading approach this means that any text output from the parallelised `pulseq` block will be rather jumbled.

Keeping things efficient

Here are some tips for keeping your simulations as efficient as possible:

- Comment out output lines (`putmatrix`, `echo`) and keep the verbosity to a minimum. Don't just to divert verbose output to a file; the output commands are intended to help debugging, not for efficient execution.
- Keep it simple. The simpler your `.in` file, the more chance pNMRsim has of finding an efficient route. In particular, keep the number of RF channels to a minimum as this allows more blocking using the Fz quantum number.
- Avoid changing the contents / timing of a pulse sequence fragment since this will force all stored propagators associated with that fragment to be discarded.
- Try to ensure that sampling, rotation and RF are synchronised as far as possible e.g. adjust the RF power so that the TPPM decoupling period is an integral divisor of the rotor period. If the two timescales are asynchronous, propagators needs to be integrated over the entire NMR signal / indirect time dimension, considerably slowing the calculation. Note that the frequency domain calculations do not require synchronisation with the sampling rate.
- Try to use `acq` rather explicit propagation using `prop` as optimisations are then easier to deduce. If using `prop`, appropriate "synchronisation hints" can maximise the opportunities for re-use of previously calculated propagators (see `store`).
- The calculation of phase modulated sequences under MAS is particularly efficient if all the elements have the same length (or rather the duration of each element is a multiple of the smallest). Gaps between pulses (strictly periods with a different, but common, amplitude) can be accommodated provided the restriction on the element durations is always satisfied.
- Use coherence filters rather than explicit phase cycling. Similarly the `transfer` command is much "cleaner" and quicker than an explicit simulation of CP.

- Keep the number of distinct powder orientations to a minimum. Restricting the orientations to a hemisphere tends not to help greatly, but restricting to an octant (for problems with a single principle axis) allows the number of orientations to be significantly reduced. `gamma_steps` has a less direct impact, since the time-limiting step is the stepwise integration of the propagators i.e. `maxdt` has a big impact on the efficiency. `auto_opt` can be used to optimise these parameters, but be careful not to significantly modify the problem after heavy tuning.
- Use ideal pulses unless specifically interested in the effects of soft pulses. The propagators for ideal pulses can be calculated once and re-used. `pulse180` is especially useful if it allows an RF channel to be dropped. Similarly, use 1st order treatments of quadrupoles unless non-secular effects are important.
- Start with *x* magnetisation where possible, rather than *y* or applying a 90 pulse to *z* magnetisation. Similarly you may be able to avoid an explicit "read" pulse by using an appropriate detection operator.
- You can use `-nochecks` to disable initial checking on problems which have been previously checked. The savings in start-up time are unlikely to be noticeable (particularly for large problems), but it will get rid of many warning messages. `-silent` will suppress all output.

Are you sure you need pNMRsim?

If you can do your simulations without significant pain in something like SIMPSON, do so. SIMPSON has been used and tested extensively, and the fact that it doesn't try to be as clever makes it unlikely that obscure bugs will suddenly appear in unusual circumstances.

- **Sign conventions**: the handling of signs/Euler angles etc. has not been rigorously evaluated.
- **Stability**: pNMRsim is a work in progress which will evolve according to research interests and so is likely to remain in a "beta" state for some while. If you can't put up with new versions that add some features, break others etc., use SIMPSON! If a calculation is behaving oddly, avoid multi-row simulations (where the `miminalupdating` optimisation may mess up) and try the `-noshortcuts` flag.
- **Specialisation**: pNMRsim is designed to be efficient for multiple spin systems and flexible enough to handle a wide variety of problems as intuitively as possible. This means that it won't always be as computationally efficient or user friendly for particular specialised tasks e.g. fitting of simple spinning sideband manifolds. For small spin system problems, however, its relative inefficiency may be outweighed by ease of use factors.
- **Support**: although we like pNMRsim and are happy to see people use it, we are deliberately not trying to give it away, won't accept any liability for mental distress or otherwise.

