`libcmatrix` (release 3)

(November 15, 2005)

Contents

1 Introduction

`libcmatrix` is a C++ library of classes and functions designed for numerical studies of problems in NMR. Its major current application is as the back-end of a SIMPSON-like general simulation program, provisionally titled NMRsim. Most the functionality of the library can be accessed in a user-friendly fashion via NMRsim and so most users will have little call for dealing with the underlying library directly. As a consequence, this library documentation has trimmed of its more tutorial aspects and the description of the higher-level functionality is left rather vague since this functionality is both most subject to change and of less general application. Differences from release 2 are noted in the margins and Section 9.1 contains information on significant changes which may affect the compilation of existing programs.

The `libcmatrix` source directory is divided into the following subdirectories:

`coredefs` Definitions and data types, such as `cmatrix` that are basic to `libcmatrix`.

`NMR` Functions and datatypes that are particular to NMR simulations.

`utils` Useful functions and data types that do not fit neatly into a particular module e.g. Fourier transforms, input/output etc.

`optim` Code for non-linear model-fitting and optimisation.

`local` Local and/or temporary additions/modifications. *This directory will be generally be absent in public releases.*

`test` Test programs/example programs. These can be compiled using `make` *program*.

`contrib` Fragments that are nominally compatible with the library but have either been removed or not incorporated.

`docs` Documentation.

1.1 Using libcmatrix

The `cmatrix` source directory contains a template Makefile, `Makefile_template`. The important Makefile variables are

`FLAGS` Two sets of values of values, one for when when debugging, and the other for when the program is working properly which turns on optimisation and disables certain error checking (discussed in more detail in Section 4.3).

`LIBCMATRIX` is the path to the source directory. This is required for both the header files and compiled libraries.

`LIBRARIES` is the list of libraries that linked with compiled program. This must, of course, include `libcmatrix` itself!

`libcmatrix` programs are generally straightforward to debug. The great majority of problems are caught as part of argument checking and result in "exceptions" being thrown. Unless a program sets up its own exception handling this causes the program to core dump. Running the program in a debugger *may* show where the error occurred, although with some systems, notably g++/gdb, it is not possible to examine the state of the program after an exception i.e. the debugger must be used to trap execution before the error occurs.

If the program crashes "badly" (mysterious segmentation faults etc.), make sure that it has been compiled without `-NDEBUG` which has the effect of disabling range checking etc. Deleting the `-NDEBUG` should cause the recompiled program to fail with a well-defined `BadIndex` exception. Depending on the run-time system, exceptions may cause programs to terminate with an unhelpful core dump. In this case, it makes sense to trap `libcmatrix` exceptions and print the error message e.g.

```
try {
code...
} catch (MatrixException& exc) {          catch exceptions from libcmatrix
    std::cerr << exc << std::endl;        print error message
}
execution continues here
```

Alternatively, the individual exceptions derive from the standard exception classes, cf. Table 12.

Unless you are writing your own "dangerous" code e.g. using pointers, unchecked array accesses etc., nasty segmentation-violation type crashes are very unlikely.

1.2 Combining `cmatrix` with other code

C++ and C are link-compatible, that is to say, C routines can be linked directly with C++. Linking C (and hence C++) with FORTRAN is significantly more painful. The details (including compilation options) vary with system and compiler, but simple numerical types, 1D-arrays etc. can generally be passed without too much difficulty. The data in `cmatrix` and `rmatrix` types is stored as a simple array which can be passed to FORTRAN (or C) routines, although such routines cannot manipulate or create the full `cmatrix` data structure. Since complex numbers are stored as double precision numbers, the `cmatrix` arrays can be manipulated with "`z`" functions of LAPACK, and `rmatrix` arrays with the "`d`" functions. It important to remember that the matrices will appear to FORTRAN as their transposes, since FORTRAN uses column rather than C's row ordering. Fortunately, these problems are becoming less relevant as more code is re-implemented in C/C++ e.g. the MINUIT optimisation library.

`libcmatrix` can be configured at compile time to use an external optimised library (e.g. ATLAS) for key routines such as matrix multiplication. Optimised versions perform much better and will, for larger matrices at least, outperform the native `libcmatrix` routines. ATLAS provides a "self-tuned" version of core linear algebra routines for arbitrary architectures and delivers much better performance for large problems ($>\sim 16\times16$) over the native `libcmatrix` routines.

2 General features

2.1 Standard functions and arguments

Functions that manipulate objects of a certain type can be divided into two categories: "in-place" functions that modify one of the input arguments, and functions that create a new data object to hold the result. For example, the increment operator, `A+=B`, stores the result of adding `A` and `B` back in `A` and so is an "in-place" operation. Such operations are often very efficient, but certain operations, such as matrix multiplication, cannot be performed in-place and so a third (output) matrix is required.

Care is taken to ensure that the nature of functions is unambiguous. In-place functions, such as `+=` are naturally coded as member functions, hence `A.conj()` is the function that turns `A` into its complex conjugate. Such members fcuntions generally return `void` which prevents ambiguous code such as `B=A.conj()` (is `A` first turned into its complex conjugate and copied into `B` or is `B` simply set to the conjugate of `A`?)[1]. However, it is not appropriate to use member functions for every in-place operation. For instance, a Fast Fourier Transform is not really an elementary operation of the `cmatrix` data type, and including every possible in-place operation would clutter the `cmatrix` type with many functions from high-level modules. In these cases, a global function is used with suffix `_ip`. Hence `fft_ip(A)` would replace `A` by its FFT.

Other functions return a result in a new object. This can either be passed as an additional argument to the function, or the function can return a new data object which is created as a temporary variable. The latter method has the advantage of simplicity of appearance; complex expressions can be written such as `A=B*C+D` and the compiler will create the necessary temporary objects to store the result of `B*C` and the addition operation. This is the way that languages such as *Matlab* operate, and operators like `*`, `+` etc. always return new temporary objects. The drawback of this approach is that the creation, followed by the destruction, of the temporary variable can involve significant overhead e.g. for small matrices. Note that compilers do not necessarily warn that results are being ignored, so `fft(a)` may compile without warnings even though the result of `fft` is being ignored!

