Supplementary Information for:

INFOS: Spectrum Fitting Software for NMR Analysis

Albert A. Smith*

ETH Zürich, Physical Chemistry, Vladimir-Prelog-Weg 2, 8093 Zürich, Switzerland

A.S.: alsi@nmr.phys.chem.ethz.ch

INFOS User Manual

Albert A. Smith

Contact: alsi@nmr.phys.chem.ethz.ch (ETH-Zürich email)
alsi-nmr@users.sourceforge.net (INFOS email)

Table of Contents

1. Getting Started

1.1. Description

The INtelligent Fitting Of Spectra (INFOS) is a software package written in MATLAB, intended for fitting of spectra with many resonances, in order to better quantify peak amplitudes and integrals, improve peak positioning, and improve identification and separation of peaks, as compared to methods such as integration and standard peak picking. INFOS is able to handle large spectra and large numbers of peaks by breaking spectra into parts and fitting the sub-spectra, and iteratively recombining and refitting the sections. Additionally, INFOS uses pre-calculated lineshapes to accelerate spectrum calculation. Processing and acquisition information is used to generate lineshapes for fitting, therefore producing improved fits. Optional peak picking is done iteratively in combination with fitting; periodic addition, removal, splitting, and combination of peaks are interleaved with re-optimization of the spectrum fit. Statistical methods are used to establish the level of noise and of peaks, and in order to determine whether spectral features result from one or more peaks. INFOS combines ease of use with flexibility; all settings for fitting can be calculated internally by the program, however, almost all fitting settings can be controlled by the user in order to either improve fitting or accelerate calculations. Further details and examples of applications are provided in the accompanying publication.

1.2. Basic fitting: the FitSpec function

Fitting a spectrum may be done completely within the FitSpec function. FitSpec may either optimize a given peak list by varying position, amplitude, and/or linewidth, or may perform the peak picking as well, with periodic modification of the peak list to optimize the spectrum fit. We will refer to the former case as a *fixed peak list* and the latter as a *dynamic peak list*. Additionally, settings controlling the behavior of FitSpec may be determined internally or fitting instructions may be provided by the user. The commands are formatted as follows.

With a dynamic peak list, without any user instructions:

```
fit=FitSpec(spec);
```

or with a dynamic peak list, with additional user instructions:

```
fit=FitSpec(spec,par);
```

In the first case, the program uses all default options, some of which are calculated during the program run depending on properties determined from the ‘spec’ structure. In the second case, the user determines some of these options, and specifies them in the ‘par’

structure. The ‘spec’ structure contains the experimental spectrum and frequency axes (in ppm), and also may contain acquisition and processing information for the spectrum. The fields and their form for ‘spec’ are detailed in 2.1. The ‘par’ structure contains instructions for spectrum fitting, which is detailed in 2.2. The output structure, ‘fit’, contains lists of peak positions, linewidths, amplitudes, and integrals, as well as the calculated spectrum and error spectrum, and information on the analysis of the fit. Details are found in 2.3.

With a fixed peak list, without any user instructions:

```
fit=FitSpec(spec,[],fit0);
```

or with a fixed peak list, with additional user instructions:

```
fit=FitSpec(spec,par,fit0);
```

As before, the ‘par’ structure is optional, but now an initial fit is given in ‘fit0’. ‘fit0’ has a very similar form as ‘fit’, and one may use the output of another fitting run in the ‘fit0’ argument. ‘fit0’ also may containing additional fields to restrict how variables are changed (fixing parameters or setting ranges on the parameters). This is described in 2.5. Note that when ‘fit0’ is provided, then peak position/linewidth/amplitude are optimized, but by default, the peak list itself is not changed (no peaks added or removed). This can be overridden with user settings in the ‘par’ structure.

1.3. Summary of options

Here we summarize the options for the FitSpec function. All options have default values, so do not need to be user specified, but it is often useful to have control over some of these. Details on setting the individual fields found in the ‘spec’, ‘par’, and ‘fit0’ structures are given in section 2.

1.3.1. Restricting fit variables

The FitSpec function allows restriction of fit variables, either by establishing allowed ranges for those variables or by locking variables in place. If an initial fit is given, then the ‘fit0’ structure may contain the field ‘fit0.fixed’ (or alternatively, ‘par’ may contain ‘par.fixed’). This variable allows one to lock all intensities, peak positions, and/or linewidths separately for each dimension. Also, if an initial fit is given, then ‘fit0’ may have fields ‘fit0.rangeX’ and ‘fit0.lw_rangeX’ which give upper and lower bounds for peak position and linewidth, respectively, for each peak (X is the dimension number). Then, one may restrict each peak differently if desired. Typically, it is not a good idea to use these options if the peak list is not fixed (i.e. peaks are going to be added during the fit), although it is possible.

Alternatively, one may specify restrictions on fit variables in the ‘par’ structure. ‘par.rangeX’ and ‘par.lw_rangeX’ restrict peak position and linewidth, but are a single element in the ‘par’ structure, and restrict the variation of peak position and linewidth from its initial position (+/– half the value given in ‘par.rangeX’ or ’par.lw_rangeX’). One may also restrict the maximum and minimum linewidths of all peaks using ‘par.max_lwX’ and ‘par.min_lwX’. These options may be used with non-fixed peak lists, but care should be taken to not over-restrict the fit.

1.3.2. Lineshape settings

Parameters describing lineshapes for a given dimension may be placed in two different locations. The ‘spec’ structure stores these in ‘spec.par.dX’, and the ‘par’ structure stores them in ‘par.dX’. If the FitSpec function encounters the same parameter in both structures, it will use the value in ‘par.dX’. Typically, one stores acquisition and processing information in the ‘spec’ structure, but specifies the type of signal decay in the ‘par’ structure, although the FitSpec function does not require this convention.

Currently, INFOS provides import functions for Bruker Topspin data, which is described in section 3.1.1. This imports all necessary acquisition and processing parameters. The definitions for these parameters are detailed in 2.3. In addition to acquisition and processing parameters, the type of decay may be specified. Here, options are Gaussian and Lorentzian decay (corresponding to signals that decay with Gaussian and exponential functions). This is specified in ‘par.dX.Broad’, as ‘gauss’ or ‘lorentz’. One may also specify a fixed mixture of Gaussian and Lorentzian broadening, as ‘mixXX’ where XX/100 is the fraction of Gaussian broadening. One may also include a fixed amount of Lorentzian or Gaussian linewidth, specified in ‘par.dX.lorentz0’ and ‘par.dX.gauss0’. The default if not specified is pure Gaussian decay.

1.3.3. Iteration settings

The FitSpec function uses iterative fitting both when using a fixed peak list, and when adding peaks. When using a fixed peak list, ‘par.IterFact’ and ‘par.n_iter’ control the number of iterations within a sub-spectrum fit and number of fit iterations over the whole spectrum. When using a dynamic peak list, an additional parameter, ‘par.add_peaks’, controls the number of times that peaks will be added and removed. Setting this equal to zero will fix the peak list, which is the default if ‘fit0’ is provided. See section 2.2 for details of the ‘par’ structure.

1.3.4. Peak addition/removal settings

If the peak list is being edited during fitting, then four functions are used to modify the list, one which adds peaks to badly fit regions, one which removes peaks that do not significantly improve the fit, one which splits single peaks into multiple peaks, and one which combines pairs of peaks into single peaks. Decisions to add, remove, or split peaks is determined based on whether or not noise exceeds a given cutoff, specified by 'par.cutoff'. Decisions to combine peaks are determined by calculating whether peak combination raises the noise level less than is expected from removing a noise peak. The parameter giving the relative expected change is the *noise per peak*, given in 'par.npp'.

At each step of peak addition, add, remove, split, and combine functions may be run. By default, none of these run after the last fit optimization (such that peaks are not added or removed without another chance to optimize parameters), and only remove and combine functions run immediately before the last fit optimization. Otherwise, all functions run. However, the user may take full control over which functions run at each step in the 'par.control' field, described in 2.2.8.

The final option in peak addition is the sign of peaks. Peaks may be all positive, all negative, or both. This is determined in the 'par.sign' field ('+','–','+–').

1.3.5. Advanced options

As a fourth argument to the FitSpec function, one may provide a cell, 'shapes0', which allows usage of user-defined lineshapes. Usage and the form of this cell are detailed in 2.6. A fifth argument, 'shapes', may be also provided, which contains lineshapes for sub-spectra and should be generated with the FitSpec function as discussed in 2.7.

1.4. Additional programs

INFOS provides additional fitting programs that setup and use the FitSpec function for particular applications. The first of these is the FitTrace function (section 3.6.2), which simultaneously fits a series of spectra for which the amplitudes in those spectra are described by some user defined function, for example for fitting exponential decay in a time series of spectra. The second program is FitError (section 3.6.4), which analyzes a fit calculated with the FitSpec function to determine the approximate error on the various fitting parameters. Finally, FitEditor2D (section 3.6.1) can be used to interactively edit a fit of a 2D spectrum.

1.5. Features and limitations

1.5.1. Handling of large spectra

The INFOS fitting algorithm is able to handle fitting of large spectra with many peaks by breaking the spectrum into sections and fitting each of these. Breaking the spectrum into small sub-spectra makes the individual fit optimization simpler, and therefore less demanding on memory and computer time. However, if sections become too small, peaks overlap between sections and fitting becomes worse, since fits of individual sections become highly interdependent. The FitSpec function calculates sizes of the sub-spectrum, which is optimized for most spectra. However, spectra with many peaks by default have smaller sub-spectra, which makes fitting of broad peaks more difficult. In the case that a large range of linewidths exists in a spectrum, then may be useful to decrease the number of sub-spectra from the default, using 'par.grid' (see section 2.2).

1.5.2. Shape generation from acquisition and processing information

INFOS improves fit quality by generating lineshapes based on acquisition and processing information. Shapes are generated by processing either Gaussian and/or Lorentzian signal decay, according to the acquisition time, spectrum width, processed resolution, and the apodization function. The shapes are then generated for a range of linewidths, which are applied in the spectrum fitting. This calculation is done during setup, and then is stored for later use in the program, contributing significantly to the fitting speed.

Because lineshape calculations are done during setup and stored, the FitSpec function is limited to only one variable describing the linewidth. Therefore, optimizing mixes of Gaussian and Lorentzian lineshapes separately for each peak is not possible. However, one may specify a fixed amount of Gaussian or Lorentzian line-broadening to be applied to all peaks, and optimize the other type of broadening. Additionaly, one may specify a particular fraction of Gaussian and Lorentzian broadening, so that both widths vary proportionally according to the given fraction. Specification of acquisition and processing parameters, and signal decay is described in section 2.3. One may go further and specify arbitrary lineshapes. This is considerably more complicated to setup, but is executed by giving a cell, 'shapes0', which contains structures for each dimension that contain the lineshapes. This cell is described in section 2.6.

The major limitation to this method of lineshape generation is that only one optimization variable may describe the lineshape. This is helped by the ability to add some Lorentzian character to Gaussian lineshapes and vice versa. A second limitation is that

lineshapes may not be correlated between dimensions. For example, if a lineshape is the result of a poor shim leaving a foot, then that foot should extend diagonally between two dimensions. However, even if an arbitrary lineshape is specified by the user to include the foot, it will not appear diagonally across the dimensions since this would require correlation of lineshapes between the dimensions.

