NAME

nmrPipe − Vector Processing of Spectral Data

SYNOPSIS

nmrPipe [**−fn** *fnName*] < *inFile* > *outFile* [**−in** *inName* [**−nofs**]] [**−out** *outName* [**−ov**] [**−nofs**] [**−inPlace**]] [**−tty**] [**−fdata** *loc1 parm1...*] [**−xi | −dt | −di**] [**−ai | −ad | −ac**] [**−select** *y1 yn [z1 zn...]* [**−inside | −outside**]] [**−verb** [*verboseLevel*]] [**−inc** *verboseIncrement*] [**−dbg** *debugReportLevel*] [**−X11**] [**−noSig**] [**−help** [**−allHelp**]] [**−client** *serverListName*] [**−server** *portFileName*] [**−sID** *serverID*] [**−inSock** *inSocketPort*] [**−outSock** *outSocketPort*] [**−path** *pathName*] [**−smt** *serverMaxTries*] [**−sto** *serveruSecTimeOut*] [**−buff** *ioBuffCount*] [**−ibuff** *inputBuffCount*] [**−obuff** *outputBuffCount*] [**−mt** *maxTries*] [**−to** *uSecTimeOut*] [**−noRd**] [**−noWr**] [**−inSwap**] [**−outSwap**]

AUTHORS

The **NMRPipe** System of spectral analysis tools was designed and implemented by Frank Delaglio at the National Institutes of Health, Laboratory of Chemical Physics, NIDDK, Bethesda MD, in the Section on Biophysical NMR directed by Ad Bax. The software includes contributions from Guang Zhu, Geerten Vuister, Lewis Kay, John Pfeifer, Dan Garrett, Dennis Benjamin, Toshio Yamazaki, Dean Carlson, E. Neil Lewis, and Stephan Grzesiek. The software can be cited by referring to the following publication:

NMRPipe: a multidimensional spectral processing system based on UNIX Pipes

F. Delaglio, S. Grzesiek, G. Vuister, G. Zhu, J. Pfeifer, and A. Bax

J. Biomol. NMR, **6** (1995) 277-293.

Comments on the software are welcome, and questions will be answered when possible. Electronic correspondence can be addressed to delaglio@speck.niddk.nih.gov.

DESCRIPTION

The **nmrPipe** program is the central part of a system of tools for multi−dimensional spectral processing using UNIX pipes. It applies a given spectral processing function, for instance a one−dimensional Fourier Transform, to a series of spectral data vectors; successive applications of the **nmrPipe** program are used to build up a processing scheme for a single dimension of a multidimensional data set. When used in conjunction with data input and output programs such as **xyz2pipe** and **pipe2xyz**, **nmrPipe** can be used to construct complete multi−dimensional processing schemes.

In most cases, **nmrPipe** processing schemes are constructed as UNIX shell−scripts, which are text files used to execute **nmrPipe** and related programs. In practice, these shell−scripts just define the input data source, the list of processing functions to apply, and the output data destination. The shell scripts can be constructed with a UNIX text−editor such as **vi**, or inside a graphical interface such as **nmrDraw**. Sample shell−scripts for various purposes are given in the **EXAMPLES** section below, as well as in the manual pages of related commands.

The following steps are involved in a typical processing session:

Convert the spectrometer format data to **nmrPipe** format; this is usually performed by editing a shell−script which runs a conversion program such as **bruk2pipe** or **var2pipe**. The conversion script defines the input and output data names, and the acquisition parameters.

Inspect the raw data, and choose parameters such as phasing; this step will commonly be performed in the interactive graphical interface **nmrDraw**.

Create and execute a processing scheme; this step involves editing a shell−script which runs several copies of **nmrPipe**, each defined to perform one aspect of the overall processing.

Inspect the processed result, to decide if any additional processing is required, for instance re−phasing or baseline correction. As before, this step will commonly be performed in the interactive graphical interface **nmrDraw**.

BACKGROUND: UNIX FILTERS AND PIPES

Many of the NMRPipe System's advantages derive from its use of specific features in the UNIX operating system, so some details of UNIX are reviewed here.

The **nmrPipe** program operates like a UNIX filter; this means that it reads an input stream of data, processes the data in some way, and writes the result as an output stream. In UNIX terminology, the input stream of data is called the standard input, and the output stream of data is called the standard output.

By default, standard input is data entered at the keyboard, but commands can be constructed so that the input is taken from a file instead; this is called *input redirection*.

On the other hand, data written to the standard output is displayed on the computer screen, but commands can also be constructed so that the output is written to a file instead; this is called *output redirection*. Input redirection is performed by the command character "<", while output redirection is performed by the command character ">".

Programs that read standard input and write results to standard output can also be connected together by UNIX pipes. Pipes allow separate UNIX commands to be connected together in a series, where the output of one program is automatically used as the input of the next program. A series of programs connected this way is sometimes called a pipeline. A pipe is specified in a UNIX command line by the "|" vertical bar character.

USING NMRPIPE

The **nmrPipe** program operates on data that has been converted from the spectrometer format by utilities such as **var2pipe** or **bruk2pipe**. When using the **nmrPipe** program to process converted spectral data, the following information is specified:

1. Source of the input data to process.
2. Processing function to apply.
3. Parameters required for the processing function.
4. Destination of the processed output data.

If this information is not specified explicitly in the command line, default values will be used, as outlined below. For simplicity, some of the following examples are incomplete commands (e.g. missing input or output); they are indicated as "command fragments" in the text.

SOURCE OF INPUT DATA

By default, the **nmrPipe** program reads its standard input for data to process. This means that UNIX facilities can also be used to re−direct the program's input so that it comes either from a file or from another program which writes to standard output. In addition, the argument **−in** *inName* can also be used to specify that input should be read from a given file. The following command fragments all show **nmrPipe** using data from the 2D spectral data file *test.fid* as input (note that the UNIX command **cat**(1v) displays the contents of a file):

```
nmrPipe < test.fid
cat test.fid | nmrPipe
nmrPipe −in test.fid
```

SPECIFYING THE PROCESSING FUNCTION

By default, the **nmrPipe** program applies the **NULL** processing function, which leaves the data unchanged. A different processing function can be selected by using the **−fn** *fnName* argument. The names of all available **nmrPipe** functions can be listed by using the **−help** flag. Most **nmrPipe** processing functions perform the same operation on all 1D vectors from the input stream.