A more efficient, if less visually elegant, solution is to use functions such as `multiply(C,A,B)` that put the result of multiplying `A` and `B` in `C`. The previous contents of `C` (if any) are overwritten. If this code is part of a loop, `C` will probably already be an output variable of the correct dimensions, thus eliminating the need for any re-allocation of memory. Like in-place functions, such "supplied-destination" functions are declared `void`. In many circumstances, the input matrices `A` or `B` cannot also be used as output matrices, in which case an `ArgumentClash` exception is thrown.

The standard *order* for arguments is[2]

`function`(output argument (if any)`,` input arguments`,` any optional arguments)`;`

Functions that return information, such as size, about an object fall into a slightly different category. In normal C++ usage, such functions would normally be member functions e.g.

[1] However, functions such as `+=` (e.g. `A+=B`) are conventionally defined not `void` and return the result *after* the operation

[2] The opposite convention is used in the standard library...

Operation	in-place	new-object	supplied-destination	*Matlab* equivalent
Addition	`A+=B`	`A+B`	`add(C,A,B)`	`A+B`
Subtraction	`A-=B`	`A-B`	`subtract(C,A,B)`	`A-B`
Negation	`A.minus()`	`-A`	`minus(C,A)`	`-A`
(Matrix) multiplication	`A*=B`	`A*B`	`multiply(C,A,B)`	`A*B`
Division	`A/=B`	`A/B`	`divide(C,A,B)`	`A/B`
(Element) multiplication	`A.emultiply(B)`	`emultiply(A,B)`	`emultiply(C,A,B)`	`A.*B`
Similarity transformation	`A.simtrans(B)`	–	`simtrans(C,A,B)`	`inv(B)*A*B`
Inverse sim. transf.	`A.isimtrans(B)`	–	`isimtrans(C,A,B)`	`B*A*inv(B)`
Conjugation	`A.conj()`	`conj(A)`	`conj(C,A)`	`conj(A)`
Transposition	`A.transpose()`	`transpose(A)`	`transpose(B,A)`	`A.'`
Multiply and add	`mla(C,A,B)`	–	–	`C=C+A*B`
Real part	–	`real(A)`	–	`real(A)`
Imaginary part	–	`imag(A)`	–	`imag(A)`

Table 1: Names used for standard operations. "In-place" functions modify the input matrix `A`. "New-object" functions return a new object (e.g. a new `cmatrix`) containing the result of the operation. "Supplied-destination" functions put the result into the object supplied as the first argument (`C`).

`A.sum()` might be used to return the length of a vector. This avoids cluttering the global namespace with functions that are only relevant to a particular data type. In mathematical usage, however, the notation `sum(A)` is more readable and since the same function might be applied to several different data types, it makes sense to define these as global functions. *Low-level* functions are always member functions (if they are visible at all). Functions that start `isXXXX` invariably return boolean `true`, `false` information e.g. `issquare`.

A standard set of names are used for common operations that are shared between objects. These are listed in Table 1 in the various different ways that they can called. Some of these combinations are never defined e.g. there is no "new-object" similarity transform function. This is because it is rare for similarity transformation to be part of a more complex expression e.g. `C=simtrans(A,B)+D` and this form is unsuitable in critical loops since it involves the creation of an additional temporary. The "multiply and accumulate" operation, `mla`, is a little unusual, but the expression `C=C+A*B` is often encountered, and so it is useful to define this function for some key data types. Data types are not obliged to define all these functions, e.g. transposition for scalar types.

Where common combinations of operations can be written more efficiently as a single operation, the function names are combined with an underscore, so `f_g(A)` is equivalent to `f(g(A))`. For example, the conjugate transpose operation can be written `conj(transpose(A))` but the equivalent function `conj_transpose(A)` may also be defined.

Some functions have special forms for specific inputs e.g. if the matrices involved are unitary, hermitian etc. These special forms are written `hermitian_X` etc., e.g. `unitary_simtrans` should be used when the transformation matrix is unitary. Note that you are responsible for ensuring that the matrices are of the declared type e.g. the function `ishermitian(`*A*`,`*tol*`)`

returns true if the difference between $a_{i,j}$ and $^{*}_{ji}$ exceeds the specified tolerance (which can be zero to check for exact hermiticity).

2.2 "undefined" and non-dynamic objects

Some objects have an "undefined" state, created by a empty constructor, that can subsequently be "filled in". This is the case for example, for the matrix data types. and so functions check that their inputs are defined using `empty()` (container types e.g. `Matrix`) or the `!` operator for "objects". Passing an undefined matrix as an *output* argument is perfectly acceptable.

Some other data types need to be fully constructed when the object is initialised. For instance, the `spin_system` type has no null constructor, and the number of spins must be specified when the object is created. This number cannot be changed subsequently. Although simple C-style arrays can be created from objects without a null constructor, the `libcmatrix` (and most of the STL) storage classes can be used on such objects e.g. `List<spin_system> list(5,spin_system(2,"1H"))` creates a list of 5 identical `spin_system` objects. Modified

In most cases, the data storage for a container object is allocated "dynamically". In some circumstances, however, the memory may have be allocated by another object. The "flat" memory allocation of a $r \times s \times t$ three-dimensional matrix, for example, means that, in effect, the first $s \times t$ data items correspond to the first 2D matrix "slice" in the 3D matrix, the next to the second and so on. These 2D slices can be returned efficiently as "views", if we can create a matrix object that refers to this independently-allocated memory. The `libcmatrix` container objects(`List`, `Matrix` and `MultiMatrix`) can be flagged as "non-dynamic" when created, in which case the data space is expected to be managed externally. New!

There is distinction between initialising a matrix from a non-dynamic matrix and assignment. If `A` is a non-dynamic `cmatrix`, for example, the result of `cmatrix B(A)` (or `cmatrix B=A;`) will be a new non-dynamic matrix pointing to the same data space as `A`. The initialisation will be very quick, but it is important to realise that `A` and `B` are referring to the same data! On the other hand, `cmatrix B; B=A;` will first create a normal matrix and then copy the contents of `A` into `B`. `B` and `A` are completely independent.

Like the view objects described below, attempts to pass temporary non-dynamic objects as variable arguments to functions will often generate compilation warnings and the compiler may refuse such code entirely. The solution is to create a non-temporary non-dynamic object from the temporary i.e. `cmatrix B(`*my_function*`);` *using_function*`(B,...)` rather than *using_function*`(`*my_function*`,...)`.

The memory allocation for a non-dynamic object must be supplied when the object is created and is then fixed. Attempts to change the size of the object or to explicitly release the memory (i.e. `clear`) generate exceptions.