1.5.3. Peak addition, removal, splitting, and combination

If a fixed peak list is not used, then INFOS attempts to optimize the peak list, such that the spectrum is well fit, but fitting of noise is minimized. A cutoff is used to determine what peak height is no longer considered noise, and is used to add/remove/split peaks. If no user settings are given, then the FitSpec function will analyze the noise level, and the peak heights and attempt to set the cutoff for noise such that ~1% of peaks that are fitted are statistically likely to be noise. Furthermore, FitSpec will determine how well a peak can fit the noise in the spectrum, and use this information to decide when combining peaks reduces over-fitting of the spectrum.

These methods of evaluating the noise for fitting are powerful in obtaining optimal fits of spectra. However, the main limitation here is that they avoid over-fitting only noise. Therefore, artifacts above the noise level will be fit. Poor baselines are a severe problem in this case. A constant offset will cause mis-evaluation of noise, and uneven baselines will make defining a good noise cutoff impossible. Additional problems will arise if the selected lineshape type (‘gauss’, ‘lorentz’, ‘mixXX’) is not a good match for the experimental data, so that FitSpec will add extra peaks to fit out the mismatch.

1.5.4. Notes on parallelization

INFOS uses parallelization for some calculations if a ‘Matlabpool’ is available, using the ‘parfor’ function. However, not all operations in INFOS are possible in parallel. Therefore, parallelization will usually speed up INFOS calculation, but for N cores, INFOS may be far from gaining the full Nx speed-up desired. This should be considered when running on shared clusters, or any system for which resources could be used more efficiently.

Specifically, the FitSpec function fits sub-spectra in parallel, but must reconstruct the fit and spectrum in between fitting iterations in series, and must also perform dynamic peak list editing in series. In contrast, the FitError function runs fully in parallel and is usually an efficient usage of parallelization. The FitTrace function runs initial fitting fully in parallel (see section 3.6.2), but the latter fitting of the full spectrum runs with the same limitations as the FitSpec function. Finally, one should note that fits for which many parameters are fixed, the

ratio of communication overhead to the cores versus time savings from parallelization can become unfavorable. Alternatively, if multiple spectra must be fit, using a ‘parfor’ loop around the fits is typically the more efficient usage of computational resources. See section 2.2.19 for details on controlling parallelization.

2. FitSpec Input and Output Structures

2.1. ‘spec’ structure: Spectrum, acquisition, and processing Information

The spectrum is stored in the ‘spec’ structure, in addition to all relevant acquisition and processing information. The user may also opt to store signal decay information in the ‘spec’ structure, but since this information is not obtained from the experimental data files, it is more typically stored in the ‘par’ structure. The following describes the fields so that the user may create the ‘spec’ structure, but it may also be quickly generated from Topspin files with the ‘getSpecBruker’ function, described in 3.1.1.

2.1.1. spec.fX

These vectors (X is the dimension number) give the frequency axis, in ppm, for each of the dimensions. The size is 1 X n, for a dimension with n points, and the points are in ascending order.

2.1.2. spec.S

This is an array containing the intensity data of the real part of the spectrum. The dimensionality of the matrix needs to match the dimensionality of the spectrum, and the array dimensions should correspond in size and order with the spec.fX vectors

2.1.3. spec.NucX

This is a string specifying the nucleus, typically in the form ‘1H’ or ‘15N’ for example. These fields are not necessary for spectrum fitting, but may later be useful for exporting results, or in faster labeling of spectrum plots.

2.1.4. spec.par

This field contains the fields ‘spec.par.dX’ for each dimension, which then contain relevant acquisition and processing information, and are further described in 2.3. They are optionally provided in the ‘spec.par’ parameter or in the ‘par’ parameter for the FitSpec function. This information may be omitted, in which case the program will default to fitting with pure

Gaussian lineshapes (or Lorentzian if specified), but better performance is expected if the information is provided.

2.1.5. spec.title

This field provides the title for the spectrum. This is not used for spectrum fitting, but is useful for labeling of plots, and will be exported if writing to the Topspin format.

2.2. 'par' structure: User instructions for FitSpec

The FitSpec function is capable of fully automated spectrum fitting, without additional user instructions aside from the spectrum itself. However, many situations will require some adjustments by the user to fully optimize the fitting. The 'par' structure gives the user control over most aspects of the spectrum fitting. The details of each field for user instructions is given here, although one should also note that the 'par' field is output by the FitSpec function as 'fit.par', and contains the settings either chosen by FitSpec or set by the user. This is useful when optimizing fitting parameters, as one can see the initial parameter settings determined by FitSpec, and then optimize those deemed to be suboptimal.

2.2.1. par.grid

The FitSpec function breaks the spectrum into sub-spectra, which are fitted individually, and then periodically recombined. The number of sub-spectra is determined by 'par.grid', which is a 1 X N vector, where N is the number of dimensions. Each element of 'par.grid' specifies how many parts that dimension should be separated into, so that the total number of sub-spectra is the product of the values in 'par.grid'. Large values in 'par.grid' will accelerate fitting, since the size of each sub-spectrum is reduced, but peaks that overlap strongly into other sections will be fit less accurately. Usually, default settings are sufficient, but spectra with peaks that are quite broad in comparison to the spectrum resolution may not fit well with the default values of 'par.grid'. The best remedy for this is to process with fewer data points, but if this is not possible due to a mix of broad and narrow peaks, then one may reduce the values of 'par.grid'.

3 entries for a 3D spectrum:

```
par.grid=[10 10 5];
```

2.2.2. par.IterFact

Each sub-spectrum is fitted separately, using a gradient-based minimization routine. The number of fitting iterations in this fitting routine is determined by taking the product of ‘par.IterFact’ with the number of fitting variables describing the sub-spectrum. Increasing ‘par.IterFact’ will improve fitting accuracy, but often returns are diminishing.

1 entry:

```
par.IterFact=3
```

2.2.3. par.add_peaks

When using a dynamic peak list, ‘par.add_peaks’ determines how many times the peak list will be edited. After initial peak picking, and fitting the initial peaks, the peak list is edited to try to optimize the fit. This is given by a single value in ‘par.add_peaks’, so that the total number of loops over the fitting step is ‘par.add_peaks’+1. Complicated spectra may benefit from additional peak addition steps, so ‘par.add_peaks’ may be increased. Also, if an initial fit is given to the program, then by default, FitSpec assumes a fixed peak list, so that ‘par.add_peaks’ is set to zero, although this will be overridden in the user sets ‘par.add_peaks’ to be non-zero.

For 3 steps of peak addition (dynamic peak list):

```
par.add_peaks=3;
```

2.2.4. par.n_iter

After each sub-spectrum is fitted, the fit is recombined to correct for peak overlap among sub-spectra, after which the fitting of sub-spectra is repeated. The number of repetitions is determined by ‘par.n_iter’. If a dynamic peak list is being used, then the iterative fitting is repeated after each peak addition/removal step. This means that the number of iterations must be specified at each step. Thus, the number of elements in ‘par.n_iter’ must be equal to ‘par.add_peaks’+1. As with ‘par.IterFact’, increasing the values in ‘par.n_iter’ will improve fitting, but improvements will be reduced as values get higher. However, because ‘par.n_iter’ changes the number of iterations that include corrections for peak overlap, increasing the value of ‘par.n_iter’ is often more beneficial than increasing the value of ‘par.IterFact’.

For ‘par.add_peaks=5’:

```
par.n_iter=[3 3 3 3 3 6];
```

2.2.5. par.cutoff

When using a dynamic peak list, the FitSpec function defines a particular peak height which, when exceeded, considers a peak as signal, and when below this height, is considered as noise. This definition varies, and is given one value for the initial peak picking, and then is typically decreased at the first several steps of peak list editing. Thus, par.cutoff must be a vector with ‘par.add_peaks’+2 elements. ‘par.cutoff’ is defined relative to the maximum peak height in the spectrum, so that one gives the fraction of the largest peak to be defined as the cutoff. The last value in the ‘par.cutoff’ vector indicates the cutoff for adding/removing peaks after the last fitting iteration. Since this is typically undesirable for fitting, the default for the last value is ‘NaN’ (not a number, which is an allowed value in MATLAB).

Note that during the program run, ‘par.cutoff’ must take on several different values, however, the user may specify a single value, in which case this will be set to the second to last value of ‘par.cutoff’ (the last value is NaN), and the other values will be calculated by FitSpec. Usually, this is the simplest method of setting the cutoff.

The FitSpec function will calculate the cutoff values by default, considering noise in the spectrum. The value is picked so that approximately 1% of the fitted peaks are noise. However, FitSpec is not able to take spectrum artifacts into account (pickup, incomplete phase cycling, residual water peak, etc.), and deviations of the experimental peak shapes from the calculated peak shapes are not considered. For this reason, the user may find that increasing ‘par.cutoff’ from the default value leads to fewer fittings of artifacts and irregular peak shapes. Also, different degrees of noise fitting may be desired, which can be adjusted directly with ‘par.cutoff’, or with the ‘par.n_noise_pks’ and ‘par.noise_frac’ fields.

For ‘par.add_peaks=5’:

```
par.cutoff=[.2 .1 .05 .05 .05 .05 NaN];
```

2.2.6. par.noise_frac, par.n_noise_pks

‘par.noise_frac’ and ‘par.n_noise_pks’ are indirect methods of setting the cutoff level. When either parameter is specified, the FitSpec function will determine a probability distribution of the noise peaks in the spectrum, from which it can estimate the number of noise peaks that will be above a given cutoff level. If ‘par.n_noise_pks’ is set, then this estimate can be used

to set the cutoff so that the expectation value of the number of noise peaks above the cutoff is approximately equal to ‘par.n_noise_pks’ (the exact expectation value is given in ‘fit.noise.n_fit_noise_pks’). If ‘par.noise_frac’ is set, then the cutoff is set by estimating the number of noise peaks and total number of peaks to be fitted, and approximates their ratio to be equal to ‘par.noise_frac’. Since it is not straightforward to estimate the total number of peaks that will be fitted, this will not be exactly correct, but the expectation value of the number of fitted noise peaks is still returned in ‘fit.noise.n_fit_noise_pks’.

For 0.5 % of peaks fit to be noise:

```
par.noise_frac=.005;
```

For 10 noise peaks to be fit:

```
par.n_noise_pks=10;
```

For no noise (approximately) to be fit:

```
par.n_noise_pks=0;
```

2.2.7. par.npp

‘par.npp’, which stands for ‘noise per peak’ controls when two peaks are combined into a single peak, during the peak addition/removal steps. Noise per peak indicates how much noise a single peak can fit, on average. Therefore, if having two peaks compared to one peak fails to reduce the error by at least the value given in ‘par.npp’, then the peaks will be combined, since this indicates that having two separate peaks is likely an overfit. Note that the input value of ‘par.npp’ is scaled relative to the maximum peak height. Therefore, the value is typically on the order of magnitude of ‘par.cutoff’. FitSpec will automatically calculate ‘par.npp’, but it may be manually input. Larger values will result in more frequency combination of peaks (fewer fitted peaks).

For a typical fitting (S/N~10):

```
par.npp=.08;
```

2.2.8. par.control

When using a dynamic peak list, each step of peak addition typically executes peak addition, subtraction, splitting, and combining. The exception is before the last fit iteration, only peak removal and combination are executed. However, these settings may be edited with the ‘par.control’ parameter. ‘par.control’ is a 4 x (par.add_peaks+1) sized array with logical values. After the n^{th} fitting, the n^{th} column determines which of the four peak editing

steps are performed. The rows are in the following order: add peaks, subtract peaks, split peaks, combine peaks (1=use method, 0=do not use method).