For example, the function named **FT** is used to apply a Complex Fourier Transform to all the 1D vectors from the input data. So, the command fragment for transforming all the 1D vectors in the acquisition dimension of the 2D spectral data file *test.fid* might look like this:

```
nmrPipe –in test.fid –fn FT
```

Note that each instance of **nmrPipe** can have only one function as specified by the **–fn** *fnName* option.

SPECIFYING PROCESSING PARAMETERS

Depending on the processing function selected, **nmrPipe** accepts other arguments which define relevant processing parameters. The arguments associated with a given function can be displayed by using the **–fn** *fnName* argument along with the **–help** flag. This will generate a brief description of the specified function, its arguments, and their default values. For instance, the following example shows the command used to display the arguments for the adjustable sine window function, **SP**:

```
nmrPipe –fn SP –help
```

This command will produce the following help text:

```
SP: Adjustable Sine Window. [SINE]
 –off   offset [0.0]  Sine Start*PI.
 –end   end    [1.0]  Sine End*PI.
 –pow   exp    [1.0]  Sine Exponent.
 –size  aSize  [APOD] Apodize Length.
 –start aStart [1]    Apodize Start.
 –c     fScale [1]    Point 1 Scale.
 –one                 Outside = 1.
 –inv                 Invert Window.
```

Each line shows a command flag followed by the name of its argument value if any. The default parameters for each argument are shown in square braces "[]"; a terse description of the argument is given on the right. Floating–point and integer parameters can be distinguished by their default values. In addition to this help text, many processing functions also have a more extensive *manual page*, which can be viewed by using the **man** command, for example:

```
man SP
```

In the help listing above, the default parameters call for a sinusoid with a starting point (offset) of 0.0*Pi radians, and an endpoint of 1.0*Pi radians. This means that when the function **SP** is used with no additional arguments, it will generate an exact sine–bell window, as in this command fragment:

```
nmrPipe –in test.fid –fn SP
```

The function can be altered by including other arguments; for example, this command fragment:

```
nmrPipe –in test.fid –fn SP –off 0.5 –pow 2 –c 0.5
```

will generate a cosine–squared window which also multiplies the first point in every 1D vector by 0.5.

In the help listing for **SP** above, the default length of the window function is listed as "APOD", which means that the default value will be taken from the time–domain size recorded in the data during conversion. Default values for many functions may also be expressed in terms of "SIZE" or "XSIZE" (the 1D size of the current dimension), or "YSIZE" (the 1D size of the next dimension). Note that these upper–case keywords should not be used in a command line; they are just reminders to the user that the default values depend on the parameters stored in the given data file.

DESTINATION OF OUTPUT DATA

By default, the **nmrPipe** program writes its processed data to standard output. As before, this means that UNIX facilities can be used to re–direct the program's output so that it goes either to a file or to another program which reads from standard input. In addition, the argument **–out** *outName* can also be used to specify that output should be written to a given file. The following command fragments all show **nmrPipe** writing data to the output file *test.dat* (note that the UNIX command **dd** is a data copying program):

```
nmrPipe –in test.fid > test.dat
nmrPipe –in test.fid | dd of=test.dat
nmrPipe –in test.fid –out test.dat
```

CREATING PROCESSING SCHEMES

A complete instance of the **nmrPipe** program includes a data source, processing function specification, and a data destination, as outlined above. So, for example, a complete command to Fourier Transform the acquisition vectors of *test.fid*, with the result saved in *test.ft*, might look like this:

```
nmrPipe –in test.fid –fn FT –out test.ft
```

As mentioned before, the **nmrPipe** program will apply only a single processing function; however, in most spectral processing schemes, several functions must be applied. To achieve this, multiple instances of the **nmrPipe** program can be joined together by UNIX pipes.

For example, the following scheme applies both a window function and a Fourier Transform to the acquisition vectors in file *test.fid*:

```
nmrPipe –in test.fid –fn SP      \
| nmrPipe –fn FT –out test.ft
```

Notice that the above command includes two instances of the **nmrPipe** program, connected by a pipe. The character "\" is used at the end of the first line to indicate that the command is continued on the next line.

Note well that there should be *no trailing blank spaces* after the "\" character, or the processing scheme will not work.

Following the example above, additional processing functions can be included simply by adding instances of **nmrPipe**. Here, the zero filling function **ZF** is added to the scheme above:

```
nmrPipe   –in  test.fid –fn SP    \
| nmrPipe                –fn ZF    \
| nmrPipe –out test.ft  –fn FT
```

The scheme above performs a window function, zero fill, and Fourier Transform on all vectors in the acquisition dimension of file *test.fid*, and saves the processed result in *test.ft*. Considered in more detail, the scheme above consists of three instances of **nmrPipe**, connected by pipes, and running at the same time. The first instance of **nmrPipe** reads some data from *test.fid*, applies the window function **SP**, and writes the result vectors to the pipeline. At the same time, the second instance of **nmrPipe** reads these result vectors from the pipeline, applies the zero fill function **ZF**, and writes the zero−filled vectors to the pipeline. The third instance of **nmrPipe** reads zero−filled vectors from the pipeline, applies the Fourier Transform function **FT**, and writes the result vectors to the file *test.ft*. It is worthy to emphasize that a stream of vectors is moving continuously through the pipeline, so that all three functions are active and running asynchronously during processing.