Non-dynamic objects should only be used when necessary as they can easily lead to "aliasing" problems where apparently distinct objects actually refer to the same data. Use references whenever possible.

Operation	in-place	new-object
addition, `+`	X	X
subtraction, `-`	X	X
negation	X (`minus`)	X (`-`)
multiplication, `*`	X	X
division, `/`	X	X
`conj`	X	X
`mla`	X	–
`real`	–	X
`imag`	–	X

Table 2: Standard operations defined for `complex` data type indicated by X. Note that R indicates the option of a form taking a real argument.

3 The complex data type

The complex data type is defined in the header file `cmatrix_complex.h`. Note that this is different from and independent of other complex implementations, including the templated complex type provided in ANSI C++; including the ANSI `complex` header is not recommended! The complex data type has three constructors: `complex(double` r`,double` i`)` which creates a complex number from its real and imaginary components, `complex(double` r`)` which creates a complex number with a zero imaginary component, and the null constructor `complex()` which creates an "undefined" complex number, where the memory is allocated for the real and imaginary components, but their contents are undefined[3].

The standard operations defined in `cmatrix_complex.h` are shown in Table 2. Most functions have additional forms that handle real arguments efficiently e.g. `complex(2,1)*2.0` is compiled with a function that multiplies a real by a complex. If such real forms do not exist, `double` will be automatically promoted to `complex` and the "full complex" form used. Table 3 lists the additional operations that are specific to the `complex` type. The `expi(`a`)` function, which calculates $\exp(ia)$ where a is real, i.e. $(\cos a, \sin a)$, is a little unusual, but is much more efficient than `exp(complex(0,`a`))`.

4 The `Matrix<T>` class

The `Matrix<T>` class is a general matrix type about which most of the rest of the library is built. The `rmatrix` and `cmatrix` types, used for real and complex matrices, are in fact `Matrix<float_t>` and `Matrix<complex>` respectively. This section describes operations of the `cmatrix` type, but the same operations are (mostly) applicable to the general type.

Note that "mixed arithmetic" is supported e.g. multiplying an `int`-based object with a `double`-based object. Provided the data types are "known", the compiler can deduce the correct output type for a given input combination, for instance, `Matrix<int> * Matrix<double>` will create a `Matrix<double>`. Modified

[3]The empty constructor in some other definitions of the complex data type sets the components to zero.

C complex(), complex(R), complex(C)
R norm(C)
R arg(C)
C polar(R,R)
R abs(C)
C [sin,cos,tan](C)
C [sinh, cosh, tanh](C)
C exp(C)
C expi(R)
C pow(R,C), pow(C,R), pow(C,C)
C sqrt(C)

Table 3: Additional complex functions. R denotes a real argument/result, C a complex one.

There is partial support for types that lack null constructors. For instance `Matrix<spin_system> a(3,2)` is impossible since `spin_system` has no default constructor. By using functional forms which take a constructed object e.g. `Matrix<spin_system> a(3,2,spin_system(2,"1H"))` it is possible to create a `Matrix` (or `List`) of such objects. Many matrix functions assume the presence of a default constructor and so will not work for such objects, but these functions are rare applicable to such objects.

4.1 Complex matrices

The most important data type is the complex matrix, `cmatrix`, which is shorthand for `Matrix<complex>`. Most functions are provided by the generic `Matrix<T>` functions, but operations that are particular to complex matrices e.g. conjugation, are declared in the header file `cmatrix.h`.

The descriptions of functions do not necessarily show the full prototype definition for a function, hence for `issquare(cmatrix` *A*`)`, the details of how *A* is passed to `issquare` are omitted. In fact, the full definition of `issquare` is `issquare(const cmatrix&)` i.e. the matrix is passed as a (constant) reference, which is the most efficient way of passing the matrix. By contrast the presence of the reference, &, in `read_matrix(&`*matrix*`,char*` *fname*`)` indicates that *matrix* is altered by the function (it is overwritten by the contents of *fname*).

4.2 Creation and deletion of `cmatrix`

The simple constructors are:

`cmatrix()` creates an undefined complex matrix.

`cmatrix(int` *rows*`,int` *cols*`)` constructs an uninitialised *rows* by *cols* matrix. Both *rows* and *cols* must be greater than zero.

`cmatrix(int` *rows*`,int` *cols*`,complex` *value*`)` constructs a *rows* by *cols* matrix, initialised to a constant *value*.

`cmatrix(int` *rows*`,int` *cols*`,complex*` *addr*`)` creates a matrix from a list of elements. The elements are copied from the vector with the column assumed to vary most rapidly, i.e. standard C order. This is the only way to initialise a matrix with specific values in source code. It is also useful for constructing a `cmatrix` from a C/FORTRAN supplied data array (the FORTRAN matrix will be transposed).

`cmatrix(cmatrix)` is the simple copy operation[4].

`cmatrix(Matrix<T>)` creates a complex matrix from a matrix of another type e.g. `Matrix<double>`.

In addition to the constructors, there are a couple of `create` member functions which can be used to (re)create an existing matrix: `create(int` *rows*`,int` *cols*`)` and `create(int` *rows*`,int` *cols*`,complex*` *addr*`)`. So `A.create(5,5)` would turn `A` into a 5×5 matrix initialised to zero; its contents are undefined, however. `A` must, of course, have previously been *declared*, but need not have been initialised. Obviously, if `A` is already a 5×5 matrix, the existing allocated memory will be re-used. If `A` is a different size, the memory will be released, and a new matrix allocated.

In common with other "container" objects, matrices have a `swap` member function: `A.swap(B)` swaps the contents of objects `A` and `B`. This operation is quick since only pointers to the data and not the data itself are swapped. The memory allocated to containers can be released using the `clear()` member function, which returns the `cmatrix` to the undefined state[5]. Matrices (if defined) can be set to a constant value using the `=` operator e.g. `A=complex(5,4)` sets the elements of `A` to $5 + 4i$.

4.3 Information functions

The following member functions return information about a `cmatrix` object:

`size_t rows()` and `cols()` returns the number of rows or columns of a matrix. These can be assumed to be zero for an undefined matrix.

`size_t size()` returns the total number of data elements i.e. `rows()*cols()`[6]. Modified

`T* vector()` returns the address of the data considered as a vector. This should not be abused!

`T* vector(size_t r)` returns the address of row r of the data. Again, to be used with care! New!