For ‘par.add_peaks’=4:

```
par.control=[1 1 1 0 0;1 1 1 1 0;1 1 1 0 0;1 1 1 1 0];
```

2.2.9. par.sign

By default, FitSpec assumes all peaks are positive. However, setting ‘par.sign’ to ‘+’, ‘–‘, or ‘+–‘ will change this to positive, negative, or positive and negative peaks.

For positive and negative peaks:

```
par.sign='+-';
```

2.2.10. par.noise

The FitSpec function analyzes the spectrum noise for the determination of the ‘par.cutoff’ and ‘par.npp’ parameters, and later returns the noise analysis. If ‘par.noise’ is not specified, then the entire spectrum is sampled for noise. To filter out real peaks, noise peaks are determined to be those which are concave up and negative if ‘par.sign’ is set to ‘+’, and the opposite if set to ‘–‘. If ‘par.sign’ is set to ‘+–‘, then all peaks are taken. However, ‘par.noise’ may be used to specify a region or regions of a spectrum that contain only noise. ‘par.noise’ is a vector, or cell of vectors if multiple regions are specified. Each vector consists of first the lower bound of the region in the first dimension, followed by the upper bound of the first dimension, and then followed similarly in subsequent dimensions.

For three noise regions, specified in a 2D spectrum:

```
par.noise={[75 120 75 120],[75 120 140 170],[140 170 75 120]};
```

2.2.11. par.verbose

By default, FitSpec outputs a status at the end of each fitting step. This may be suppressed using ‘par.verbose’.

For no output:

```
par.verbose=0;
```

2.2.12. par.rangeX

For each dimension, X, one may specify the range (in ppm) that the peak positions may vary. Using a peak list with initial peak positions ‘fit0.deltaX’, the peaks may then move only between ‘fit0.deltaX–par.rangeX/2’ and ‘fit0.deltaX+par.rangeX/2’. This is useful if the peak positions are approximately known, but some variability is need to optimize the fitting. One typically only uses this setting with fixed peak lists, although when used with a dynamic peak list, then this will restrict the peak positions relative to their initial placement. This usage may cause poor fitting, however, since the initial peak placement may be somewhat far from the correct position. ‘par.rangeX’ only accepts one argument.

For the 1st dimension:

```
par.range1=.5;
```

2.2.13. par.lw_rangeX

For each dimension, X, one may restrict the range (in ppm) that the linewidth may vary. If ‘par.lw_rangeX’ is specified, then the linewidth may vary approximately between ‘fit0.lwX–par.lw_rangeX/2’ and ‘fit0.lwX+par.lw_rangeX/2’. Note that this is only approximate, since the allowed linewidths are discrete values, and so FitSpec chooses the nearest linewidth. As with ‘par.rangeX’, this is typically used only with fixed peak lists, although if specified when using a dynamic peak list, it will restrict the linewidths relative to their initial values.

For the 2nd dimension

```
par.lw_range2=.25;
```

2.2.14. par.min_lwX, par.max_lwX

For each dimension, X, one may specify a maximum and/or a minimum linewidth (in ppm), using ‘par.min_lwX’ and ‘par.max_lwX’. This will affect the lineshape generation, so that lineshapes are only calculated between the values of ‘par.min_lwX’ and ‘par.max_lwX’, or if only one of these is specified, then a default value is used for the other. Note that specifying ‘par.max_lwX’ will also cause some peaks to be split after peak picking, if their initial linewidth is broader than the specified maximum value.

For the 1st dimension, restricting both minimum and maximum:

```
par.min_lw1=.25;
par.max_lw1=2;
```

2.2.15. par.fixed

If one is using a fixed peak list, it is possible to restrict which variables can be changed, by specifying ‘par.fixed’, or alternatively ‘fit0.fixed’ (both fields are equivalent). The ‘.fixed’ field is a logical with 2N+1 entries, N being the number of dimensions. The first entry specifies whether the peak amplitudes are fixed, the second and third entries specify whether to fix the peak positions and linewidths, respectively, of the first dimension, and subsequent dimensions follow.

For a 2D spectrum, with the amplitudes and linewidths variable, but positions fixed:

```
par.fixed=[0 1 0 1 0];
```

[Amplitude, Position 1, Linewidth 1, Position 2, Linewidth 2]

2.2.16. par.accur

‘par.accur’ specifies the accuracy for which peak shapes are calculated. Specifically, when lineshapes are initially calculated, if the ratio of the value at a particular position and the maximum of the lineshape falls below ‘par.accur’, then the value at that position is set to zero. Therefore, the closer ‘par.accur’ is to zero, the fewer truncation artifacts. Defaults are 1% of the minimum of ‘par.cutoff’, when using a dynamic peak list, or 1% of the ratio of the minimum initial peak height and maximum initial peak height, when using a fixed peak list. Higher values of ‘par.accur’ will accelerate calculation of full spectra in the FitSpec function, although eventually may lead to significant artifacts in the calculated spectra.

For no lineshape truncation:

```
par.accur=0;
```

2.2.17. par.setup_only

‘par.setup_only’ accepts a string, ‘y’, or ‘n’. When set to ‘y’, the FitSpec function will not fit the given spectrum, but rather will only determine the parameters in the ‘par’ structure and perform the initial peak pick/linewidth measurement. This is useful if one wants to see how the fit will be setup before running the entire fitting routine.

For setup only:

```
par.setup_only='y';
```

2.2.18. par.noise_eval

By default, the FitSpec function only performs noise analysis (see section 2.4.2) if it is necessary for setting the fitting parameters (‘par.cutoff’ and ‘par.npp’). Therefore, if a fixed peak list is used, or the user specifies both ‘par.npp’ and ‘par.cutoff’, then FitSpec does not perform noise analysis. However, if ‘par.noise_eval’ is set to ‘y’, then analysis will be performed anyway.

For noise evaluation:

```
par.noise_eval='y';
```

2.2.19. par.parallel

INFOS will automatically use parallelization if a ‘Matlabpool’ is available (via the ‘parfor’ function). It will not start the pool automatically. The user may override the automatic settings by setting ‘par.parallel’ to ‘y’ or ‘n’. This will either force INFOS to use a ‘parfor’ loop or a ‘for’ loop, respectively.

To disable parallelization

```
par.parallel='n';
```

2.3. d1, …, dn sub-structures: Lineshape information

For each dimension, the lineshapes are specified by supplying acquisition and processing information, and also by specifying the type of signal decay (Lorentzian and Gaussian, for example). This information is provided, for dimension X, in the substructures ‘spec.par.dX’, or ‘par.dX’. Both locations may be used simultaneously, although if a parameter is specified twice, then the value in ‘par.dX’ takes priority. Typically, one specifies the acquisition and processing information in ‘spec.par.dX’, since this information is known for the particular spectrum and will not change, and information on signal decay in the ‘par.dX’ structure, since one may want to test different values here to improve fitting.

2.3.1. Acquisition and processing information

This information is loaded by the ‘getSpecBruker’ function, so usually does not need to be specified by the user, but other formats may require manual specification.

dX.WDW: Specifies the window function. May be a string or number. See 2.3.3.

dX.SSB: Sinebell broadening parameter.

dX.GB: Gaussian broading parameter.

dX.LB: Lorentzian broadening parameter, in Hz.

dX.AQ: Processed acquisition time, in *s*.

dX.SI: Processed size before spectrum truncation.

dX.SIp: Processed size after spectrum truncation.

dX.TD: Number of processed points in time domain (real+imaginary).

dX.SF: Spectrometer frequency of dimension, in MHz.

dX.SWH: Processed sweep width, given in Hz.

dX.SW: Processed sweep width, given in ppm.

dX.user: Vector providing user-defined apodization functions (dX.TD/2 points)

dX.PHC1: Linear phase correction applied, in degrees.

dX.LPC: Compensate for linear phase correction in fitting ('y' is default)

2.3.2. Signal decay information

Aside from acquisition and processing information, lineshapes are determined by how the signal decays. Typically, one chooses either Lorentzian (exponential decay), or Gaussian (Gaussian decay) lineshapes. However, advanced options allow adding a fixed amount of Lorentzian or Gaussian character, or a fixed mixture or Lorentzian and Gaussian decay.

dX.Broad: 'dX.Broad' is a string, specifying the type of decay which is varied during the fitting. This is set to 'lorentz', 'gauss', or 'mixXX'. The first two options correspond to pure Lorentzian or Gaussian decay, respectively. 'mixXX' gives a fixed mixture of Gaussian and Lorentzian and decay, such that the fraction of Gaussian linewidth is **/100 and the fraction of Lorentzian linewidth is 1–XX/100 (any number of digits may be used, so that the fraction is given by 0.XX…).

dX.gauss0: If 'dX.gauss0' is specified, a fixed amount of Gaussian broadening (in ppm) is added to all lineshapes. The output linewidth ('fit.lwX') does not include the fixed amount of Gaussian broadening, although the FWHM of each output peak ('fit.FWHMX') must include all contributions to the linewidth.

dX.lorentz0: If 'dX.lorentz0' is specified, a fixed amount of Lorentzian broadening (in ppm) is added to all lineshapes. As with 'dX.gauss0', the output linewidth ('fit.lwX') does not include this fixed amount of Lorentzian broadening, but it is included in the FWHM ('fit.FWHMX').

2.3.3. Apodization function definitions

The following defines the apodization functions used in the INFOS software. These are setup to match the definitions used by Bruker Topspin. For each dimension, X, the variable 't' runs from 0 to 'dX.AQ', with 'dX.TD' points in between. The type of apodization function is specified in the 'par.dX.WDW' or 'spec.par.dX.WDW' field, using an index or a string, which are given below, along with the function definition. The other parameters (LB, GB, AQ, etc.) are also provided in either 'par.dX' or 'spec.par.dX'.

No Apodization:

`dX.WDW=0; dx.WDW='no';`

$$f_{apod}(t) = 1$$

Exponential:

`dX.WDW=1; dX.WDW='em';`

$$f_{apod}(t) = \exp\left(-\pi * LB * t\right)$$

Gaussian:

`dX.WDW=2; dX.WDW='gm';`

$$f_{apod}(t) = \exp\left(-\pi * LB * t + \frac{\pi * LB * t^2}{2 * GB * AQ}\right)$$

Sine:

`dX.WDW=3; dX.WDW='sine';`

$$SSB \geq 2$$

$$f_{apod}(t) = \sin\left(\pi\left(1 - \frac{1}{SSB}\right)\frac{t}{AQ} + \frac{\pi}{SSB}\right)$$

$$SSB < 2$$

$$f_{apod}(t) = \sin\left(\pi\frac{t}{AQ}\right)$$

Squared Sine:

`dX.WDW=4; dX.WDW='qsine';`

$$SSB \geq 2$$

$$f_{apod}(t) = \sin^2\left(\pi\left(1 - \frac{1}{SSB}\right)\frac{t}{AQ} + \frac{\pi}{SSB}\right)$$

$$SSB < 2$$

$$f_{apod}(t) = \sin^2\left(\pi\frac{t}{AQ}\right)$$

Sinc:

`dX.WDW=7; dX.WDW='sinc';`

$$f_{apod}(t) = \sin\left(2\pi * SSB\left(\frac{t}{AQ} - GB\right)\right)$$

Squared Sinc:

`dX.WDW=8; dX=WDW='qsinc';`

$$f_{apod}(t) = \sin^2\left(2\pi * SSB\left(\frac{t}{AQ} - GB\right)\right)$$

User defined apodization function:

`dX.WDW='user';`

$$f_{apod} = \mathrm{dX.apod}$$

Note that dX.apod must have the correct length, based on the acquisition time and sweep width of the dimension

2.3.4. Linear phase compensation

INFOS will calculate lineshapes that incorporate distortions due to delay in acquisition, followed by linear phasing to correct for those delays. This is activated by specifying the size of the linear phase correction (in degrees, see 2.3.1), but can be deactivated by setting 'par.dX.LPC' to 'n' or 0. The delay is calculated as

$$t_{delay} = -\frac{PHC1/360}{SWH0}$$

where 'PHC1' is the provided phase (difference of phase from left to right side of spectrum before truncation), and 'SWH0' is the sweep width of the spectrum (in Hz, before spectrum truncation). Note that this can slow down fitting of complicated spectra considerably.