---

[1] SIMPSON: `acq` (and `pulseq`) are *procedures* not blocks

[2] In its previous incarnation `initialproc` was applied after powder averaging but before || summation arrays. Since its reintroduction in V12.08.02, the processing in initialproc is applied to every transient. This allows orientation-dependent processing but may considerably slow computations if used unwisely. The `mergeprocessing` optimisation allows an `initialproc` to be defined but its contents will be merged into `proc` if independent of summation variables.

[3] SIMPSON: `proton_frequency` must be in `par` rather than `spinsys`, but this breaks pNMRsim's "define before first use" rule.

[4] The state restriction is incompatible with the "generalised" treatment of >1st order quadrupoles.

[5] Not all temporary propagators count towards this limit e.g. accumulated propagators in indirect dimensions. Reaching the cache limit is used as a general signal that memory is tight and results in more temporary memory allocations being released.

[6] Parameters for the 3-angle ZCW sets kindly supplied by Matthias Ernst (ETH Zürich).
[7] This can be combined with other arrayed variables / 2D, but the size of the 2D array can never exceed the number of powder orientations i.e. the powder orientation is never "looped".
[8] Unlike other commands, the rest of the line is treated as a whole and is not broken up into "tokens".
[9] Note that the defaults do not exactly correspond to `-sw/2` and `+sw/2`; they are adjusted by half a bin width to ensure that zero frequency falls in the middle of a bin.
[10] It is not possible to output the "current propagator" as this has no meaning in pNMRsim.
[11] Pulse sequences work very differently in SIMPSON. The propagation under a pulse sequence is essentially "interpreted", stepping through `pulseq` and accumulating propagators which can be stored and recalled. This is effective for simple problems, but makes it more difficult to express high-level concepts such as phase cycling.
[12] The sign conventions used by the underlying library and earlier pNMRsim versions are such that the offset has the opposite sign to shifts. The `-version` output from recent versions show which behaviour was defined at compile time.
[13] By default, phases are expressed relative to a notional time origin and a correction is applied following an off-resonance pulse for the accumulated phase difference between on and off-resonance frames; the `-coherent` flag suppresses this frame shift.
[14] Previous versions had a `par` version of `prop` for this purpose. This has been dropped in favour of the more flexible `include` alternative.
[15] SIMPSON: `store` simply stores the current propagator. If you need to continue a previous fragment, use `prop` to insert the freshly stored fragment into the new sequence. pNMRsim has no equivalent of `reset` (`store` automatically clears the current sequence).
[16] The entire FID must be acquired in one shot (SIMPSON allows the signal to be acquired point by point). This makes it much easier to look for optimisations.
[17] In fact, if the sequence satisfies certain restrictions (see Keeping things efficient), it may be sufficient to integrate over a single rotor period rather than the entire synchronisation period. The synchronisation period must still be specified in order to find the correct timebase for the algorithm.
[18] The `get` directive subtly breaks this rule by allowing variables to be created which are not known at the start of the simulation. This limits the uses to which the values from `get` can be used to essentially output and reporting.
[19] Note the differences from the SIMPSON main commands; no leading "f", data set is not specified (there is only one active signal). The other SIMPSON functions are not supported (they are more relevant to data processing rather than simulation).
[20] The time domain and frequency domain versions of the mixed lineshape are not identical, but this differential definition is common (e.g. SIMPSON).
[21] In SIMPSON usage, ranges are specified in terms of frequencies (which are then translated into discrete indices). pNMRsim uses indices directly, although these may be computed using (constant) expressions.
[22] Unlike SIMPSON, expressions involving coherence matrices are not accepted. Such matrices only consist of 1s and 0s and so can't really represent spin operators.
[23] `n0` can be used as a synonym for `np`.
[24] The "if missing" text is expanded recursively so `$(2?$1)` will expand to the contents of $2 if it is defined or $1 if not.
[25] Note that definitions such as `variable` and `function` must be used in or before the `spinsys` or `par` blocks.
[26] The overhead of `include` is not large in any case. Even in blocks which are "evaluated" repeatedly such as `pulseq` and `proc`, the *parsing* is only done once.
[27] ~~`module` loads are slightly different: the relevant system variable is LD_LIBRARY_PATH and this is only used if the module name does not contain a directory separator (/).~~
[28] This will not fully work if expressions involving user defined variables e.g. `$rf`, need to be *evaluated*, since the `variable` directives will not have been executed. Quoted arguments are not evaluated and so will not cause problems.
[29] ~~Dynamic loading will work on most Unix platforms, but is not compatible with Windows-based systems such as Cygwin~~.
[30] This 10% default is inapplicable to parameters with default values of zero. In this case, the uncertainty must be supplied.
[31] In versions prior to V11.03.06, () was used to denote a "sum array". This, however, created a parsing ambiguity with () as a grouping operator.