PROCESSING SCHEMES WITH TRANSPOSE FUNCTIONS

The examples so far only process vectors from the acquisition dimension of the input data. In order to process the next dimension, the transposition function **TP** can be included in the pipeline; it exchanges the row and column vectors in the 2D input planes, accommodating real, complex and hypercomplex data. So, for example, the following scheme is a simple 2D Fourier Transform of the 2D hypercomplex (e.g. States Mode) file *test.fid*:

```
nmrPipe   –in  test.fid –fn FT –di \
| nmrPipe               –fn TP     \
| nmrPipe –out test.ft  –fn FT –di
```

In this 2D example, the first instance of **nmrPipe** reads the input data from *test.fid*, applies a Fourier Transform to the input vectors, deletes the imaginary data for this dimension as requested by the **–di** flag, and writes processed results to the pipeline. The second instance of **nmrPipe** collects all the vectors in the 2D plane, and only then begins to write vectors in transposed order to the pipeline. The third instance of **nmrPipe** reads the transposed vectors from the pipeline, applies a Fourier Transform, deletes the imaginaries, and writes the result to file *test.ft*. This means that the second Fourier Transform in the example above is idle while the first Fourier Transform is running, since it will not receive data vectors until the transpose step is finished.

Processing schemes can include more than one transpose step. In the following example, the automated baseline correction function **POLY –auto** is applied to the acquisition dimension only after the 2D data is Fourier Transformed in both dimensions:

```
nmrPipe   –in  test.fid \
| nmrPipe –fn SP –off 0.5       \
| nmrPipe –fn FT –di            \
| nmrPipe –fn TP                \
| nmrPipe –fn SP –off 0.5       \
| nmrPipe –fn FT                \
| nmrPipe –fn PS –p0 90.0 –di   \
| nmrPipe –fn TP                \
| nmrPipe –fn POLY –auto        \
| nmrPipe –out test.ft
```

PROCESSING SCHEMES AS SHELL–SCRIPTS

Multi–line processing schemes like the ones above will usually be typed into a text file first, and then executed as a UNIX shell script. Note that some versions of UNIX may require that the shell–script is identified as a C–shell script in order to work properly. This means that the very first line of every **nmrPipe** shell script should be:

```
#!/bin/csh
```

As a reminder of this, the schemes in the **EXAMPLES** section below begin with "`#!/bin/csh`".

Schemes in shell–script files easy to modify and re–use as they are developed. For convenience, the **–fn** *fnName* option is often omitted at the head or tail of a processing pipeline, so that the processing functions listed in the interior of the pipeline can be changed or adjusted without effecting the input or output specification. For example, this processing scheme:

```
nmrPipe   –in  test.fid \
| nmrPipe –fn SP –off 0.5    \
| nmrPipe –fn FT –di         \
| nmrPipe –fn TP             \
| nmrPipe –fn SP –off 0.5    \
| nmrPipe –fn FT             \
| nmrPipe –out test.ft
```

can easily have a final phase–correction (function **PS**) added just by inserting an additional line:

```
nmrPipe   –in  test.fid \
| nmrPipe –fn SP –off 0.5    \
| nmrPipe –fn FT –di         \
```

```
| nmrPipe –fn TP             \
| nmrPipe –fn SP –off 0.5    \
| nmrPipe –fn FT             \
| nmrPipe –fn PS –p0 90.0    \
| nmrPipe –out test.ft
```

USING VERBOSE MODE

The example processing scheme above will work silently unless an error occurs; however, in most cases, it is useful to design processing schemes which generate status messages during processing. This can by done by including the **–verb** flag with selected instances of **nmrPipe**. The **–verb** flag will cause **nmrPipe** to print status messages showing how many vectors have been processed so far. Usually, the **–verb** flag is used for one and only one instance in each dimension of a processing scheme:

```
nmrPipe   –in  test.fid \
| nmrPipe –fn SP –off 0.5          \
| nmrPipe –fn FT –di       –verb   \
| nmrPipe –fn TP                   \
| nmrPipe –fn SP –off 0.5          \
| nmrPipe –fn FT                   \
| nmrPipe –fn PS –p0 90.0 –verb    \
| nmrPipe –fn TP                   \
| nmrPipe –fn POLY –auto  –verb    \
| nmrPipe –out test.ft
```

If the **–verb** flag is used for more than one instance in a given dimension, the status messages from the different processing functions will overlap and replace each other, since the functions in a given dimension are operating at the same time.

PROCESSING FUNCTIONS

The following is a brief list of processing functions available in the current version of nmrPipe:

NULL **Null Function:** the default **nmrPipe** function; this leaves the data unchanged.

FT **Fourier Transform:** applies a forward or inverse Complex Fourier Transform to complex, real, or sequential (Bruker) vectors of any size.

RFT **Real Transform:** applies a Real Fourier Transform to real data. Note that this is seldom used; Complex Transforms of real data are more usual.

HT **Hilbert Transform:** adds or replaces imaginary data. Mirror Image Hilbert transforms are also available to accommodate data with half dwell–time delays.

MC **Modulus Calculation:** replaces complex data with its modulus.

TP **2D Transpose XY–>YX:** exchanges the rows (X–Axis) and columns (Y–Axis) for real, complex, and hypercomplex 2D input planes. The 2D transpose function can also be invoked as **TP**.

ZTP **3D Transpose XYZ–>ZYX:** exchanges rows (X–Axis) with Z–Axis vectors of 3D or 4D input streams. It provides an in–memory alternative for 3D processing; its use is described in the xyz2pipe manual pages.

ZF **Zero Fill:** pads the data with zeros; the zero pad count can be selected by the user, or a suitable value can be selected automatically. In inverse mode, the ZF function reverses effects of zero fill by extracting the original time–domain data.

REV **Reverse Data:** reverses the order of the points in each vector.

PS **Phase Correct:** applies zero and first order phase correction, with the option of first reconstructing imaginary data via hilbert transform. In inverse mode, the PS function negates the phase parameters before applying a phase correction. The phase correction function also includes options for convenient frequency shifting via time−domain correction. The shift parameter can be specified in terms of points or spectral units such as PPM, and the chemical shift calibration information is updated accordingly.