`bool empty(`*matrix*`)` returns `true` if *matrix* is undefined, that is, no memory is allocated for elements.

`bool issquare(`*matrix*`)` returns `true` if the matrix is square.

[4]The new matrix is completely independent of the source matrix. Some matrix implementations use virtual copies so that the source and destination are actually the same matrix. Use references for this in `libcmatrix`.

[5]This function has been re-named from `kill` for consistency with the STL.

[6]Renamed from `length` for consistency with the STL.

Matrix elements and submatrices are accessed using the `()` operator e.g. `A(3,5)` returns the element in the third row and fifth column of `A` [7]. The `NDEBUG` symbol determines whether bounds checking is applied. In normal compilation, `NDEBUG` is not defined and bounds checking is enabled. If `A` was a 3×3 matrix, the above array access would generate a `BadIndex` exception when the statement was encountered. Obviously such range checking is extremely useful for debugging but considerably slows execution. Compiling the (tested!) program with `-DNDEBUG`, suppresses bounds checking and element access is inlined to direct array access.

Selection of submatrices can be done in a couple of ways. They can either be specified in terms of the `slice` or `range` objects which defines a regularly spaced sets of indices, or using a `List<size_t>` to list the indices required.[8] The standard type `size_t` is used for row/column indices. Usually it is not necessary to worry about the type used for indices since the integer types are automatically interconverted. It is more important, however, when using list objects, since a list of `unsigned` will not be converted automatically to a list of `int` etc. In both cases, combination of a matrix and a selection creates an object which provides a "view" onto the original matrix. This can then be used to create a new matrix object (i.e. a submatrix of the original), or to manipulate the specificed rows/columns of the *original* matrix. Although the functionality of `slice` can be duplicated with a `List<size_t>`, the `slice` is somewhat more efficient.

`range` is a specialisation of `slice` which fixes the stride at 1. This is useful for selecting ranges which are (by definition) contiguous e.g. complete rows or columns of matrices.

`slice(`*start*`,`*size*`,`*stride* `=1)` where *start* is the starting index (from 0), *size* is the number of elements in the slice and *stride* is the step from one element to the other[9]. For instance `stride(1,3,2)` will select indices 1, 3, 5.

`range(`*start*`,`*end*`)` selects the index values *start* to *end* (inclusive). When used with matrices, an empty `range` object refers to the entire row/column e.g. `A(range(0,1),range())` would correspond to the *Matlab* `A([1:2],:)`.

`slice` and `range` objects may be multiplied (scaled) by a positive number i.e. `range(0,3)*4` would make a new `slice` object for the indices 0, 4, 8, 12. `slice` objects (but not `range`) support in-place scaling. Positive offsets can be added to both objects. They can also be used as function objects taking an input `size_t` and returning the corresponding `size_t` index, so `range(0,3)(2)` would return the 3rd element (indexing from 0!) of the selection i.e. 2.

[7] Previous releases, following `Matlab usage`, permitted matrices to be addressed directly as one-dimensional vectors i.e. `A(3)` would return the fourth element in the data storage for `A`. This functionality is provided more cleanly by `row()` or the `begin()`, `end()` iterators, and conflicts with the `MultiMatrix` usage where `A(3)` would refer to the fourth *row* of `A`.

[8] Note that `A(i,j)` will only reduce to a direct element access if `i` and `j` are supplied as `size_t` i.e. unsigned integers. Integers are by default signed, so `A(1,2)` when compiled without optimisation will result in a few layers of functions, which is confusing when debugging! This can be avoided using `A(1U,2U)` which indicates the numbers are unsigned, or by defining the pre-processor symbol `LCM_SUPPRESS_VIEWS` which disables all 'exotic' uses of `()`.

[9] The `slice` object is analagous, but not necessarily identical, to the `slice` object of the Standard C++ Library. For instance, the latter may not permit negative strides. They are defined in different namespaces so there is no direct clash.

A `slice` or a `List<size_t>` is used together with (*rowsel*, *colsel*) to define a row and/or column selection. If both *rowsel* and *colsel* are "list" objects of some form (e.g. `slice`, `BaseList<size_t>`, the result is an `IndirectMatrix<T,`*rowsel*`,`*colsel*`>` object, although it is not usually necessary to know the exact type being returned as "view" objects are generally used immediately, e.g. `A(2,range())=2`. The selection objects can be any type that has the same "form" as a `BaseList<size_t>`.

A`(`*rowsel*`,`*colsel*`)` defines a general submatrix, returned as an `IndirectMatrix`.

A`(`*rowsel*`,`*col*`)` create a one-dimensional view for a column (as an `IndirectList`).

A`(`*row*`,`*colsel*`)` creates a one-dimensional view onto the specified row. Result is `IndirectList`.

A`(`*row*`,range(`*colsel*`))` is a special case which returns a simple `BaseList` since the selection refers to a contiguous list of elements. *A*`(`*row*`,range())` is equivalent to *A*`.row(`*row*`)`.

A`.diag(`*n* `=0)` returns a view onto the *n*th diagonal e.g. `A.diag()=0` would set the diagonal elements to zero. Positive *n* refers to upper right diagonals, negative *n* to diagonals in the lower triangle.

It is important to realise `IndirectMatrix` objects refer to the original matrix (which must remain in scope over the lifetime of the object!). Operations applied to them will the selected elements of the source matrix. Valid operations are

`=` *b* replaces each elements of the defined submatrix by the value *b*.

`=` $\boldsymbol{B}$ replaces the submatrix with corresponding elements of a general matrix $\boldsymbol{B}$

`+=` *b* adds *b* to each element of the submatrix. Also `-=`, `*=`.

`+=` $\boldsymbol{B}$ adds the corresponding element of $\boldsymbol{B}$ to the submatrix. Also `-=`.

$\boldsymbol{B}$ `=` creates a new matrix $\boldsymbol{B}$ from the submatrix. $\boldsymbol{B}$ is independent of the original matrix and so subsequent operations on $\boldsymbol{B}$ will not affect the original matrix. Note that $\boldsymbol{B}$ may be another "view" object, but operations that involve the same matrix as source and destination are illegal.

Iterators are also defined for these objects, which allows additional functions to be defined cf. Section 4.4.

One-dimensional views using `IndirectList<T,`*sel*`>` work in the same way but are more efficient and operate with `List<T>` rather than `Matrix<T>` objects.