2.4. 'fit' structure: Output of FitSpec

Upon completion of the fitting routine, the FitSpec function returns a structured variable, 'fit'. This contains parameters for fitting each peak, and information on the fitting settings and results. The individual fields are detailed here.

2.4.1. Peak parameters

The following are the parameters that describe each peak in the spectrum.

fit.I: Peak amplitudes, given in arbitrary units.

fit.deltaX: Peak positions in dimension X, given in ppm.

fit.lwX: Linewidths in dimension X, given in ppm. Note that this is the unprocessed linewidth; for example, for a spectrum fitted with Lorentzian broadening, the value returned in 'fit.lwX' will be $1/(\pi T_2)$ where T_2 is the time constant describing the signal decay before apodization. Therefore, the fitted values in 'fit.lwX' are, in principle, independent of the acquisition and processing settings. In practice, the accuracy of 'fit.lwX' values can be affected by acquisition and processing settings, and also will be more accurate if the settings for signal decay are well matched to the real type of signal decay.

fit.FWHMX: Full width at half maximum for each peak in dimension X, given in ppm. This is the processed linewidth, and includes broadening that results from acquisition and processing settings. Therefore, for isolated peaks, the values returned in 'fit.FWHMX' should approximately match the measured linewidth, whereas the values returned in 'fit.lwX' often will differ from the measured linewidth, depending on acquisition and processing settings.

fit.lw_indX: Since lineshapes are pre-calculated, and then recalled for fitting, each linewidth corresponds to a particular index. This index then gives the position in the 'shapes0' structure (section 2.6) to find the particular lineshape. This can be useful for accelerating spectrum calculation if the 'shapes0' structure is stored, and is used by the FitEditor2D program (3.6.1), although this parameter does not have relevance outside the INFOS software.

fit.int: Integrals of each peak. Note that this value is simply the sum of intensities of each peak across the spectrum. This is opposed to being a boxed integral- since it is a fit, it is possible to include the entire peak volume without having to consider. Note that the volume is not scaled by Δf_X (resolution in each dimension).

2.4.2. Noise analysis and error evaluation

The FitSpec algorithm uses noise analysis for setting some of the fitting parameters ('par.cutoff' and 'par.npp'). If noise analysis is performed, either to determine these parameters or because the user sets 'par.noise_eval' to 'y', then the result is returned in 'fit.noise', and also a calculation of the reduced-χ^2 of the spectrum fit is returned. Noise analysis consists of first generating synthetic noise, and determining a probability

distribution of the heights of noise peaks, and secondarily, from fitting that distribution to experimental peaks. The fields in ‘fit.noise’ are summarized below.

noise.x: This is the x-axis for the calculated probability distribution of peak heights, such that the values in ‘noise.x’ are peak amplitudes. The values in ‘noise.x’ have been scaled to be a best fit to the experimental peaks. Note that this distribution only includes concave down noise peaks, although the concave up peaks would have an x-axis equal to –noise.x.

noise.fx: This is the probability density corresponding to each value in ‘noise.x’. The distribution is normalized such that the values in ‘noise.fx’ add up to one.

noise.x_exp: This is the x-axis for the experimental noise peaks, which have been grouped using a histogram (‘histc’ function). As with ‘noise.x’, this then corresponds to experimental noise peak amplitudes.

noise.count_exp:

This gives the number of experimental peaks at amplitude given in ‘noise.x_exp’. Note that the total count has been scaled so that the values in ‘noise.fx’ and ‘noise.count_exp’ correspond. The un-scaled count can be obtained with the following:

```
noise.count_exp*noise.np_exp/sum(noise.count_exp)
```

noise.np: This is the number of synthetic noise peaks measured. This number includes both positive and negative (concave up and concave down) noise peaks. It is a good estimate for the number of noise peaks expected in the experimental spectrum, not considering that real peaks will sit on top of the noise peaks.

noise.np_exp:

This is the number of experimental peaks considered in the noise analysis. This will vary considerably, depending on whether regions of noise were specified, and on the ‘par.sign’ setting.

noise.rms: This is the root mean square of the amplitude of spectrum noise. It is determined from the fitting of synthetic noise to experimental noise. Therefore, it should not be heavily biased by artifacts or real peaks, even if no noise region has been specified.

noise.noise: This is the root mean square of noise peaks, and so is higher than the value of noise.rms, since it is selective for local maxima. It is also obtained by fitting synthetic noise to experimental noise.

noise.n_fit_noise_pks:

This is an estimate of the number of noise peaks that will be above 'par.cutoff' in the final peak picking step. This is based on the fitting of the synthetic noise distribution to the experimental noise and the number of synthetic noise peaks measured ('noise.np'). If FitSpec is calculating the value of 'par.cutoff', then it attempts to set this parameter to a particular amount (see 'par.noise_frac' and 'par.n_noise_pks' in 2.2.6).

The synthetic noise distribution and its fit to the experimental noise can easily by compared using the following:

```
bar(fit.noise.x_exp,fit.noise.count_exp);
hold all
plot(fit.noise.x,fit.noise.fx)
```

One usually observes differences between the synthetic and experimental noise. Partly, this is due to the resolution of the histogram of the experimental noise, but also occurs because the experimental noise also usually includes various spectrum artifacts, that are not produced in the synthetic noise distribution. Due to the fitting of synthetic to experimental noise, however, these artifacts have very little effect on noise characterization.

Aside from the 'noise' structure, the following fields are also reported in the output structure, 'fit'.

fit.chi_red: The reduced-χ^2 is returned if noise analysis is performed. The value returned for 'fit.chi_red' is calculated according to:

$$\chi^2_{red} = \frac{1}{N - n_{fit}} \sum_{i=1}^{N} \frac{\left(I_i^{exp} - I_i^{calc}\right)^2}{rms^2}$$

Here, rms is the value returned in 'fit.noise.rms' (see 2.4.2). I_i^{exp} and I_i^{calc} are the intensities at each point in the spectrum of the experimental and calculated spectra, respectively. N is the number of

points in the spectrum, and n is the total number of fit variables being used to the fit the spectrum.

fit.resnorm: 'fit.resnorm' returns the value of the squared 2-norm of the fit residual, as given by

$$\sqrt{\sum_{i=1}^{N}\left(I_i^{exp}-I_i^{calc}\right)^2},$$

where the parameters are the same as those for 'fit.chi_red'.

2.4.3. Other outputs

The FitSpec function includes several other useful outputs in the 'fit' structure, which are detailed here.

fit.par: This is the full 'par' structure which is used internally by the FitSpec function. This may be used to examine how fitting was done, and may be used as an input to another fitting, in order to replicate all fitting conditions (this includes acquisition and processing parameters- remove par.d1…par.dN if spectra with different processing and acquisition are being used).

fit.spec: This is the best-fit calculated spectrum determined at the end of the program. This includes all of the fields described in section 2.1, and can be treated as a normal spectrum.

fit.resid: This is the difference spectrum between the experimental spectrum and the calculated spectrum. It also includes all fields from section 2.1, and can be treated as a spectrum.

fit.fixed, fit.rangeX, fit.LI_rangeX:

Restrictions on the fitting variables are returned in the 'fit' structure (excluding the 'max_lwX' and 'min_lwX' settings, which are returned in 'fit.par'). This includes 'fit.fixed' (2.2.15), 'fit.rangeX' (2.2.12) and 'fit.lw_rangeX' (2.2.13). The latter two entries are specified for each peak, with a lower and upper bound. 'fit.LI_rangeX' is the range for the index of the linewidth, and therefore does not give the actual range of the linewidths.

2.5. ‘fit0’ structure: Initial fits and controlling fit variables

The ‘fit0’ variable is input to give the FitSpec program an initial guess for the spectrum fit, which will then be refined. By default, if ‘fit0’ is provided, FitSpec will use a fixed peak list. However, the user may override this by setting ‘par.add_peaks’ to a non-zero value. One may also provide various restrictions on how the fitting parameters may be changed by the program, including ranges and fixing of variables.

2.5.1. Initial guess

The inputs for an initial guess are as follows, where the dimensions of all inputs are (*np* x 1), np being the number of peaks in the initial guess. Note that the only required fields for the initial guess are the ‘fit0.deltaX’ fields. INFOS will estimate an initial value for the other fields if omitted. See section 2.4.1, as the input ‘fit0’ variable has the same parameter format as the FitSpec output.

fit0.deltaX: Peak positions (required)

fit0.I: Peak amplitudes

Only one of the following is used:

fit0.lwX: Peak linewdith

fit0.FWHMX: Peak FWHM

fit0.lw_indX: Peak index. This may only be used if the ‘shapes0’ variable is provided (see 1.3.5 and 2.6)

2.5.2. Fitting restrictions

Fitting restrictions may be applied via the ‘fit0’ variable, in a similar manner as is done with the ‘par’ variable (see 2.2.12, 2.2.13, and 2.2.15). ‘fit0’ allows one to fix each variable type separately (intensity, and position and linewidth in each dimension), or to restrict ranges on the position and linewidth for each peak individually. Note that defining minimum and maximum linewidths must be done using the ‘par’ variable (see 2.2.14).

fit0.fixed: Same as ‘par.fixed’ (see 2.2.15), allows fixing of all amplitudes, positions in each dimension, and/or linewidths in each dimension.

*fit0.*rangeX: Range for the peak position in dimension X. This can be given as a single argument to restrict the positions with respect to their initial positions, as is done using ‘par.rangeX’ (see 2.2.12). Alternatively, one may specify an N x 2 sized vector, where N is the number of peaks, so

that each row specifies the lower and upper bound of the corresponding position.

fit0.lw_rangeX:

Range for the peak linewidth in dimension X. This can be give as a single argument to restrict the linewidths with respect to their initial linewidths, as is done using 'par.lw_rangeX' (see 2.2.13). Alternatively, one may specify an N x 2 sized vector, where N is the number of peaks, so that each row specifies the lower and upper bound of the corresponding linewidth.

2.6. 'shapes0' cell: User specified lineshapes

The FitSpec function uses pre-calculated lineshapes to perform calculation of the spectra. This allows fast calculation, and therefore efficient spectrum fitting. The pre-calculated lineshapes are stored internally in the 'shapes0' variable in FitSpec, and may be obtained by the user by setting a second output argument when calling the FitSpec function.

```
[fit shapes0]=FitSpec(spec,par,…);
```

If one is fitting multiple spectra that are acquired and processed under the same conditions (and have the same type of signal decay, see 2.3.2), then the 'shapes0' cell may be recycled. Then, one may enter 'shapes0' as a fourth argument to the FitSpec function.

```
fit=FitSpec(spec,par,fit0,shapes0);
```

Note that both 'par' and 'fit0' may be replaced by '[ ]' if the user wants to omit these arguments.

Advanced users may also want to define their own lineshapes to be used in spectrum fitting. This will cause FitSpec to ignore all specifications of acquisition and processing information, as well as signal decay (see section 2.3), and simply use the specified shapes. This requires generating the 'shapes0' structure independently and providing it as an input to the FitSpec function. The requirements are described here.