EM **Exponential Multiply:** multiplies the data by an exponential window function, with the option of multiplying the first point of each vector by a given constant. In inverse mode, the EM function divides the data by the given window function and first point scaling factor.

GM **Lorentz−to−Gauss:** multiplies the data by a mixed exponential−Gauss window function, with the option of multiplying the first point of each vector by a given constant. In inverse mode, the GM function divides the data by the given window function and first point scaling factor.

TM **Trapezoid Multiply:** multiplies the data by an trapezoidal window function, with the option of multiplying the first point of each vector by a given constant. In inverse mode, the TM function divides the data by the given window function and first point scaling factor.

SP **Adjustable Sine Bell:** multiplies the data by an adjustable sine window, with the option of multiplying the first point of each vector by a given constant. In inverse mode, the SP function divides the data by the given window function and first point scaling factor.

ZD **Zero Diagonal:** sets a range of points on or near the diagonal to zero; it is used in diagonal suppression schemes.

MIR **Mirror Image:** performs various mirror image operations on the input data vectors.

EXT **Extract Region:** extracts a given region of the spectral data; often used to discard empty regions of a spectrum during processing. The extract region can be specified in terms of general regions (left middle or right), specific point locations or chemical shifts. It accurately exploits and updates chemical shift calibration information.

CBF **Constant Baseline:** applies a zero−order baseline correction calculated by averaging a portion of each slice. This is commonly used to remove constant offsets from time−domain data. Baseline parameters can be specified in terms of points or spectral units such as PPM.

FSH **Frequency Shift:** applies a frequency shift via inverse Fourier processing and phase correction. The shift can be a non−integer number of points. The shift parameter can be specified in terms of points or spectral units such as PPM, and the chemical shift calibration information is updated accordingly.

MAC **MACRO Processing:** applies a user−written processing function to the data vectors. The processing function is written in the subset of the C programming language supported by the **nmrPipe** utility **M**, a simple C language interpreter. This macro processing facility provides direct access to the data vectors and individual data points without the need to create, compile, and link a complete program. It also provides a large library of vector processing functions, including all the spectral processing functions available to **nmrPipe**.

LP **Linear Prediction:** applies Complex Linear Prediction to predict or replace data at the head, tail, or interior of the data vectors. Forward, backward, and mixed Forward−Backward modes are supported, along with mirror image modes.

DX **Derivative:** calculates the numerical derivative of the data vectors by central differences.

INTEG **Integral:** calculates the numerical integral of the data vectors by simple summation.

SOL **Solvent Filter:** applies a time−domain convolution function for solvent subtraction.

BASE **Linear Corrections:** applies a series of linear corrections to user−selected regions of the input vectors. Baseline parameters can be specified in terms of points or spectral units such as PPM.

POLY **Polynomial Subtract:** subtracts a polynomial from time− or frequency−domain data. In the time−domain, it can be used for solvent subtraction. In the frequency−domain, it can be used for manual or automated baseline correction; the automated baseline correction is very effective. Baseline parameters can be specified in terms of points or spectral units such as PPM.

SMO **N−Point Smooth:** applies a convolution filter for smoothing the data. Smoothing width can be specified in terms of points or spectral units such as PPM.

CS **Circular Shift:** applies a circular shift by an integer number of points, with the option of inverting the sign of shifted points. The shift parameter can be specified in terms of points or spectral units such as PPM, and the chemical shift calibration information is updated accordingly.

RS **Right Shift:** applies a right shift by an integer number of points, replacing shifted points with zeros. The shift parameter can be specified in terms of points or spectral units such as PPM, and the chemical shift calibration information is updated accordingly.

LS **Left Shift:** applies a left shift by an integer number of points, replacing shifted points with zeros. The shift parameter can be specified in terms of points or spectral units such as PPM, and the chemical shift calibration information is updated accordingly.

SHUF **Shuffle Data:** performs various data reordering operations, including separation or interleaving of real and imaginary data, swapping of real and imaginary data, and four−byte swapping.

SIGN **Sign Manipulation:** performs various sign manipulations, including negation of all or alternating points, negation of imaginary data, or negation of the left or right half of each vector.

SET **Set Data to Constant:** sets a range of points in each data vector equal to a specified constant. Range parameters can be specified in terms of points or spectral units such as PPM.

ADD **Add a Constant:** adds a specified constant to a range of data points in each vector. Range parameters can be specified in terms of points or spectral units such as PPM.

MULT **Multiply Data:** multiplys a range of data points in each vector by a specified constant. Range parameters can be specified in terms of points or spectral units such as PPM.

QART **Quad−Artifact Reduction:** reduces quad−artifacts by a complex scaling of time−domain data. This implementation will automatically inverse−transform frequency−domain data to apply the scaling. When used via the graphical interface **nmrDraw**, it can also automatically estimate scaling parameters.

COAD **Co−Addition of Data:** reduces the data by co−adding points in rows or columns, according to specified mixing coefficients.

OPTIONS

The **nmrPipe** program has two kinds of command−line options; general options, which pertain to any instance of nmrPipe, and function−specific options, which pertain only to selected processing function as specified by the **−fn** *fnName* argument. Only the general options are listed below; use **nmrPipe** with **−help** and related flags for details on function−specific options. Note that many options are not for general use; some are used for testing and development purposes, while others are used by programs which invoke **nmrPipe** to perform processing.

RULES FOR COMMAND−LINE OPTIONS

Command−line arguments play an important part in the use of **nmrPipe** and related programs. So, it is helpful to list the rules used to interpret a given command line, along with some illustrations. These rules pertain specifically to **nmrPipe** and the other programs in the NMRPipe System:

Command−line arguments and flags are case−sensitive.

Command−line options for can be given in any order.

Unknown command−line arguments will be ignored.