Using a "view" object as an output argument to other functions is problematic. If the function is expecting, say, a `cmatrix`, a temporary `cmatrix` will be created (from the view object) in which the results are placed before it is destroyed! Some compilers refuse to compile such code entirely. For instance `eigensystem(A(range(0,3),D,B))` will pass the upper-right 4×4 diagonal block of `A` to `eigensystem` as a new matrix, which is overwritten by the eigenvalues of `B`. To put the eigenvalues of `B` in a submatrix of *A*, you need a temporary matrix e.g. `eigensystem(tmp,D,B); A(range(0,3))=tmp`. On the other hand

Operation	in-place	new-object	supplied-destination
`add`	TX	TX†	X
`subtract`	TX†	TX	X
`minus`	–	X†	X
`multiply`	TX†	RCX†	X
`emultiply`	X	X†	X
`divide`	T	T	–
`conj`	X	X†	X
`mla`	RC	–	–
`simtrans`	X‡	–	X†
`isimtrans`	X‡	–	X†
`inv`	–	X‡	X†
`transpose`	X	X†	X
`real`, `imag`	–	X†	X

Table 4: Standard operations defined for the `cmatrix` data type. Most functions accept only complex scalars, T, an exception being multiplication. Functions marked with † require the creation of a temporary variable.

`eigensystem(A,D,B(range(0,3)))` will have the effect of diagonalising a 4×4 block of `B` since the fact the creation of a temporary *input* matrix is harmless. Some (templated) functions may take "generic" matrices, in which case temporaries are not passed, and a view object will work as expected for input or output arguments

4.4 Relation to STL containers

The Standard Template Library (now part of the standard C++ library accompanying compilers) is largely based around a collection of "container" classes and functions that manipulate them. These classes are mostly aimed at "computing" applications, but the `valarray` and `vector` types are quite similar to `List`[10]. The `libcmatrix` types share many of the characteristics of the STL types and the types can often be used interchangeably. In contrast to `libcmatrix`, the STL types are always single-dimensional and so a complete one-to-one correspondance is not possible, particularly in respect of object creation and resizing.

On the other hand, all `libcmatrix` containers provide "iterators" (via `begin()` and `end()`) and so can be used in standard algorithms (see `test/testiter.cc` for some examples). Hence the STL algorithms can now be relied on for standard operations such as sorting. As most objects (with the single exception of `List`) cannot be dynamically resized, however, it is not possible to create "insert iterators" (e.g. `front_inserter`) from `libcmatrix` types.

4.5 Mathematical operations

The standard operations defined for `cmatrix` are shown in Table 4. Some functions need a little more in the way of comment:

[10]The STL `list` container involves linked lists and so does not resemble the `libcmatrix List`.

- The mathematical functions accept scalar arguments when this is appropriate, for instance `A+2.0`, or `add(B,A,2.0)`, adds $2 + 0i$ to each element of `A`.
- The `+=`, `-=` and `mla` operators are unusual in accepting an undefined matrix as valid input. The undefined matrix is treated as a zero matrix i.e. `A+=B` becomes `A=B` if `A` is undefined. In the same spirit, `*=` treats an undefined matrix as an identity matrix i.e. `A*=B` becomes `A=B` if `A` is initially undefined. An `Undefined` exception is thrown if `B` is undefined, or if either input matrix is undefined for the `+` and `add` operators.
- Division is only defined for scaling e.g. `A/2`. *Matlab* has special uses for divisions involving matrices.
- `A*=B` is defined as $A = A * B$. Since matrix multiplication is non-commutative in general it the otherwise unused operator `&=` is used (à la GAMMA) for $A = B * A$. Note that in both cases a temporary is required to hold the result of the multiplication.
- The following "composite" functions are also defined: `conj_transpose`, `transpose_multiply(`A`,`B`,`C`)` performs $A = B^T C$, `multiply_transpose`(A,B,C) performs $A = BC^T$, `conj_transpose_multiply(`A`,`B performs $A = B^\dagger C$, `multiply_conj_transpose`(A,B,C) performs $A = BC^\dagger$.
- `real` and `imag` return real (`rmatrix`) rather than complex matrices.

4.6 General operations

Some of the following functions have special forms for particular types of input matrices. Functions beginning `hermitian_` and `unitary_`, for example, are optimised for Hermitian and unitary matrices respectively. These are always quicker than their non-specialised counterparts and may return different result types e.g. the eigenvalues of a unitary matrix are real rather than complex. It is important to note, however, that the results will be meaningless if the matrix is not of the declared type. In some situations it is clear what type a matrix will be (e.g. Hamiltonians must be Hermitian). In others, more care is needed e.g. if using non-Hermitian detection operators when the propagation code uses the `hermitian_trace` function.

`complex trace(`A`)` returns the trace of A, i.e. the sum of its diagonal elements,

`complex trace_multiply(`A`,`B`)` returns the trace of AB[11]. This is very much quicker than evaluating `trace(`A`*`B`)` since it avoids the calculation of the unneeded off-diagonal elements of the matrix product. Note that to 'detect' operator Q, the appropriate detection operator is $Q^\dagger$. Hence for non-Hermitian operators, $\mathrm{tr}(AB)$ is not the projection of A on B, $\mathrm{tr}(AB^\dagger)$. Modified

`complex trace_multiply(List<complex>` A`,cmatrix` B`)` returns the trace of AB where A is a diagonal operator stored as a complex vector.

`double hermitian_trace_multiply(`A`,` B`)` returns the trace of AB assuming that A and B are Hermitian. This function is useful if the observable is real since the calculation

[11] Function renamed from simply `trace` late in R3.

is almost twice as quick as `trace` and is often a rate limiting step in some propagation techniques.

$$\sum_i A_{ii}B_{ii} + 2\sum_{i,j>i} \mathrm{Re}(A_{ij}B_{ji}) \tag{1}$$

`double hermitian_trace_multiply(List<double> A, cmatrix B)` returns the trace of AB where A is the diagonal of a Hermitian detection operator.

`double normsq(A)` returns the square of the norm of matrix A, i.e. $|A|^2$. `normsq` is used to avoid the unfortunate confusion over the meaning of `norm`.

`complex sum(A)` returns the sum of all the elements of matrix A. Note that the *Matlab* `sum` function performs the sum over a single dimension of a matrix i.e. it returns a projection.