The 'shapes0' cell is a 1 x N cell, where N is the number of dimensions in the spectrum. Then, each element of the 'shapes0' cell corresponds to a dimension ('shapes0{1}' corresponds to dimension 1, 'shapes0{2}' to dimension 2, and so on). Each element of the 'shapes0' cell then must have the fields that are described below. These give the actual lineshapes, and also the linewidths and frequency axis for each dimension.

shapes0{X}.f:

This is the frequency axis of the shapes structure for dimension X. If dimension X of the spectrum being fitted has a size of 1 x N points, then this axis must have a size of 1 x 2N+8 points. Secondarily, the spacing between points must match that of the original spectrum dimension, and the values must be ascending. Finally, the point found in the position N+5 must be zero.

shapes0{X}.f0:

This is an abbreviated form of 'shapes0{X}.f', and always has three entries (1 x 3). The first entry is the first value in 'shapes0{X}.f' (the lowest value). The second entry is the spacing between points in 'shapes0{X}.f'. Finally, the last argument is the length of 'shapes0{x}.f'. To calculate the values in MATLAB:

```
shapes0{X}.f0=...
[shapes0{X}.f(1) diff(shapes0{X}.f([1 2])) ...
length(shapes0{X}.f)];
```

shapes0{X}.lw:

This is a 1 x N vector of linewidths, where N is the number of different lineshapes specified. In principle, the user may define this list however desired. Typically, however, it is specified with equally spaced values from some minimum to some maximum linewidth, and should have some relationship to the breadth of the line. Note that values returned in 'fit.lwX' and values used 'par.lw_rangeX' or 'fit0.lw_rangeX' refer to this vector.

shapes0{X}.lw0:

This is an abbreviated form of 'shapes0{X}.lw0'. The first value is the initial value in 'shapes0{X}.lw', the second is the spacing between values in 'shapes0{X}.lw', and the third is the length of 'shapes0{X}.lw'. In fact, FitSpec only uses the third value, so if values in 'shapes0{X}.lw' are not equally spaced, one may simply place a zero in the second entry.

shapes0{X}.FWHM:

This is a 1 x N vector of FWHM values, where N is the number of different lineshapes specified. Typically, this should give the 'full width at half maximum' of each defined lineshape. Values returned in

'fit.FWHMX' refer to this vector, and also this vector is used when an initial guess is made at the breadth of a peak.

shapes0{X}.shape:

This array stores the actual lineshapes. Each column corresponds to a different linewidth, so that if there are M elements in 'shapes0{X}.f' and N elements in 'shapes0{X}.lw', then this should be an M x N array. For each lineshape, the maximum is typically set to be found at the zero position in 'shapes0{X}.f'. Also, each lineshape should be normalized so that its maximum is one.

When generating the 'shapes0' cell, one should take consideration of how the different elements are used by FitSpec. When using a dynamic peak list, FitSpec needs to make initial guesses as to where peaks should be placed, and what their linewidths and amplitudes should be. In order to do this, FitSpec finds a local maximum, and estimates its FWHM and amplitude. It then places a peak from the 'shapes0' cell, which has a value in 'shapes0{X}.FWHM' that is closest to the measured value, and also aligns the zero position of the 'shapes0{X}.f' to the measured peak position. Finally, it sets the amplitude equal to the measured amplitude, which requires normalized peak shapes. Therefore, if one uses shapes that do not have their maximum at the center of the peak (not found at 'shapes0{X}.f=0'), then initial guesses will be inaccurate. Similarly, if the values in 'shapes0{X}.FWHM' are not corresponding to the measured width, then initial values will be inaccurate.

These restrictions are lifted, however, if a fixed peak list is used. Then, FitSpec does not need to make initial guesses at the peak positions, so the peak maximum does not need to correspond to the peak center, and the FWHM does not need to be well defined (it still must be included in 'shapes0', but will not effect fitting quality). This allows one to use shapes that might be split into more than one local maximum, or other irregular shapes. Note that gradient based fitting will fail if changes between adjacent lineshapes are not somewhat continuous.

2.7. 'shapes' cell: Recycling lineshapes for similar spectra

The 'shapes' cell is used to build sub-spectra in the FitSpec routine. It is therefore calculated from 'shapes0', and is expanded into all dimensions. For spectra with the same acquisition and processing parameters, and the same type of signal decay (see 2.3), one may recycle the 'shapes' cell to accelerate spectrum fitting (primarily useful when setup and

actual fitting taking similar lengths of time). The content of the 'shapes' cell, however, is not described here, as correct calculation of the cell is difficult, and unnecessary for the user. One may obtain the 'shapes' cell, however, by setting three output arguments for the FitSpec function.

```
[fit shapes0 shapes]=FitSpec(spec,…);
```

The 'shapes' cell may then be input for another fitting, but must be used in conjunction with the 'shapes0' cell.

```
fit=FitSpec(spec,par,fit0,shapes0,shapes);
```

As always, one may omit the 'par' and 'fit0' variables by replacing either with '[ ]'.

3. Supplementary Programs

3.1. Data import and export

Currently, spectrum import and export is supported for the Bruker TopSpin and NMRPipe format. Export capability in this format also allows loading spectra into the CCPN software. The XEasy format for peak lists exporting is also supported.

3.1.1. getSpecBruker

This program loads spectra from the Bruker TopSpin format, in addition to all relevant acquisition and processing parameters, according to the definitions given in section 2.1. It should be noted that not all parameter definitions in Topspin exactly match those used in the INFOS software. 'getSpecBruker' will fix the discrepencies, however. 'getSpecBruker' may be used by simply specifying the data location, but also has several options. The following format is used for loading spectra:

Using all defaults:

```
spec=getSpecBruker(location);
```

Note that 'location' specifies the location a folder- either the acquisition folder, or a specific processing folder- but not the data file itself (not the 2rr file, for example).

By specifying a particular range of the spectrum:

```
spec=getSpecBruker(location,range);
```

or by specifying several options:

```
spec=getSpecBruker(location,opt);
```

Here, one may simply load the spectrum as is, truncate the spectrum according to the argument ‘range’, or specify several options using the structure ‘opt’. The arguments for ‘getSpecBruker’ are detailed here.

location: One may specify the pathway to the folder containing the ‘pdata’ folder, in which case the user may specify the processing folder number in ‘opt’, or the processing folder ‘1’ is used if unspecified. Alternatively, one may specify the full path to the processed data folder, in which case any processing number is ignored.

range: List of the lower and upper bounds of each frequency axis. Specified as follows:

```
range=[LB1 UB1 LB2 UB2 ...];
```

where ‘LBX’, ‘UBX’ are the lower and upper bounds for each dimension.

opt.range: Same as ‘range’ argument.

opt.proc_no: Specify the processing number, as a numeric argument (not a string). Only used if full pathway to processing folder is not given.

opt.phase: Set to ‘y’ or ‘n’. If set to ‘y’, then complex data will be loaded in addition to the real valued data. In the case of 1D spectra, ‘spec.S’ will be complex valued. For 2D spectra, ‘spec.S’ will be real valued, but additional fields ‘spec.Sri’, ‘spec.Sir’, and ‘spec.Sii’ will be loaded to contain the hyper-complex data. For 3D spectra, only the complex data in the direct dimension will be loaded, in ‘spec.Srri’. Higher dimensional complex data is not currently supported.

3.1.2. getSpecPipe

This program loads spectra from NMRPipe (Delaglio et al. **J. Biomol. NMR,** 6(3)*, 277*), in addition to relevant information on processing. Arguments are the data location and the range of the spectrum to be loaded.

location: The pathway to the processed data must be specified. If data is stored in multiple files, this argument may be given with a wildcard. For example, if files are test001.ft3, test002.ft3, etc., then the location should be given as test*.ft3.

range: List of the lower and upper bounds of each frequency axis. Specified as follows:

```
range=[LB1 UB1 LB2 UB2 ...];
```

where ‘LBX’, ‘UBX’ are the lower and upper bounds for each dimension.

Using all defaults:

```
spec=getSpecBruker(location);
```

Specifying a range

```
spec=getSpecBruker(location,range);
```

3.1.3. spec2Bruker

‘spec2Bruker’ writes spectra in the INFOS spectrum format into the TopSpin format. This may be used to view both real spectra and calculated spectra in other software, in particular Bruker TopSpin and CCPN. Note that if a spectrum has been loaded into MATLAB with getSpecBruker, and then is later exported back into TopSpin, only the acquisition and processing parameters used by INFOS will be written in the new file, whereas other parameters may take on arbitrary values. Input format is as follows:

```
spec2Bruker(directory,spec);
```

If the directory to be written to already exists, and the user wishes to overwrite it *without* being prompted (normally, a warning prompt is given)

```
spec2Bruker(directory,spec,’overwrite’);
```

Note that the folder created will contain both acquisition and processing files, so that it can be opened in TopSpin without generating errors. However, only the parameters used by INFOS will be set correctly in these files. ‘spec2Bruker’ will not write imaginary data.

3.1.4. XEasy_write

‘XEasy_write’ generates a peak list file from the ‘fit’ structure for exporting peaks into other programs, using the *XEasy* format. Usage is as follows, where path is the location for which the file will be saved, and ‘fit’ is the usual spectrum fitting variable.

```
XEasy_write(path,fit);
```

To change the order of dimensions for the output file, one may specify an output ordering as a vector with the desired dimension order For example, ‘order=[1 3 2]’ will output the first dimension of the fit first, the third dimension of the fit second, and the second dimension of the fit third.

```
XEasy_write(path,fit,order);
```

Finally, as with ‘spec2Bruker’, one may overwrite existing files without being prompted by specifying

```
XEasy_write(path,fit,order,’overwrite’);
```

Here, ‘order’ may be replaced with ‘[ ]’, which will then use the default ordering.

3.2. Peak picking and linewidth measurement

INFOS provides very basic programs for quickly searching for peaks and for measuring linewidths. This includes the ‘peaks_nD’ program, and the ‘FWHM_nD’ program.

3.2.1. peaks_nD

‘peaks_nD’ returns all local maxima and/or minima above a given threshold. This threshold is given as a variable, ‘cutoff’, which is the fraction of the maximum (or minimum) of the spectrum that should be returned in the peak list. For example, if ‘par.cutoff=.05’, and the spectrum maximum is 15000, then all peaks with amplitudes above 750 will be returned. ‘peaks_nD’, by default, will only return positive peaks, but one may specify the ‘sign’ as ‘+’, ‘–‘, or ‘+–‘ which will return only positive, only negative, or both. These options are specified in the ‘par’ structure. ‘peaks_nD’ then returns a list of peak positions (one for each dimension), a list of intensities at each peak position, and a logical index with the same dimensions as the spectrum, with ‘true’ at each peak position.

par.cutoff: This is multiplied by the spectrum maximum/minimum to determine the peak threshold. Note that if searching for negative peaks, one still uses a positive value for ‘par.cutoff’. The exception is if one wants all concave down peaks (upward pointing peaks) that have amplitudes greater than some *negative* value, then one must specify a negative

cutoff (one may also do the same for concave up peaks with amplitudes less than some positive value).

par.sign: Specifies the sign of the peaks to be searched for. Given as a string, either '+', '–', or '+–'.

Note that to return *all* peaks of a given sign, one should specify 'par.cutoff=-Inf', rather than 'par.cutoff=0', since for example, some positive peaks (concave down) may still have negative amplitudes.