If an option is used more than once on the command−line, the first (leftmost) option will be used, and the remaining ones will be ignored,

According to the rules, the following commands perform the equivalent action, since options can be given in any order:

```
1. nmrPipe –out test.ft –fn FT –in test.fid
2. nmrPipe –in test.fid –out test.ft –fn FT
3. nmrPipe –fn FT –in test.fid –out test.ft
```

Since **nmrPipe** and related programs will ignore repeated or unknown command–line arguments without generating an error message, commands like the following are valid, although they may not produce the intended results:

```
1. nmrPipe –in test.fid –out test.ft –fn FT –c 0.5
2. nmrPipe –in test.fid –out test.ft –FN FT
3. nmrPipe –in test.fid –out test.ft –in FT
4. nmrPipe –in test.fid –out test.ft –FT
```

In command 1, the option "–c 0.5" is used, although the FT function does not use a "–c" flag. The "–c 0.5" will be ignored.

In command 2, the option "–FN" is used, rather than the correct lower–case option "–fn"; the function specification will be ignored, and the NULL function (no change to data) will be applied instead.

In command 3, the function is specified as "–in FT", rather than as "-fn FT" (a typographical error); the function name will be ignored, and the NULL function will be applied instead.

In command 4, the "-fn" flag was omitted by accident; the "-FT" will be ignored, and the NULL function will be applied instead.

FILE AND PROCESSING FUNCTION OPTIONS

–fn *fnName*

Specifies the name of the processing function which **nmrPipe** should apply; by default, this is the NULL function, which leaves the data unchanged. The name of the processing function is not case–sensitive. More information about individual processing functions can be obtained with the **–help** and **–allHelp** options.

–in *inName*

Specifies the name of the input file which **nmrPipe** will read and process, as an alternative to reading data from standard input. Generally, this option will be used at the head (first part) of a processing pipeline. The given input file inName must already exist, and the input data must be in a valid **nmrPipe** data format.

–out *outName*

Specifies the name of the output file where **nmrPipe** will write results, as an alternative to writing data to the standard output. Generally, this option will be used at the tail (last part) of a processing pipeline. If the given output file *outName* already exists, nmrPipe will abort without processing unless either the **–ov** or **–inPlace** options have also been specified.

–ov When used with the **–out** *outName* option, this flag will permit over–write of an existing data file. When the **–ov** flag is not used with the **–out** *outName* option, **nmrPipe** will abort without processing if the given output file already exists.

–inPlace

When used with the **–out** *outName* option, this flag allows in–place processing, i.e. a processing pipeline where the input data file is the same as the output data file. Note that if the **–inPlace** flag is not used in these situations, nmrPipe may create empty or incorrect results without issuing any error or warning messages. The **–inPlace** flag implies the **–ov** flag.

DATA TYPE AND FORMAT OPTIONS

–xi –dt –di

Delete Imaginary Data After Processing

These flags are used to remove imaginary data from the **nmrPipe** output. Note that in most processing schemes, imaginary data for a given dimension is removed during processing, once Fourier Transform and phase correction have been performed. For historical reasons, there are two options to delete imaginary data, depending on whether 2D output planes are complex in one or both dimensions; a third option is provided which automatically chooses the correct case for a given data stream:

−**xi Delete Imaginaries; Complex Data Only**. This discards the existing imaginary data, so that the 2D output planes are all−real.

−**dt Delete Imaginaries; Hypercomplex Data Only.** This deletes the imaginary data from the current dimension (the X−dimension), so that the 2D output planes are complex. This has the effect of reducing the size of the next dimension (the Y−dimension) by a factor of two.

−**di Delete Current Imaginaries;** this is equivalent to **−xi** in the case of Complex Data, and **−dt** in the case of Hypercomplex Data. Recall that when the imaginaries of hypercomplex data are deleted, the Y−Axis size is reduced by a factor of two.

Note that the **−xi** and **−dt** flags are available for backward−compatibility with earlier versions of nmrPipe; the **−di** flag should be used instead. Only one of these flags may be used for a given invocation of nmrPipe. Some **nmrPipe** processing functions, such as the 2D Transpose function **TP**, cannot be used with these flags; incompatible functions will print an error message.

−ai −ad −ac

Add Space for Imaginary Data After Processing

These flags are used to add space for imaginary data in the output. Note however that they do not actually generate correct imaginary data; they only change the recorded data type and pad the data.

Some **nmrPipe** processing functions, for instance the Hilbert Transform function **HT**, generate complex data from real data. These functions will automatically change the data type and pad the data without the need for any special flags. However, these functions will not generate hypercomplex from complex data automatically; instead, they will just replace the existing imaginary data with new imaginary data.

Therefore, these flags are used primarily in situations where hypercomplex data is being reconstructed, for instance in two−dimensional inverse−Fourier processing schemes.

−**ai Create Imaginaries; Real Data Only**. This adds space for imaginary data, resulting in complex 2D planes; the imaginary part of the result will be zeros.

−**ad Create Hypercomplex Imaginaries; Complex Data Only**. This adds space for imaginary data in the next dimension (the Y−dimension); it will increase the size of the next dimension by a factor of two by writing each output slice twice.

−**ac Create Imaginaries; Real or Complex Data**. this is equivalent to **−ai** in the case of Real Data, and **−ad** in the case of Complex Data.

−noRd This flag will keep **nmrPipe** from reading data during processing; it is intended for use with processing macros implemented with the **MAC** function, since the macros can optionally perform their own data input. **−noWr** This flag will keep **nmrPipe** from writing data during processing; it is intended for use with processing macros implemented with the **MAC** function, since the macros can optionally perform their own data output.

−inSwap

This flag will cause **nmrPipe** to perform four−byte swap operations (byte order is reverse for each four bytes of data) on the input data header before it is interpreted. It is intended to accommodate data which originate on computers with byte−swapped architectures. Note that future versions of **nmrPipe** may perform this operation automatically.