[32] The order in which components of an expression are evaluated is not defined. Some expression components, for instance, may not evaluate some arguments at all e.g. in `if(N,extract(A,N),0)`, the `extract(A,N)` argument will only be evaluated if N is non-zero (otherwise `extract` would fail). Since operations should not have side effects, evaluation order should not be significant.

[33] The "gamma" angle is problematic since it is determined by a number of parameters; the powder averaging scheme, `gamma_zero`, explicit vs. implicit integration etc. As of V11.09.29 the value of `$gamma` is not well defined.

[34] The number of arguments is determined by simple scanning of the text of the expression prior to full *parsing* of the expression. This could misidentify a character sequence as an argument in pathological cases, e.g. in `` `cat myfile#1` `` the `#1` is part of a text string and not an argument. The best solution simple solution is to specify the number of arguments explicitly with the L/S argument specification, or in current versions, use named arguments.

[35] Like variables, "in use" means being part of an unevaluated expression. For instance, if `x` is a constant, `sin($x)` would be evaluated immediately and so the sin function would not be "in use". If, however, x was not fixed e.g. `variable x {0:10:90}`, then the definition of sin must be preserved and redefinition is not permitted.

[36] This contrasts with variables where new definitions silently overwrite previous ones. While variables are naturally used as temporaries, function definitions are expected to be global and so overwriting definitions to change functionality always generates warning messages (disabled by `-nochecks` or `-silent`).

[37] Optimisation is only available if pNMRsim has been compiled with the Minuit optimisation library.

[38] Note that Minuit defines its convergence parameter with a scaling factor of 10^3. As of V11.03.06, pNMRsim takes care of this difference in definitions automatically.

[39] Note that if constrained parameters have "internal" values which differ from the corresponding "external" value. The parameter values displayed during the fitting reflect these internal values and so can't be directly related to the "normal" value of a parameter.

pNMRproc

pNMRproc is essentially a cut-down version of pNMRsim for processing of data. It is particularly useful for batch processing of data or applying transformations not supported in normal NMR processing software. That said, the pNMRsim model doesn't always sit easily with processing e.g. the number of data rows in fixed over the course of simulation, whereas this may changes in processing. This creates conflicts with features such as virtual dimensions.

It is used in the same way as pNMRsim with the following differences:

- The command line is `pNMRproc [<options as pNMRsim>] <.in file> <data file> [<arguments]` i.e. the input data file is specified explicitly. Multiple data sets are not supported, since this can be handled more flexibly via command-line scripting.
- The variable `$name` is set to the name of the input file (trimmed of directory information) rather than (as pNMRsim) the name of the `.in` file.
- The only allowed blocks are an optional `par` block and an obligatory `proc` block.
- `sw`, `sw1`, `np` and `ni` are taken, if possible, from the input data file, together with information about time vs. frequency domain, and referencing information (V11.06.14). The spectral widths can be set or overridden explicitly in `par`. nD structure can be set up in the same way as pNMRsim, although support for >2D data sets is minimal. `set`/`setdomain` can be used to set/override information about the domains and/or 2D quadrature detection.
- `add` <*filename*> `[`<*scale*>`]` adds another dataset (which must be of the same size) multiplied by an optional scaling factor (default 1).
- `autophase -zeroonly` (V13.1.13) tries to automatically phase the spectrum based on a fairly crude algorithm that maximises the intensity in the real part of the spectrum (which will only work well with spectra with clean baselines and resolved peaks). The `-zeroonly` flag restricts the optimisation to zero-order phase, which is useful for spectra where the first order correction is known to be negligible e.g. spectra taken from the top of an echo.
- `sum` [<*row indices*>] adds all the rows of a 2D data set to create a 1D data set. Optionally (V15.08.13) a specified subset of the rows can be added together (and the remainder discarded).
- `baselinecorrect [`<*baseline range*>`]` does a simple DC offset correction based on the average of the selected range of data points (selected by index). Note that this is applied individually to each row of a multi-row dataset which is likely to be the Wrong Thing for true 2D data.
- `splitsincos` (V09.11.10) takes 2D SpinSight data in which cos and sin fids are combined in a single row and creates a "normal" hypercomplex data set with cos and sin fids in alternate rows.
- `ftindirect` [(flags as for `ft`/`ft2d`)] performs Fourier Transformation in the indirect dimension only.
- Write protection is enabled by default. Use `-disable:preventoverwrite` to override this.

Note that the output file will only contain information that pNMRsim/proc understands i.e. spectral widths, time vs. frequency domain, spectrometer / reference frequencies (V15.08.13). Any other metadata will be missing. The Matlab format preserves the most information, followed by the SIMPSON formats.

Notes

Interoperability with SIMPSON

Different design choices in pNMRsim mean that the input files are becoming increasingly incompatible with SIMPSON. The `pNMRsim2SIMPSON.sed sed` script can be used to convert a pNMRsim file into something that is more likely to be SIMPSON compatible e.g. `sed -f pNMRsim2SIMPSON < RR.in > RR_SIMPSON.in` will create a SIMPSON input file from the pNMRsim input `RR.in`. The script makes the following changes:

The `proc` and `pulseq` blocks are turned into Tcl functions

Arguments to `verbose` are discarded and replaced with a default

Processing commands, `addlb` etc, are adjusted to SIMPSON usage (`faddlb` etc.)

Any powder averaging qualifiers (`:hemisphere` etc.) are removed. *This may result in noticeably coarser sampling in the SIMPSON run*.

Commands unsupported in SIMPSON are stripped out (`auto_opt`, `tolerance`, `precision`, `log_file`, `scalefirst` etc.)

Trailing Vs (following digits) are removed

$(*variable*) is replaced by $par(*variable*)

Lines immediately following ##DELETENEXT are removed. The # line is a comment to pNMRsim (and SIMPSON), but instructs the conversion script to delete lines that are incorrect (or in the wrong place) for SIMPSON

##DELETE strings are deleted; useful for commands that are valid (correct for) SIMPSON but not pNMRsim

Some other minor differences can also be accommodated:

Always include a `channels` directive in `spinsys` (even if blank). SIMPSON insists on this.

The `method` directive is ignored by pNMRsim, but can be included for compatibility with SIMPSON (which will run very inefficiently if `method` is set inappropriately).

Don't use `F`<op> to specify spin operators or omit Euler angles in interaction specifiers; SIMPSON will not accept these.

More fundamental differences, in particular the different models for pulse sequences, cannot be readily accommodated by automatic translation and must be changed by hand.

Future directions

There are a number of things that pNMRsim is missing and which may (or well may not) be worth providing:

- The number of data processing commands is more limited than in SIMPSON. On balance, however, these belong more naturally in distinct data processing programs. In this spirit, pNMRproc provides some additional functions.
- Relaxation and exchange: a full implementation, beyond the available manipulations of the density matrix, would require major additions to the underlying library and a separate Liouville-space version of pNMRsim.

Known problems

- pNMRsim cannot always work out how to best exploit block structure (e.g. spotting double quantum operators), particularly if different parts of a simulation (with and without RF for example) have different structure. It would be difficult and error-prone to address this and the best approach for these problems would be to switch to describing the problem in sparse Liouville space.