3.2.2. FWHM_nD

'FWHM_nD' returns an estimation of the full width at half maximum of a given peak list, which may be generated with 'peaks_nD'. For each peak, 'FWHM_nD' measures the peak height, and then moves away from the peak in each dimension and in both directions, until it finds a point nearest to half the amplitude of the peak. For a given peak and dimension, if the distance in both directions is within 20% of each other, then the distance between these two points is given as the FWHM. However, if one distance is more than 20% larger than the other, then the FWHM is given as 2x the smaller distance. This attempts to account for some distortions due to peak overlap to one side of a peak. Note that no linear prediction is used to attempt to account for linewidth error from spectrum digitization. Therefore, this program should only be used for initial estimates, even for well-isolated peaks. Usage is as follows, with 'peaks_nD' being used to generate the initial peak list.

```
par.cutoff=.05;
par.sign='+';
pks=peaks_nD(spec,par)
FWHM=FWHM_nD(spec,pks);
```

The output then includes a 'FWHM' for each dimension and each peak, and also returns the amplitude of each peak.

3.3. Spectrum manipulation

The following programs are used for manipulating spectra (truncating, calculating slices, etc.), and will maintain the correct format of the 'spec' structure for use in other programs in the INFOS package. All programs work on spectra of arbitrary dimensionality.

3.3.1. clip_spec_nD

The ‘clip_spec_nD’ program allows one to truncate a spectrum to a particular region. This is called by providing the initial spectrum, followed by a range. The ‘range’ variable is a vector with 2 entries for each dimension, order as [LB1 UB1 LB2 UB2 …], where LBX and UBX are the lower and upper bounds in ppm for each dimension. ‘clip_spec_nD’ is called as follows:

```
spec=clip_spec_nD(spec0,range);
```

3.3.2. proj_nD

The ‘proj_nD’ function performs a projection across one or more dimensions. One specifies the dimension(s) to be removed via projection in a vector, ‘dim’, and optionally one may provide the range of the spectrum to be projected over (2 entries, LB and UB, for each dimension to be projected over). If the ‘range’ variable is omitted, then the projections are performed across the entire dimension. Alternatively, proj_nD may be used to obtain slices, in which case one provides only one value per dimension in the ‘range’ argument. This will return the slice nearest to the given frequency position.

To obtain a 2D projection across the whole 2nd dimension of a 3D spectrum:

```
spec2D=proj_nD(spec3D,2);
```

To obtain a 1D projection across part of the 1st and 2nd dimensions of a 3D spectrum:

```
spec1D=proj_nD(spec3D,[1 2],[40 70 100 105]);
```

To obtain a 1D slice from particular positions in the 2nd and 3rd dimensions of a 3D spectrum

```
spec1D=proj_nD(spec3D,[2 3],[50 103]);
```

3.3.3. slice_nD

The ‘slice_nD’ function extracts slices from one or more dimensions in a spectrum. It utilizes linear prediction unless the slice to be extracted exactly matches a frequency position in the original spectrum. To avoid using linear prediction, use ‘proj_nD’ with one frequency position per dimension. ‘slice_nD’ requires an argument specifying the dimension(s) to extract slices from, and a second argument specifying the frequency position(s) at which to extract the slices.

To obtain a 1D slice from a 3D spectrum, at positions in the 1st and 3rd dimensions:

```
spec1D=slice_nD(spec3D,[1 3],[45 110]);
```

3.3.4. add_spec_nD

The ‘add_spec_nD’ function sums a series of spectra together. If the spectra do not have exactly the same set of frequency axes, then this function will calculate a common set of axes and linearly extrapolate the amplitudes in the individual spectra to obtain the amplitude for the common set of axes before adding the spectra together. Acquisition and processing parameters will be taken from the first spectrum in the series- note that adding spectra together with different acquisition and processing parameters will lead to less reliable fitting. A weight may also be included.

To obtain a sum of spectra stored in a cell

```
spec_sum=add_spec_nD(spec_cell{:});
```

To take the difference of two spectra, using a weight argument

```
spec_diff=add_spec_nD(spec1,spec2,[1 -1]);
```

3.3.5. combine_specs

The ‘combine_specs’ function takes a series of spectra and combines them into a single spectrum, by adding a dimension to the spectra. The additional dimension is given a frequency axis that is simply numbered from 1 to the number of spectra. Useful for plotting a series of 2D spectra as a 3D plot.

```
spec=combine_specs(spec1,spec2,...);
```

3.3.6. baseline_corr

This ‘baseline_corr’ function measures the average amplitude of a region (or regions) which are specified as baseline regions by the user, and then corrects the spectrum so that the average amplitude of these regions is zero. Having baseline corrected spectra is important for noise determination before fitting, although note that poor baseline is often indicative of larger processing or acquisition problems.

To baseline correct a 2D spectrum, while specifying two baseline regions

```
spec=baseline_corr(spec0,{[80 120 80 120],[80 100 140 160]});
```

3.4. Spectrum generation

INFOS provides two programs for calculating spectra outside of the FitSpec function. The first is for calculating spectra from a list of peak positions, linewidths, and intensities, and

the second program is for generating spectrum noise according to processing and acquisition parameters.

3.4.1. FullSpecCalc

The ‘FullSpecCalc’ function generates spectra from a list of peak positions, linewidths, and intensities for arbitrary spectrum dimensions. This information is provided in a structure, ‘spec_par’. Then, ‘spec_par’ must have fields ‘spec_par.I’ to specify the intensity, and for each dimension, X, it must have the field ‘spec_par.deltaX’ to specify the position and either ‘spec_par.lwX’ or ‘spec_par.FWHMX’ to specify the linewidth. These fields follows the same formatting as in the ‘fit’ variable (section 2.4.1). One must also provide processing and acquisition information, and signal decay information (see section 2.3). This is provided in a ‘par’ variable, which contains fields ‘par.dX’ for each dimension, X (see section 2.3 for formatting). Alternatively, one may provide a ‘shapes0’ structure, either being user generated (see section 2.6), or generated by FitSpec. Finally, the lowest ppm value in each dimension must be provided in a vector, ‘f’, so that ‘f’ has one entry for each dimension. This is required because the ‘par’ variable or the ‘shapes0’ variable provide information on spectrum breadth and number of points in the spectrum, but do not specify its position.

To calculate a spectrum from ‘spec_par’ and ‘par’ variables:

```
calc_spec=FullSpecCalc(spec_par,par,f);
```

To calculate a spectrum from ‘spec_par’ and ‘shapes0’ variables:

```
calc_spec=FullSpecCalc(spec_par,shapes0,f);
```

Alternatively, if the ‘spec_par’ variable can be replaced by a ‘fit’ variable (the output from ‘FitSpec’). Then, INFOS can determine ‘f’ and ‘par’ from information stored in ‘fit’ (assuming it has not been deleted by the user). The user may edit the positions, intensities, and linewidths in the ‘fit’ to calculate differences in the spectrum.

3.4.2. noise_gen

The ‘noise_gen’ function generates a spectrum of noise. This is done by taking white noise (normally distributed, uncorrelated) in the time domain and processing it according to the acquisition and processing parameters for a given spectrum. Acquisition and processing information is provided in a ‘par’ structure, which contains the fields ‘par.dX’ for each dimension X. This is described in section 2.3, although note that no information on signal decay is necessary. The result is then very similar to spectrum noise, although

discrepancies may occur because of the presence of artifacts, and any frequency filtering within the spectral width. The format of the output of ‘noise_gen’ follows that of all ‘spec’ variables (see section 2.1). Note that the frequency axes are arbitrary, so they will always be centered at a frequency of zero. The noise returned by ‘noise_gen’ will always have an rms of 1.

To generate a noise spectrum:

```
noise_spec=noise_gen(par);
```

Noise generation is useful for visual evaluation of fit quality. This is because the form of the spectrum noise heavily influences the appearance of lineshapes with low signal to noise. Furthermore, the FitSpec function may opt to fit a spectrum feature that appears to be the result of overlapping peaks with only a single peak, and when a fit is viewed with spectrum noise, it becomes apparent that the feature is likely to be a result of a single peak and spectrum noise. The following example demonstrates how to generate a calculated spectrum with noise after spectrum fitting. Note that if FitSpec runs a noise evaluation, then the output variable, ‘fit’, contains the rms of the input spectrum.

```
fit=FitSpec(spec);
calc_spec=fit.spec;
noise_spec=noise_gen(calc_spec.par);
calc_spec.S=add_spec_nD(calc_spec,noise_spec,[1 fit.noise.rms]);
```

Here, we have taken the fitted spectrum out of the results from FitSpec. Then, a noise spectrum is generated, using the acquisition and processing parameters of the calculated spectrum (these are the same as the original spectrum). Finally, the calculated spectrum and the noise spectrum are added together, using ‘add_spec_nD’ (section 3.3.4), with a weight of 1 for the calculated spectrum and a weight of the spectrum RMS (fit.noise.rms) for the noise spectrum, so that the synthetic spectrum has the same noise level as the original experimental spectrum.

3.5. Plotting

In addition to providing a variety of functions for fitting, INFOS also provides several graphical programs for viewing 2D and 3D spectra.

3.5.1. quik_2Dplot

'quik_2Dplot' plots 2D spectra using default settings, or according to user specified settings, provided in a 'plotpar' variable. Note that 'quik_2Dplot' supports overlaying spectra in logarithmic mode (if the user specifies 'hold all' for the axis), and by default will change the plotting color for new spectra.

plotpar.n_contours:

Single integer used to specify the number of contour levels to be used in plotting.

plotpar.cutoff:

Single value between 0 and 1 to specify the minimum spectrum amplitude to plot, as a fraction of the maximum of the spectrum. If set to an array with two entries (both between 0 and 1), the first entry will specify the minimum amplitude to plot and the second entry the maximum amplitude to plot.

plotpar.mode:

String specifying the plotting mode, which is either linear ('linear') or logarithmic ('log').

plotpar.colormap:

List of colors used in plotting, specified as an N x 3 array, in RGB format. Logarithmic plotting will only use 2 colors (2 x 3 array, first row for positive, second row for negative amplitudes)

plotpar.range:

Range of the spectrum plot. Should be in the order [LB1 UB1 LB2 UB2].

plotpar.dim: String specifying the plotting order. Default is '21', which places dimension 1 on the y-axis.

plotpar.scaled:

String, set to 'y' or 'n'. This is used for overlaying spectra. When set to 'y', overlayed spectra will take on the same contour levels as the previous plotted spectrum so that amplitude comparisons are straightforward. Default setting is 'n', but when set to 'y', this will override 'par.cutoff' and 'par.n_contours' settings.

For a plot with user settings in 'plotpar':

```
quik_2Dplot(spec,plotpar);
```

3.5.2. qk_3Dplot

'qk_3Dplot' allows quick plotting of slices of a 3D spectrum as a 2D plot. One specifies the plotting settings the same way as for 'quik_2Dplot' (3.5.1). The only exception is that 'par.dim' must be a string with 3 numbers ('321' is default). The first two listed dimensions will be plotted, and the slice is extracted from the third dimension. One must additionally specify the frequency in the third dimension to be extracted. A range may also be plotted, in which case an overlay of all spectra within the given range will be displayed. By default, qk_3Dplot will set 'plotpar.scaled' to 'y', so that overlaid plots will always have the same contour levels.

To plot a slice between frequency positions 110 and 111 ppm:

```
qk_3Dplot(spec,par,[110 111]);
```

3.5.3. qk_iso3Dplot

'qk_iso3Dplot' creates a 3-dimensional visualization of a 3D spectrum. Isosurfaces of the spectrum are displayed for a particular spectrum amplitude (all points in the spectrum with a given amplitude are shown as a surface). This is set with the 'level' variable. The user may specify further settings in the 'plotpar' variable.