−outSwap

This flag will cause **nmrPipe** to perform four−byte swap operations (byte order is reverse for each four bytes of data) on the output data header before it is written to the output destination. It is intended to convert data for use on computers with byte−swapped architectures. Note that future versions of **nmrPipe** may accommodate byte−swapped data automatically.

−select *y1 yn . . .*

This option can be used to select a region of vectors to processed or to be passed through without processing. If used with the **−outside** option, only vectors outside the selected region will be processed, and vectors inside the selected region will be passed through. If used with the **−inside** option (default), only vectors inside the given region will be processed. The region limits are specified as a Y-Axis lower and upper bound, followed by optional Z-Axis and A-Axis lower and upper bounds. The region limits can be specified in terms of points (no labels) or with labels "hz", "ppm", "%", etc. Note that the **−select** option is limited to simple functions which do not change the data sizes or types; furthermore it may not work with macro functions which perform their own I/O, or in parallel schemes.

−fdata *loc1 parm1 . . .*

The **nmrPipe** data format includes a header array of 512 four−byte floating point values at the beginning of the data stream. The **−fdata** option allows adjustment of individual file header values in the output, specified as header location/header value pairs; the locations are specified as origin=1, and the values are floating point. The locations, their meanings, and values are outlined in **sethdr**.

For example, **−fdata** *222 0.0* would set the location 222 (the 2D Transpose Flag) to zero (Not Transposed). The **−fdata** option is intended primarily for work−around and testing purposes.

REPORT GENERATING OPTIONS

−verb [*verboseLevel*]

This flag will cause the **nmrPipe** processing function to print status messages during processing; the messages are written to standard error. By default, **nmrPipe** processing functions will work silently unless some error occurs. Most **nmrPipe** processing schemes have the **−verb** option set for one and only one processing function in a given dimension of the data. If more than one function has the **−verb** option set, messages will overlap on the screen.

The *verboseLevel* parameter has the following effect:

0 = Work Silently Unless an Error Occurs
1 = Print Function Name and Data Size
2 = Print Messages as 1D Vectors are Processed

If the **−verb** flag is included without a verboseLevel argument, level 2 is assumed. The frequency of messages produced at verbose level 2 can be adjusted by the **−inc** option. Note that in some cases, printing status messages can take a substantial portion of the overall processing time, so it is sometimes best to minimize the verbose level.

−inc *verboseIncrement*

This option adjusts how often status messages are displayed during processing; this only has an effect if the verbose level is set to 2 by the **−verb** option. By default, status messages are printed for every 16 vectors processed.

−dbg *debugReportLevel*

This option enables printing of debug messages; it is included for testing and development purposes.

–X11 This option will cause **nmrPipe** to issue certain error or informational messages via an X11 graphical interface, rather than typing the message in the terminal window. It is used by graphics application programs such as **nmrDraw** which invoke **nmrPipe** to do processing.

–help When used alone, the **–help** option will generate a list of **nmrPipe** command–line options, followed by a list of available processing functions. After the help text is printed, **nmrPipe** will exit.

When used with the **–fn** *fnName* option, the **–help** flag will generate argument descriptions for the specified processing function.

When used with the **–allHelp** option, the **–help** flag will generate argument descriptions for all available processing functions.

Note that all help text is written to standard error; in order to re–direct standard error text to a file, use the ">&" idiom:

nmrPipe –help >& help.out

–allHelp

When used with the **–help** option, the **–allHelp** flag will generate argument descriptions for all available processing functions.

PARALLEL PROCESSING AND INTER–PROGRAM INTERFACE OPTIONS

–client *serverListName*

This option invokes an experimental parallel processing mode which distributes the given processing task to a list of computers on the network. The current implementation provides only static load balancing, which means that it will not compensate for changing speeds of each computer during processing.

The argument serverListName is the name of a text file which lists the computers to use along with arbitrary numbers which indicate their estimated relative speeds. In this example, the three computers named **spin**, **spam**, and **spite** are listed; according to the listing, computer **spite** is 4 times faster than computer **spin**, while computer **spam** is 2 times faster than computer **spin**:

```
spite  40
spam   20
spin   10
```

When using this parallel processing option, the following notes and restrictions apply:

The computers should be named exactly as listed by the UNIX command **hostname**.

The user should should be able to reach each computer via the UNIX remote–shell command **rsh**(1C).

The **nmrPipe** program should be on the user’s search path for all computers listed.

Due to network access times and the simple load balancing method, only the most time–consuming functions (such as Linear Prediction) will benefit from this kind of parallel processing.

The computers named should all have access to some common directory, as specified by the **–path** option; this directory will be used to establish inter–process communication.

−path *pathName*
This option specifies the directory path for the text file which an nmrPipe server process will use to establish communications with its client; by default, it is the current working directory.

−server *portFileName*
This option specifies the text file which an **nmrPipe** server process should use to establish communications with its client; it is intended for use by programs which invoke **nmrPipe** for client−server based parallel processing.

−sID *serverID*
This flag is used to associate a server ID number with a given nmrPipe server process; it is intended for use by programs which invoke nmrPipe for client−server based parallel processing.

−inSock *inSocketPort*
This flag will cause **nmrPipe** to read its input data from the existing socket designated by *inSocketPort*, as opposed to reading data from standard input or from a file. This option is generally used by programs which silently invoke **nmrPipe** to perform processing, such as the graphics interface **nmrDraw**.

−outSock *outSocketPort*
This flag will cause **nmrPipe** to write its output data to the existing socket designated by *outSocketPort*, as opposed to writing data to standard output or to a file. This option is generally used by programs which silently invoke **nmrPipe** to perform processing, such as the graphics interface **nmrDraw**.

−smt *serverMaxTries*
This option sets the count for the maximum number of times an nmrPipe server or client can re−try to read or write data over a socket connection. The default value of −1 allows an unlimited number of re−tries. This option is intended for testing and debugging purposes.

−sto *serveruSecTimeOut*
This option sets the delay time in microseconds between re−tries of client−server requests to read or write data over a socket connection. The default value is 512 uSec. This option is intended for testing and debugging purposes.