`double hermitian_sum(A)` returns the sum assuming that the matrix is Hermitian. The result is undefined is the matrix if not actually Hermitian.

$$\sum_i A_{ii} + 2\sum_{i,j>i} \mathrm{Re}(A_{ij}) \tag{2}$$

A`.identity(int n)` initialises the matrix to an $n \times n$ identity matrix. Note that the global function `identity(n)` (declared in `rmatrix.h`) returns a *real* identity matrix, which can then (if necessary) converted into a complex matrix. Hence, `cmatrix A=identity(5)` is significantly slower than the corresponding definition in terms of member functions `cmatrix A; A.identity(5)`.

4.7 Advanced mathematical operations

The following functions involve more advanced matrix algebra. Many of these functions are only defined for complex matrices. The most efficient functional form is shown, with the presence of other functional forms indicated with: IP (in-place), N (new-object), SD (supplied-destination).

The similarity transforms functions are:

`simtrans(&`B`,`A`,`V`)`† returns VAV^{-1} in B. Other forms: IP.

`unitary_simtrans(&`B`,cmatrix `A`,`V`, cmatrix* =NULL)`† assumes that V is unitary. This is much more efficient than the general form above, since the inverse of V is given simply by its conjugate transpose. Other forms: IP. An optional pointer to a "workspace" matrix can be passed to the SD form which avoids the need for a temporary matrix to be created.

`unitary_simtrans(&`B`,List<T> `A`,`V`, cmatrix* =NULL)` applies the similarity transform to a diagonal matrix stored as a real or complex vector (i.e. `T` can be `double` or `complex`).

These functions are all defined for the inverse similarity transform operation, `isimtrans`, $V^{-1}AV$.

The matrix inversion function, `inv`, is slightly unusual since the supplied-destination form, `int inv(B,A)`, returns an error value rather than being a `void` function. This error value is non-zero if the input matrix is singular. The new-object form, `B=inv(A)`, cannot return an error code and throws a `Failed` exception if `A` is singular. The choice of which form to use depends on whether singularity of the input matrix is really "exceptional".

The syntax of the diagonalisation functions is also unusual since it is necessary to return more than one result i.e. the eigenvalues and eigenvectors.

`eigensystem(cmatrix &`*evectors*`,List<complex> &`*evals*`,`*A*`)`† calculates the eigenvalues and eigenvectors of the complex matrix A. *This function should be used with care for general matrices (those which are not hermitian, orthogonal or unitary), since the eigenvectors are not well-defined for general matrices. In particular, the eigenvector matrix may not be unitary*[12].

`hermitian_eigensystem(cmatrix &`*vectors*`,List<double> &`*evals*`,`*A*`)`† should be used for Hermitian input matrices. The eigenvalues are necessarily real. The diagonalisation of 2×2 Hermitian matrices is coded explicitly for speed.

```
cmatrix eigs;                                  matrix to store vectors
List <double> evals;                           list to store eigenvalues
hermitian_eigensystem(eigs,evals,H);
cout << "Eigenvalues:  " << evals << endl;     output eigenvalues
```

N.B. Diagonalisation is not obliged to respect block structure! If a matrix has degenerate eigenvalues (always true if the Hamiltonian has some symmetry), then the eigenvectors are non-unique. The diagonalisation routine is free to mix the eigenvectors and in doing so may disrupt any block structure. If you have block structure, you should diagonalise the blocks individually. A closely related problem occurs if a matrix should be block diagonal but isn't quite due to numerical round-off. The results in inapopriate mixing of states, particularly if they are almost degenerate. In the worst case, the diagonalisation can be unstable, resulting in eigenvector matrices that are no longer unitary.
Functions which proceed via diagonalisation

`cmatrix pow(`A`,`x`)`‡ returns A^x in B for a general matrix. The `hermitian_pow` function should be used if A is hermitian. x may be complex, real or integer. Other forms: N.

`cmatrix exp(`A`,double` z `=1)`‡ returns the (optionally scaled) matrix exponential $\exp(zA)$. This function will work for any non-singular A but is much less efficient than the `hermitian_exp` function which should be used for Hermitian matrices. The last argument is optional i.e. $\exp(A)$ will return the (unscaled) matrix exponential. *N.B. the equivalent* Matlab *function is* `expm`*, not* `exp`*.*

The `hermitian` variants only also exist in supplied-destination form.

[12] An `eigensystem_controller` structure is defined in `cmatrix.h` which can be used to control warnings and checks on complex diagonalisation.

4.8 Input/output

All `libcmatrix` classes have an output stream definition i.e. `cout << A` will output something comprehensible for any given `A`. The formatting of the numbers is determined by the current stream output flags (see the `iostream` documentation for details), except for matrix output where it is necessary to force fixed width output for readable results. It is a good idea to reduce the number of output decimal places for matrices from the default six using e.g. `cout.precision(3)`.

Output to files involves a choice of file format. The functions supplied with the `cmatrix` library can store matrices (and vectors) as simple streams of binary numbers or as ASCII. Binary formats are quick to read and write, but are not very portable. Matrices can also be stored in "RMAT" format which resembles the simple format used in PGM etc. files. This is a home-grown format so very few programs will read these files! Use `Matlab` or `SIMPSON` formats for better portability.

The following functions output real or complex matrices or vectors to files:

`void write_vector(FILE*` *filep*`,List<complex>` *data*`,int` *flags* `=mxflag::doublep)` writes a vector of complex numbers to an open file pointer. The flags and return values are described below. The default flag settings give files consisting of a list of ASCII numbers to double precision.

`int write_vector(const char* fname,List<complex>` *data*`,int` *flags* `=mxflag::doublep)` opens the file *fname* and writes a vector of complex numbers to it.

`void write_matrix(FILE*` *filep*`,`*matrix*`,const char*` *comment* `=NULL,int` *flags*`)` writes a matrix to an open file pointer. The optional comment string is a set of text lines (separated by newlines) which can be used to describe the output. The default setting of *flags* is `doublep | binary`.

`int write_matrix(const char*` *fname*`,`*matrix*`,char*` *comment* `=NULL,int` *flags*`)` writes the matrix to the specified filename. The default setting of *flags* is `doublep | binary`, i.e. RMAT file containing double precision binary.

`void read_matrix(&`*matrix*`,FILE*` *filep*`)` creates *matrix* from a given file pointer, which should point to the start of a valid *matrix* file format. The raw list of numbers produced by `simple` cannot be read as a matrix, neither can complex matrices stored in `block` form. Trying to read a complex matrix into an `rmatrix` is an error. The contents of the matrix after a failed read are undefined.