Plotting if 'plotpar' structure is provided:

```
qk_iso3Dplot(spec,plotpar);
```

Plotting while only providing a level, here with both positive and negative amplitudes ('level=[-.4 .4]')

```
qk_iso3Dplot(spec,[-.4 .4]);
```

The fields of 'plotpar' are described below.

level: This controls the level of the isosurface. This is given as a fraction of the maximum spectrum amplitude, and so should take on some value between -1 and 1 (0 is prohibited). A positive value will restrict the plotting to positive amplitudes, specifying it as negative will restrict plotting to negative amplitudes. One may also enter two values into 'plotpar.level' (one positive, one negative) to plot both positive and negative amplitude peaks. By default, only positive amplitudes are plotted, which are 30% of the spectrum maximum ('plotpar.level=.3').

plotpar.level: Same as ‘level, but entered into the ‘plotpar’ structure to allow control of other settings.

plotpar.color: This controls the plotting color of the spectrum. Entry should be a 1 x 3 array if only using positive or only negative peaks. Otherwise, it should be a 2 x 3 array, for which the first row gives the positive color in RGB format, and the second row gives the negative color.

par.dim: This string gives the plotting order of the dimensions. Default is ‘123’.

par.range: This vector specifies the range to be plotted. Should be a 1 x 6 vector with order [LB1 UB1 LB2 UB2 LB3 UB3].

3.5.4. slice_disp

The ‘slice_disp’ program is a tool for viewing and extracting 1D slices of a 2D spectrum. One executes ‘slice_disp’ exactly the same way as ‘quik_2Dplot’ (see 3.5.1), using a ‘spec’ variable and a ‘plotpar’ variable. ‘slice_disp’ will then show the spectrum, but also includes 1D plots above and beside the spectrum. These initially show the projection across each spectrum dimension, but the user may then click on the 2D spectrum to obtain slices through both dimensions. For zooming, one may use the MATLAB zoom tool as usual. In order to store the slices, one must call ‘slice_disp’ with one or two outputs, and ‘slice_disp’ will allow the user to select a point in the spectrum to slice. When storing selected slices, the program will terminate after the user responds ‘y’ on the command line.

To store both 1D slices in a cell:

```
slices=slice_disp(spec,plotpar);
```

To store each slice separately:

```
[slice1 slice2]=slice_disp(spec,plotpar);
```

To display slices without storing the results:

```
slice_disp(spec,plotpar)
```

3.6. Additional fitting programs

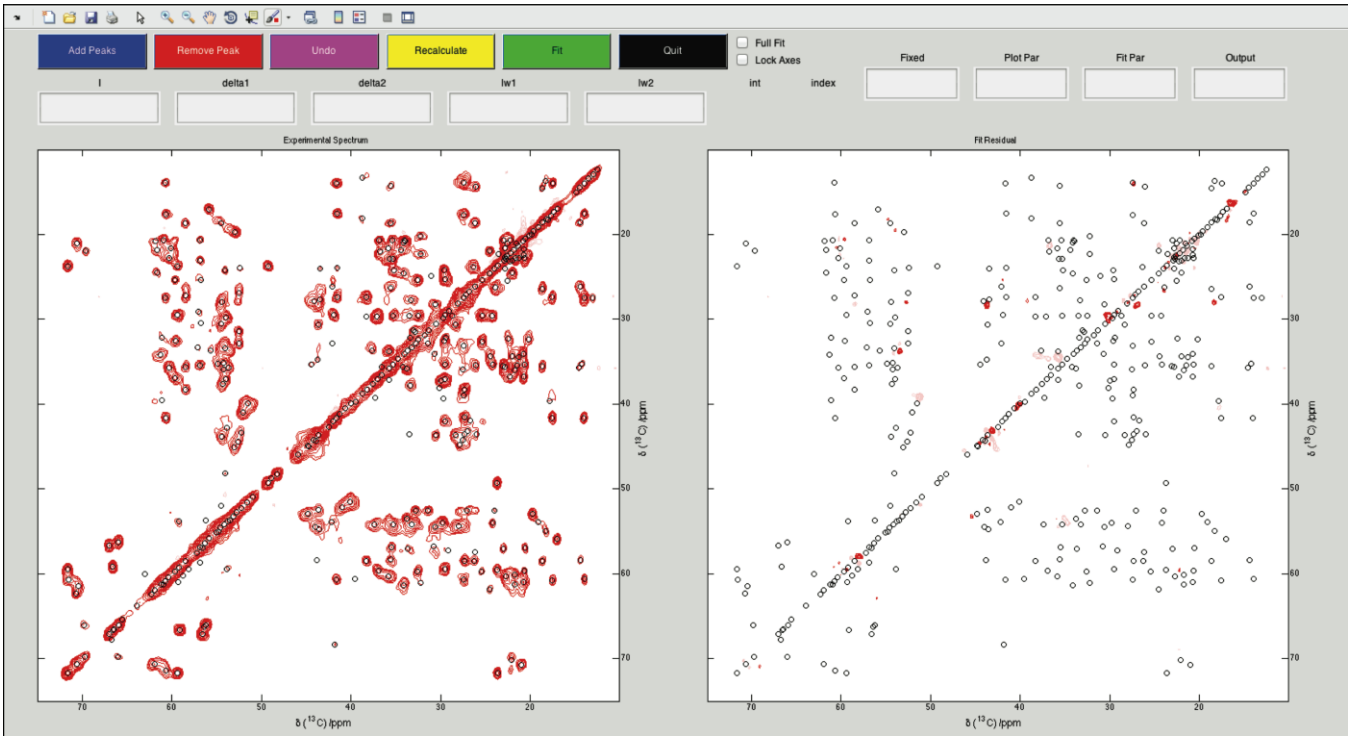
3.6.1. FitEditor2D

The FitEditor2D is an interactive program used for viewing and editing 2D spectrum fits. One may call FitEditor2D with the variable ‘fit’, which is the output of the FitSpec program. Additionally one may optionally specify plotting parameters (‘plotpar’), as are described for

the ‘quik_2Dplot’ program (section 3.5.1), and can optionally provide the ‘shapes0’ variable (section 2.6).

```
FitEditor2D(fit,plotpar)
```

Once called, FitEditor2D will open figures 20 and 21, and will have the following appearance:



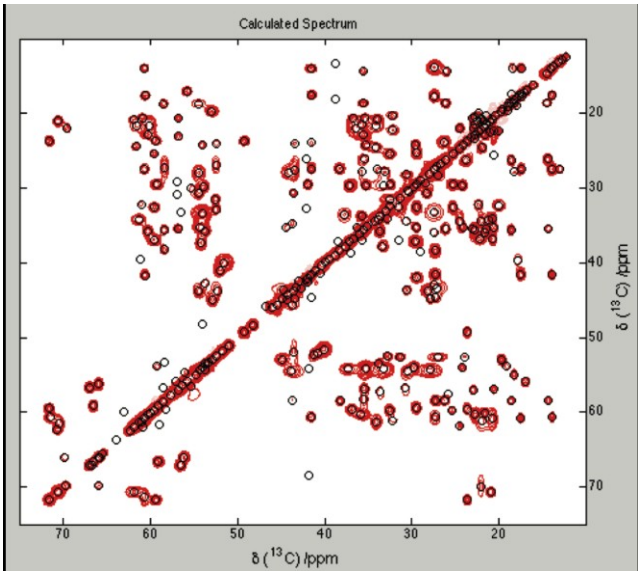


Figure 20 (the first figure), displays the experimental spectrum, and the residual of the spectrum fit, as well as all controls of the FitEditor2D program. Figure 21 shows the calculated spectrum. All plots include a contour plot and a scatter plot of all points, and the

'current' point is indicated with an X. Note that for spectra with many peaks (~1000), the MATLAB scatter plot becomes inefficient and so this may be slow.

There are several controls in the FitEditor2D program, which we describe here. Also note that all three plots are interactive, and can be clicked on to select peaks, or to place new peaks.

Buttons:

Add Peaks: This button enters the peak addition mode. In the peak addition mode, each mouse click on any of the plots will add a new peak at that position, with the linewidth and intensity estimated by the program (can be user edited after addition). Click 'Add Peaks' again to exit peak addition mode.

Remove Peaks: This button will remove the currently selected peak.

Undo: This will undo the last user action (peak addition, removal, editing). Note that if several editing steps on one peak have been made subsequently, all will be undone. Also, undo in peak addition mode only works as long as one remains in this mode.

Recalculate: This will re-plot the spectra after editing. This does not happen automatically after editing because the MATLAB plotting is somewhat slow, so the user ideally can take several editing steps before recalculating.

Fit: This will fit the spectrum, using the PartialFit routine, where the FitEditor2D function will determine what range to refit. If the user wants to refit the full spectrum, the the 'Full Fit' check box should be selected. This may be necessary for heavily overlapped regions (use if fit gets worse when the partial fit is run). Settings for fitting use either default settings or settings controlled by the user via the 'Fit Par' entry field (see below).

Quit: This quits the FitEditor2D program. Upon exit, the program will output the current fit, either as a variable named 'FE2D_fit', or in a user specified name, which is entered in the 'Output' entry field. Also, this will clear the memory used by FitEditor2D.

Entry Fields

I, delta1, delta2, lw1, lw2:

These fields give information about the currently selected peak (amplitude, position, and linewidth). The user may also enter

information into the fields to edit the currently selected peak (spectrum will not update immediately, although marker will).

Plot Par: This field allows the user to update the plotting settings during use of the fit program (see section 3.5.1 for a description of the plotting settings). One may generate and edit a 'plotpar' variable on the command line. One then should enter the variable name in this field to re-plot the spectra with new settings. It is also possible to generate the plotpar variable in the window, by constructing a structure with the 'struct' function.

Fit Par: This field allows the user to control the fitting settings of the FitSpec program. One may create a 'par' variable at the command line or using the 'struct' function in the window, using the settings described in section 2.2, and enter its name into this field.

Output: This field allows the user to create a variable that contains information on the current fit. Upon entering a name here, FitEditor2D will immediately output the current fit to a variable with the given name. Also, upon exit, the current fit will be written to the variable specified here. Therefore, if the user wants to store different fits, they should change the name here before program exit (otherwise it will overwrite the last fit).

Other Fields

int: Gives the peak integral. Only calculated after spectrum fitting, so that edited or newly added peaks will not have up-to-date integral values

index: Gives the number of the current peak index, so that the user may find/edit the peak in the 'fit' variable.

Full Fit: If this checkbox is selected, when using the 'Fit' button, the full spectrum will be refit, rather than performing only fits around recently edited peaks. This leads to better fitting but slower performance

Lock Axes: If this checkbox is selected, zooming in on one plot will cause the other two plots to also zoom in to have the same axes.