MISCELLANEOUS OPTIONS

−ibuff *inputBuffCount*
This option selects the number of 1D vectors which will be buffered during input; adjusting this value can alter the efficiency of data access, and possibly enhance processing times. By default, nmrPipe chooses its own buffer count.

−obuff *outputBuffCount*
This option selects the number of 1D vectors which will be buffered during output; adjusting this value can alter the efficiency of data access, and possibly enhance processing times. By default, nmrPipe chooses its own buffer count.

−buff *ioBuffCount*
This option selects the number of 1D vectors which will be buffered during both input and output, as specified by the **−obuff** and **−ibuff** options.

−mt *maxTries*
This option sets the count for the maximum number of times an nmrPipe process can re−try to read or write data. The default value of −1 allows an unlimited number of re−tries. This option is intended for testing and debugging purposes.

−to *uSecTimeOut*
This option sets the delay time in microseconds between re−tries of **nmrPipe** requests to read or write data. The default value is 512 uSec. This option is intended for testing and debugging purposes.

–noSig This flag will prevent **nmrPipe** from handling interrupt signals on its own; it is included for testing and development purposes.

–nofs When used with the **–in** *inName* or **–out** *outName* options, this flag will suppress any File Size Checking. Without the **–nofs** flag, nmrPipe will attempt to check if the actual input file size matches the size indicated by the header, and it will also attempt to check if the output file system has sufficient space to store the result. The **–nofs** flag is intended for use in situations where the File Size Checking can't be performed properly; this may be the case in certain multi–vendor networks.

–tty This flag will allow **nmrPipe** to send or receive data directly from a terminal screen or keyboard, like a normal UNIX filter. If this flag is not given, **nmrPipe** will abort if either its input or output seems to be directed to a terminal. Note that in almost every case, if nmrPipe is sending or receiving spectral data from a terminal, something is probably wrong with the processing scheme (a missing input or output option, or trailing blanks after the "\" continuation character). The **–tty** flag is intended as a work–around in situations where the input/output type can't be determined correctly by the program.

EXAMPLES

The examples listed here are exclusively for 2D applications; for 3D and 4D applications, see **xyz2pipe** and **pipe2xyz**.

This example shows typical processing for a 2D States–Mode spectrum:

```
#!/bin/csh

nmrPipe –in test.fid \
| nmrPipe –fn SP –off 0.5 –end 0.98 –c 0.5     \
| nmrPipe –fn ZF                               \
| nmrPipe –fn FT –verb                         \
| nmrPipe –fn PS –p0 42.0 –p1 –84.0 –di        \
| nmrPipe –fn TP                               \
| nmrPipe –fn SP –off 0.5 –end 0.98 –c 0.5     \
| nmrPipe –fn ZF                               \
| nmrPipe –fn FT –verb                         \
| nmrPipe –fn PS –p0 90.0 –p1 0.0              \
   –di –ov –out test.ft2
```

This example shows 2D inverse processing of the data created above, to restore the hypercomplex 2D States–Mode time–domain data; note the use of the option **–ad**, which allows complex data to be converted to hypercomplex data:

```
#!/bin/csh

nmrPipe –in test.ft2 \
| nmrPipe –fn HT                                      \
| nmrPipe –fn PS –p0 90.0 –p1 0.0          –inv     \
| nmrPipe –fn FT –verb                     –inv     \
| nmrPipe –fn ZF                           –inv     \
| nmrPipe –fn SP –off 0.5 –end 0.98 –c 0.5 –inv –ad \
| nmrPipe –fn TP –hyper                               \
| nmrPipe –fn HT                                      \
| nmrPipe –fn PS –p0 42.0 –p1 –84.0        –inv     \
| nmrPipe –fn FT –verb                     –inv     \
| nmrPipe –fn ZF                           –inv     \
| nmrPipe –fn SP –off 0.5 –end 0.98 –c 0.5 –inv     \
   –ov –out test2.fid
```

This example shows typical processing for a 2D TPPI spectrum, as acquired on a Bruker AM Spectrometer in sequential acquisition mode, with automated baseline correction in the acquisition axis performed after the 2D Fourier processing. Note that both the Bruker and Real Fourier Transform modes reduce the size of the data by two.

```
#!/bin/csh

nmrPipe −in test.fid \
| nmrPipe −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5 \
| nmrPipe −fn ZF                                  \
| nmrPipe −fn FT −bruk                            \
| nmrPipe −fn PS −p0 −110 −p1 −150  −di −verb     \
| nmrPipe −fn TP                                  \
| nmrPipe −fn SP −off 0.4 −end 0.95 −c 0.5        \
| nmrPipe −fn ZF                                  \
| nmrPipe −fn FT −real                            \
| nmrPipe −fn PS −p0 −31 −p1 62 −di               \
| nmrPipe −fn TP                                  \
| nmrPipe −fn POLY −auto                          \
     −out test.ft2 −verb   −ov
```

This example shows processing used for a 2D States−Mode NOE spectrum, acquired with sequential (Bruker) detection. It features polynomial solvent filtering, and separate automated baseline corrections for data to the left and right of the solvent signal:

```
#!/bin/csh

nmrPipe −in test.fid \
| nmrPipe −fn SHUF −c2ri                        \
| nmrPipe −fn POLY −time −ord 6                 \
| nmrPipe −fn SHUF −ri2c                        \
| nmrPipe −fn SP −off 0.5 −pow 2 −c 0.5         \
| nmrPipe −fn FT −bruk                          \
| nmrPipe −fn PS −p0 −14.2 −p1 13.2 −di −verb   \
| nmrPipe −fn TP                                \
| nmrPipe −fn SP −off 0.5 −pow 2 −c 0.5         \
| nmrPipe −fn ZF −auto                          \
| nmrPipe −fn FT                                \
| nmrPipe −fn REV                               \
| nmrPipe −fn PS −p0 14.0 −p1 −24.0 −di −verb   \
| nmrPipe −fn TP                                \
| nmrPipe −fn POLY −auto −xn 5.3ppm             \
| nmrPipe −fn POLY −auto −x1 5.1ppm             \
   −out test.ft2 −verb −ov
```