`int read_matrix(&`A`, const char*` *fname*`)` creates matrix A from a given filename.

The functions taking a `FILE*` throw a `Failed` exception in an error occurs e.g. the data file could not be parsed. The functions taking a filename, however, return an error code which is non-zero if the file open/read/write failed, zero if successful. The return value should be checked to ensure that the operatin was successful (see Appendix C). The possible flags are

`mxflag::binary` create binary output rather than ASCII numbers. This is much quicker to read and write and is more compact, but the resulting file is not very portable since the way binary floating numbers are stored can vary from machine to machine.

`mxflag::doublep` use double precision numbers. Storing as single precision numbers is more compact but loses accuracy.

`mxflag::simple` forces (ASCII) output to be a simple list of numbers (real/complex alternating pairs for complex numbers), without any header describing the size of the matrix.

`mxflag::block` forces (ASCII) matrices to be formatted one data row per line. The resulting output can be easily recognised as a matrix, and some programs can use the formatting to determine the matrix dimensions e.g. *Matlab*. The `block` option is not suitable, however, for large matrices, since the resulting long data lines can cause input buffers to overflow.

`mxflag::norowsep` suppress the insertion of newlines used to delimit rows in ASCII output. Combined with `simple`, this generates the simplest possible output, but is rarely needed since most input routines happily ignore the extra white space.

Flags are combined using `|` i.e. the default setting of `mxflag::doublep | mxflag::binary` means that data will be written in double precision binary. Note that the formatting flags, `simple`, `block` and `norowsep` only apply to ASCII output.

The following functions used to read and write *Matlab* files are declared in `"matlabio.h"`, which must be `included` if *Matlab* files are needed. Note that *Matlab* can also read and write simple ASCII files; these can be created by `write_matrix` using the `mxflag::block | mxflag::simple` options. *Matlab* files always have a *".mat"* extension and the `WriteMATLAB` functions always add it to the supplied filename. *Matlab* files can contain more than one matrix, and each matrix must be named. If writing a single matrix to a file, it generally makes sense to use the same name for both the matrix and the file. This name is used when the matrix is loaded into *Matlab*, and so must be a valid *Matlab* variable name; using a name such as *"my.matrix"* will cause an error when the file is loaded into *Matlab* since the "." will be interpreted as dot product.

The objects that can be written to and read from *Matlab* files are real or complex `List`, `Matrix` or `MultiMatrix` objects. When reading, the dimensionality of the destination object must match the dimensionality of the data in the file otherwise a `Failed` exception is thrown. Real data files can be read into complex objects but not vice versa.

`WriteMATLAB(FILE*` *filep*`,X` *object*`,const char*` *name*`,`*version* `=0)` writes a *Matlab* file to the given data stream. Multiple objects can be stored in the same file by sequential calls to `WriteMATLAB`; clearly nothing should be written to the stream between calls. The format used by the data object is determined by the optional *version* indicator which can be 4, 5 or 0. If zero, the "default" format is used, which is V4 except for multidimensional matrices which can only be stored in version 5 files. *It obviously does not make sense to mix V4 and V5 formats in the same file!* An exception is thrown if the write fails.

`int WriteMATLAB(const char* fname,X object,const char* =NULL), version =0` writes a (single) matrix to a file. If the name of the matrix is not supplied, it is taken from the filename. An error code is return if the write was unsuccessful e.g. the file could not be opened.

`int ReadMATLAB(X& obj,FILE* filep)` read a object from an open *Matlab* file. The *Matlab* matrix name is not retrieved. An error code is returned if the read was unsuccessful e.g. the data file was corrupt, or the end of file had been reached.

`int ReadMATLAB(X& obj,const char* fname)` initialises *matrix* from a *Matlab* file. An error code is returned if the read or file open was unsuccessful.

The *Matlab* I/O routines do not use the *Matlab* library functions, which is useful for portability but does mean that they may not cope with all possible formats, especially of the V5 format files. The (undocumented) `matlab_controller` object can be used for more sophisticated control of *Matlab* I/O e.g. looking ahead to identify the nature of the next object in the file.

The header file `simpsonio.h`declares some functions for reading and writing files in `SIMPSON` format (ASCII format only):

`int read_simpson(cmatrix& dest, char* fname)` reads a `SIMPSON` file into a complex matrix (created as a row vector if the data is one-dimensional). A non-zero error code is returned if the read was unsuccessful.

`int read_simpson(cmatrix& dest, simpsonFD& info, char* fname)` reads the file into *dest* and stores the information for the header in the supplied SIMPSON "file descriptor" object (see the header for details of what this contains).

`int read_simpson(List<complex>& dest,simpsonFD& info, char* fname` reads the file into a flat complex vector.

`write_simpson(char* fname,List<complex> data,sw,isspectrum)` writes a complex vector to a `SIMPSON` file. The spectral width must be supplied. The *isspectrum* flag (default `true`) flags the data is a spectrum rather than an FID. An exception is thrown if the write was unsuccessful.

`write_simpson(char* fname,cmatrix data,sw,sw1,isspectrum)` writes a two-dimensional matrix to a `SIMPSON` file. *sw* and *sw1* are the spectral widths in the direct and indirect dimensions respectively.

`write_simpson_rows(char* fname,cmatrix data,sw,isspectrum)` stores a 2D data set as a series files for the individual rows which will be named *fname*00, *fname*01 etc. These can then be displayed in `simplot` (which doesn't understand 2D `SIMPSON` files). The matrix cannot have more than 100 rows.

Operation	in-place	new-object	supplied-destination
`add`	TX	TX	X
`subtract`	TX	TX	X
`minus`	X	X	X
`multiply`	TX	TX	X
`divide`	TX	TX	X

Table 5: Standard operations defined for the `List<T>` template type. `T` denotes a scalar operand e.g. `*=2`.

4.9 Real and diagonal matrices

The `rmatrix` class for real matrices is declared in `rmatrix.h` and is essentially identical to its complex counterpart `cmatrix`, but with a slightly more limited number of numerical operations. Real matrices should be used when possible since they take up half the memory of a complex matrix and many operations, such as diagonalisation, are *much* more efficient:

`hermitian_eigensystem(rmatrix&` *vectors*`,List<double>&` *values*`,rmatrix` *A*`)` returns the eigenvalues and eigenvectors for a real symmetric matrix[13].