3.6.2. FitTrace

The FitTrace function allows the user to fit a series of spectra simultaneously, for which the amplitudes in those spectra can be described by some user-defined function. In this case,

the user must provide the series of spectra, a matrix describing the user-defined function, fitting parameters, and an initial fit. These are detailed below.

```
fit = FitTrace(spec,trace,par,fit0);
```

Here, ‘spec’ contains the series of spectra, ‘trace’ gives the user defined function, ‘par’ is the usual set of fitting settings, and ‘fit0’ is an initial guess of the fit.

spec: The ‘spec’ variable is constructed in one of two ways. The first option is that the series of spectra is stored in a cell, with one cell entry per spectrum. The individual spectra are then structures as described in 2.1. The second option is that a single structure is created, where the field ‘S’ has an additional dimension to contain the series of spectra. This dimension must be the last dimension. Also, it should not have a corresponding frequency axis in the spec structure (for example, a series of 2D spectra should not have a field ‘spec.f3’).

trace.x: This is a 1xN vector which contains a parameter describing the user-defined function to be fitted. For example, if one fits to an exponential decay, ‘trace.x’ would contain all possible rates of decay. Note that FitTrace will not extrapolate between values in ‘trace.x’, so that this variable must be defined finely enough to get the required precision. However, too many values in ‘trace.x’ will increase memory requirements.

trace.Fx: This is an MxN matrix. Each column of ‘trace.Fx’ corresponds to a value in ‘trace.x’, giving the form of the user-defined function for that value of ‘trace.x’. Therefore, the length of the second dimension must match the number of elements in ‘trace.x’. The first dimension must have the same number of elements as there are spectra in the series. Note that the user-defined function must be differentiable with respect to ‘trace.x’ (since it is numeric, this is necessarily the case, but sharp changes in ‘trace.Fx’ will yield poor performance)

par: The ‘par’ structure functions the same way as described in section 2.2. Note that setup should be done as if for the single spectra (so a series of 2D spectra, even if combined into a 3D structure, should be given 2D setup parameters). The ‘par.fixed’ field may contain one additional entry at the end, allowing one to fix the variable on the user-defined

function (although this option is not usually useful- since this is the variable one usually wants to extract)

fit0: The 'fit0' structure is required for the FitTrace function (unlike the FitSpec function). Setup is the same as described in section 2.5, although one only provides an initial fit for the single spectra. One may optionally include a vector 'fit0.x' which provides an initial value for the variable describing the user-defined function. This will cause the first step described below to be skipped

fit: The output variable, 'fit', is the same as described in section 2.4, except that it contains an additional field, 'fit.x', which contains the fitted values for the user-defined function.

The FitTrace function is a special implementation of the FitSpec function. If the field 'fit0.x' is not provided, then the function begins by estimating the value of 'fit0.x'. This is done by running fits of the individual spectra, with peak position and linewidth fixed, so that only the amplitude is allowed to vary. The resulting amplitudes of each peak are then fit to the user-defined function to obtain initial values of fit0.x. FitTrace then constructs a new spectrum which contains the series of spectra. The last dimension becomes the fitting dimension of the user-defined function. FitTrace sets this dimension up so that the position in this dimension does not vary, and the linewidth in this dimension is used as the user-defined variable. Because the user-defined function extends over this entire dimension, it no longer makes sense to break the dimension into sections (see 'par.grid', 2.2.1; the other dimensions are still broken up). This increases the memory requirements, so that FitTrace uses more grid sections in the other dimensions than FitSpec. This can be user adjusted to fewer grid sections if problems in fitting arise. Note that if memory requirements for FitTrace get to high, it may be beneficial to disable parallelization (see section 2.2.19).

Note that some series of spectra may include repetition of the same spectra (repetition of time points in a decay curve, for example). This is easily managed in FitTrace, simply by repeating rows in 'trace.Fx' to correspond with the repeated experiments.

3.6.3. PartialFit

The PartialFit function allows the user to select only part of a spectrum and fit that region. One must specify the region to be fitted, and optionally may select which peaks in that

region should be fit. The advantage of using this function as opposed to simply clipping the spectrum (see 'clip_spec_nD', section 3.3.1) and fitting, is that PartialFit takes into account peaks neighboring the region to be fitted either by referencing to an initial fit or by performing an approximate peak pick and linewidth estimation of surrounding peaks before fitting. Note that the PartialFit function is a critical component to the FitEditor2D and the FitError functions. PartialFit is called as follows:

```
fit = PartialFit(spec,par,fit0)
```

where the 'par' and 'fit0' variables are optional. A fourth argument, 'shapes0', can be also provided (section 2.6). The 'spec' and 'fit0' are the same as described in sections 2.1 and 2.2. 'par' has the same fields as described in section 2.5, plus two additional fields.

par.fit_range: This is given in a vector with entries [LB1 UB1 LB2 UB2 …] where LB and UB indicate the lower and upper bounds of each dimension. Therefore, this must have a size of 2 X # of dimensions.

par.index: This is an optional field, only available if an initial fit is given. This allows the user to choose which peaks should be fitted. Therefore, this is either a logical index with an entry for each peak (must be same size as 'fit0.I', 'fit0.delta1', etc.), or a list of numbers for all the peaks that should be fitted (to fit peaks 1,3,10, then fit0.index=[1 3 10])

3.6.4. FitError

The FitError function is used for estimating error of the fitting parameters. Typically, it should be run using the same fitting settings as the initial fit was acquired with, in order to obtain accurate error analysis. The function takes a spectrum fit and the calculated spectrum from that fit, adds noise to it, and refits it. This is done many times (~100s), in order to estimate the effect that experimental noise has on fitted parameters. Statistics of the refit parameters can be used to estimate statistics of the experimental parameters. Note that application of this method is limited, however. First, it assumed that noise in the time domain is uncorrelated, white noise, so that error induced by spectrum artifacts, incomplete phase cycling, etc. will not be accounted for. Second, the error values produced are the representative of experimental error only if the fit is a good representation of the 'correct' values of the peak parameters. Therefore, in regions where there are missing peaks, or if a peak is picked which is actually noise, then the error in those situations will likely not represent the real error (one can consider the 'FitError' function to be simulating experimental repetition). This is discussed in more detail in the accompanying publication.

The FitError function does not repeatedly refit the full spectrum. Rather, for each peak that is evaluated for error, FitError fits a partial spectrum around that peak, using the PartialFit function. FitError requires that if the error of a peak is determined from a partial fit, then that partial fit must also include neighboring peaks that fall within a certain range of the peak of interest (see 'par.NB_cutoff' below). Also, the region fitted around the peak and its neighbors must have a certain width (see 'par.H_cutoff'). However, since now some sections must contain multiple peaks, it is additionally possible to analyze the error of those peaks- if the section can be modified so that these additional peaks also have their neighbors and a wide enough range around the peak. This set of peaks is refererred to as a group in INFOS. To prevent a group from becoming too large (and therefore slow to fit), FitError restricts the number of peaks and the frequency range ('par.max_peaks' and 'par.max_width'). These restrictions mean that the fitting of individual groups can be performed without breaking the spectrum into further sub-spectrum, eliminating the need for multiple iterations of fitting. Because the number of parameters is limited, the fitting is relatively fast. Note that it is then possible that a peak will be fitted in a group, but it is not possible to include its neighbors, so that error analysis is not performed on that peak. In this case, that peak will have its own group, in order to obtain statistics. This will lead to redundancy in fitting some peaks, but in spite of this, breaking the spectrum into groups is significantly faster than repeating the fit of the complete spectrum many times. After fitting a spectrum, FitError can be called as follows

```
fit=FitSpec(spec,fitpar);
err=FitError(fit,par);
```

It is also possible to call without any parameters (one or both of the fitting parameter arguments can be left out.

```
fit=FitSpec(spec);
err=FitError(fit);
```

The 'fit' structure is simply the output of the FitSpec function. The 'par' structure can be constructed as described in section 2.2. However, note that the values in 'par' used during the initial fit are passed to the FitError function via the substructure 'fit.par'. Since one should use the same fitting parameters in the initial fit and in the FitError analysis, it typically is not a good idea to pass new parameters via the 'par' structure. The exception is that a few additional parameters are used by FitError for determining number of refits and for forming the groups of peaks to be fit. Note these parameters all have default values, and so do not need to be provided.

par.N: This is the number of times the refitting should be performed. Default is 100, which gives reasonable estimates of the standard deviation in most cases. However, if a full distribution of a parameters is required (for example, if distribution is not a normal distribution), then many more (~1000) iterations may be needed.

par.NB_cutoff: This field is determines how far to search around a peak for its neighbors. For a peak of interest, 'par.NB_cutoff' is multiplied by that peak's height. Then, in each dimension, FitError determines what distance from the peak center the peak falls below this amplitude. Neighboring peaks are then defined as any peak falling within this range in all dimensions. Default is 10% of the maximum peak height ('par.NB_cutoff=0.1').

par.H_cutoff: This field determines how wide the fitted area around peaks in a group needs to be. The value is defined similarly to 'par.NB_cutoff', with the width required being determined from a fraction of the peak height. Then, a group must be wide enough that all peaks in the group have at least the width around them defined by 'par.H_cutoff'.

par.max_width This is the maximum spectral width a group may have. This must have one entry per spectrum dimension. Note that when a group is formed, it is possible that the neighbors around the first peak in the group are spread out enough to already exceed the maximum width. In this case, 'par.max_width' will be overridden, although no further peaks will be added to such a group. Default is 1.5x the size specified by 'par.grid' in the initial spectrum fit.

par.max_peaks: This is the maximum number of peaks allowed in a group. As with 'par.max_width', if the first peak has more neighbors than allowed by 'par.max_peaks', then this parameter will be overridden. Default is 10.

par.index: This allows the user to determine which peaks should have error analysis performed. Field is the same that for PartialFit (section 3.6.3). It should either be a logical index with the same size as 'fit.I', or a vector listing which peaks should be analyzed for error.

The output of FitError then contains statistics calculated for the individual fitting parameters, in addition to a record of the fitting results of all 'par.N' fitting iterations. These are stored in the following fields

err.par: Contains values used for error analysis as described above ('par.N', 'par.NB_cutoff', etc.)

err.fit: Returns the initial fit (simply the input variable returned)

err.I_std: Standard deviations of the peak amplitudes

err.deltaX_std Standard deviations of the peak position in the X^{th} dimension

err.lwX_std Standard deviations of the peak linewidth in the X^{th} dimension

err.FWHMX_std Standard deviations of the peak FWHM in the X^{th} dimension

err.int_std Standard deviations of the peak integral in the X^{th} dimension

err.groups Cell with the results of fitting each of the groups (has same number of elements as the number of groups). Inside each cell of 'err.groups', the results of the 'par.N' refittings are stored. Therefore, this contains all the same fields as in the usual 'fit' structure (section 2.4), excepting that the spectra ('fit.spec', 'fit.resid') have been removed to reduce the variable size. However, each fit parameter ('fit.I', 'fit.delta1', etc.) is now an Nxpar.N matrix, with N being the number of peaks in the group, and par.N the number of repetitions. There is also an additional field, 'err.groups{k}.calc_list' which lists for each peak in the group its index in the original fit. Note that the field 'err.groups{k}.par' is as described in section 2.2, but also includes fields 'err.groups{k}.par.fit_range' which indicates the range of the spectrum that has been fitted, and 'err.groups{k}.par.index' which indicates which peaks are fitted in a group.

err.group_index This list indicates which group contains each peak. Note that some peaks can be contained in more than one group, but statistics should be analyzed from the group indicated here.

err.subgroup_index This list indicates for each peak, which element of the group corresponds to this peak

To understand indexing of groups better, a quick example on how to determine the histogram of a peak's amplitude is shown. Suppose that 'x' is an edges vector for a histogram (see histc.m), and we want the histogram of the peak with index 'k'. Then, the histogram is given by

```
h=
histc(err.groups{err.group_index(k)}.I(err.group_subindex(k),:),
```

```
x-diff(x(1:2))/2);
```

One sees that the correct group was found with the index ‘err.group_index(k)’, and the correct element of that group was found with the index ‘err.group_subindex(k)’. In the latter, case, the full row must be extracted for use in the histogram calculation, so the ‘:’ operator was used. The edges vector (‘x’) is shifted by half a unit to center the histogram.