In this DQF−COSY example, time−domain phase correction is used to shift the solvent signal closer to zero frequency, where it can be removed more effectively by time−domain convolution. After this, the shift is removed by a second phase correction, and a tentative linear baseline correction is performed after the transform of the acquisition dimension. A more thorough correction is made after the 2D Fourier processing is done:

```
#!/bin/csh

nmrPipe −in dqcosy.fid \
| nmrPipe  −fn PS −ls 9                           \
```

```
| nmrPipe  −fn SOL                                 \
| nmrPipe  −fn PS −rs 9                            \
| nmrPipe  −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5 \
| nmrPipe  −fn ZF −auto                            \
| nmrPipe  −fn FT                                  \
| nmrPipe  −fn PS −p0 255.6  −p1 78.0 −di −verb    \
| nmrPipe  −fn BASE −nw 3 −nl 0% 5% 95% 100%       \
| nmrPipe  −fn TP                                  \
| nmrPipe  −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5 \
| nmrPipe  −fn ZF −size 2048                       \
| nmrPipe  −fn FT                                  \
| nmrPipe  −fn PS −p0 −90 −p1 0 −di −verb          \
| nmrPipe  −fn TP                                  \
| nmrPipe  −fn POLY −auto −xn 5.3ppm               \
| nmrPipe  −fn POLY −auto −x1 5.1ppm               \
   −out test.ft2 −verb −ov
```

This example shows processing for a 2D plane from an HN−detected 3D spectrum, with numerical solvent suppression and extraction of the amide proton region (the left half) of the acquisition dimension:

```
#!/bin/csh

nmrPipe −in fid/test001.fid \
| nmrPipe −fn POLY −time                          \
| nmrPipe −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5 \
| nmrPipe −fn ZF −auto                            \
| nmrPipe −fn FT                                  \
| nmrPipe −fn PS −p0 43  −p1 0.0                  \
| nmrPipe −fn EXT −left −di  −sw −verb            \
| nmrPipe −fn TP                                  \
| nmrPipe −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5 \
| nmrPipe −fn ZF −auto                            \
| nmrPipe −fn FT                                  \
| nmrPipe −fn PS −p0 −135 −p1 180 −di             \
   −out hnco001.ft2 −verb   −ov
```

This example shows a slightly different processing for a 2D plane from an HN−detected 3D spectrum; in this scheme, a specific range of chemical shifts is extracted from the HN dimension:

```
#!/bin/csh

nmrPipe −in fid/test001.fid \
| nmrPipe −fn SOL                                    \
| nmrPipe −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5    \
| nmrPipe −fn ZF −auto                               \
| nmrPipe −fn FT                                     \
| nmrPipe −fn PS −p0 43  −p1 0.0                     \
| nmrPipe −fn EXT −x1 5.5ppm −xn 11ppm −di −sw −verb \
| nmrPipe −fn TP                                     \
| nmrPipe −fn SP −off 0.5 −end 0.98 −pow 2 −c 0.5    \
| nmrPipe −fn ZF −auto                               \
| nmrPipe −fn FT                                     \
| nmrPipe −fn PS −p0 −135 −p1 180 −di                \
   −out hnco001.ft2 −verb   −ov
```

This example shows re–phasing of the acquisition dimension of the 2D spectrum *hnco001.ft2* produced by the previous example; Hilbert Transformation is used to reconstruct the imaginary data, and the result file replaces the source data:

```
#!/bin/csh

nmrPipe   –fn TP –in hnco001.ft2  \
| nmrPipe –fn PS –ht –p0 90.0 –di \
  –verb   –inPlace –out hnco001.ft2
```

The example shows conversion of the file *hnco001.ft2* for use with **nmrPipe** on a byte–swapped architecture:

```
#!/bin/csh

nmrPipe –fn SHUF –bswap –in hnco001.ft2 \
 –out hnco001.ft2 –outSwap –inPlace
```

SEE ALSO

M(1), **bin2pipe**(1), **bruk2pipe**(1), **nmrDraw**(1), **pipe2xyz**(1), **sethdr**(1), **showhdr**(1), **simSpecND**(1), **simTimeND**(1), **var2pipe**(1), **xyz2pipe**(1)

DIAGNOSTICS

All help text, informational messages, and error messages are sent to standard error. Many error messages are intended to be self–explanatory, but some are included for development and testing purposes. Some common problems are listed here:

C–Shell: processing scripts must be explicitly identified as C–shell scripts in order to run properly. They should always begin with the following text as the first characters of the first line:

```
#!/bin/csh
```

Continuation Characters: probably the most common problem encountered when using **nmrPipe** concerns the use of the backslash character "\" to continue a command onto the next line of a shell–script. This use of the backslash character will only work correctly if there are *no trailing blank characters* at the end of the line.

Arguments: nmrPipe will not issue an error message if it encounters an unknown command argument; therefore, another common source of problems are mis–spelled command arguments, command arguments with dash characters "–" in the wrong places, or command arguments with the wrong combination of upper– or lower–case letters.

Input/Output: problems here include use of an input file that doesn't exist, use of an output file in a directory that doesn't exist, or insufficient disk space to save a result. Error messages will also be generated if the output file already exists, but the overwrite flag **–ov** has not been used.

Processing Schemes: in some cases, the processing scheme itself may not be appropriate; for example use of the wrong kind of Fourier Transform, or not deleting imaginary data where appropriate. When possible, **nmrPipe** will try to detect such problems, and issue warning messages.

Conversion Parameters: if the spectrometer format data has not been converted correctly, processing may complete without error messages but still give the wrong result. Pay close attention to the sizes and acquisition modes of all the dimensions when converting the data.

LEGAL NOTICE