Diagonal matrices can be handled much more efficiently than full matrices and so it is useful to define functions that make use of them. There is no diagonal matrix type as such (unlike GAMMA), but `List<double>` and `List<complex>` can be used for real and complex diagonal matrices respectively.

It is important to realise that the `List` objects are not really diagonal matrices nor row or column vectors. It is the function being applied that determines their "shape". Addition of a matrix and a `List` is unambiguous, for example; the `List` must represent a diagonal matrix. This is not the case, however, for multiplication. Hence only supplied-destination forms of multiplication are provided, since the output type determines the interpretation of the `List`.

The following functions convert between diagonal and full representations:

`List<T> diag(Matrix<T>` *A*`)` returns the diagonal elements of A. The SD form, `diag(List<T>` *d*`,`*A*`,`*n* `=0)`, puts diagonal n of A in d.

`Matrix<T> full(List<T>` *vector*`)` places *vector* along the diagonal of an otherwise zero matrix. The SD form is useful for creating a *complex* matrix from a *real* vector.

5 Other storage types

5.1 The List<T> class

There are two forms of list object; BaseList<T> and List<T>[14] (which is derived from BaseList<T>). List<T>, like Matrix<T> (and its derived classes, rmatrix and cmatrix) is dynamically created i.e. memory is allocated to store its members. In contrast, the BaseList<T> class is a convenient package for a pointer to an already-allocated list of objects (plus their number). Creating a BaseList from a pointer does not involve copying the objects (unlike List). When allocating space from an undefined List, exactly amount of memory required is requested, but subsequent create operations do not necessarily release freed memory in order to minimise expensive calls to memory allocation functions. Using BaseList, responsibility for memory allocation is entirely with the user. They are useful when interfacing libcmatrix functions to other objects/code, but they should not generally be used for storage. In particular, using non-dynamic objects inside objects is not recommended. Unless a suitable copy operator is defined, for example, copying a BaseList creates an object referring to the same place as the original—a recipe for disaster!

Because List is derived from BaseList, functions that expect a BaseList input will also accept a List (but not vice versa); the only exception being when the size of the input list needs to be changed cf. "supplied-destination" functions for the matrix classes. The equivalent of new-object functions necessarily return List<T> objects. Most supplied-destination functions have two forms; one for a BaseList destination, the other for a List. A Mismatch exception will be thrown in the former case if the BaseList is the wrong size for the list being created.

The following constructors are provided:

(Base)List() creates an empty list object. Its size() is defined to be zero.

(Base)List(int n,T* *addr*) creates a list of n objects from a pointer to n objects of type T. An important distinction is that the objects are only copied when creating a List<T>. Also exists as a create function.

List(int n) creates a list of n objects created by the default constructor for T (which must exist). For types such as complex this means the contents are undefined. Also exists as a create function.

List(int n,T v) creates a list of n copies of object v.

List<T> also has a clear() function which releases any memory allocated and returns the list to an empty state.

Note that copying one BaseList to another is only possible if the lists have the same size. Parentheses are used to obtain list elements and the access checking is again controlled by the setting of NDEBUG/BOUNDS_CHECK, Section 4.3. Sublists are returned using

[13] This used to be symmetric_eigensystem, but has been renamed so that the functions names of rmatrix and cmatrix coincide whereever possible. Since symmetric matrices are by definition hermitian, this is still mathematically correct!

[14] In fact, there is a third type, DynamicList<T> which behaves like List<T> but whose capacity cannot be dynamically changed. It is used internally as the base storage type for matrices etc., but List<T> is more flexible for normal use.

(BaseList<size_t>). Note that *initialising* a BaseList from another List creates an object that points to the *same data*, while assigning (using =) one list to another makes a new copy of the data. This distinction is confused by code such as BaseList<T> A=B which is an *initialisation* and is actually equivalent to BaseList<T> A(B) i.e. A now points to the same data as B.

Other functions:

size() returns the length of A. This is zero for an empty list.

vector() returns the pointer to the data, which is useful for passing lists to non-libcmatrix functions, but should otherwise be used with care.

truncate(size_t n) returns a new BaseList with a number of elements set to n. This is useful for working with an over-allocated array. Obviously n must not exceed the number of items originally allocated!

resize(n) is defined to change the length of a list while preserving contents (unlike create). The list is truncated if n is less than the current list size.

reserve(n) ensures that the object will hold at least n objects. It behaves like resize if n exceeds than the current memory allocation, otherwise it does nothing.

sum(*list*) returns the sum of the list. This requires the operations =0 and += to be defined for the data type.

max(*list*) and min(*list*) returns the maximum and minimum values in *list*. These operations obviously require the comparison operators to be defined for T and require the list to be non-empty (otherwise the ListEmpty() exception is thrown).

== returns true if two Lists are identical. Two lists are identical if all the corresponding elements are identical (Lists are ordered). Clearly lists that differ in length are never identical, and the == operation must be defined for T for this function to work.

The mathematical functions defined on lists are listed in Table 5 (a). If two input list arguments are used, they must have the same size otherwise a Mismatch exception is thrown. Multiplication and division are both element-wise operations. this is not the same as *Matlab* usage. Unlike vectors in *Matlab*, a List<T> has no direction, and may be treated as a row, column vector, or even diagonal matrix depending on context. The cmatrix and rmatrix types must be used if vectors with real shape are essential, though this is less efficient than the corresponding List<T>.

Other operations:

Modified

sort_ip(*list*)and sort_ip(*list*,*compare*) does an in-place sort with or without a supplied comparison function (ascending order if none supplied) using the sort function from the Standard Library.

sort(*list*)and sort(*list*,*compare*) returns *list* as a sorted new List object.

The dynamic allocation of memory for `List<T>` can be a drawback if a function uses temporary lists. This problem can often be avoided using the `ScratchList<T,`*size*`>` template type declared in `ScratchList.h`. A `ScratchList` contains an array `T[size]` (*size* is fixed at compile time) which is claimed from the stack like other automatic variables rather than free store and is used if the `ScratchList` has less than *size* items. If not, the memory will be allocated from free store. `ScratchList` is thus efficient for small vectors (less than *size*) but will still work with larger lists for which the delays in memory allocation are less important. The preprocessor symbol `SCRATCH_SIZE` contains a "suitable" value for *size* which is set to 96 unless already defined. `SCRATCH_SIZE` is used whenever scratch vectors are required inside `libcmatrix`. From the point of view of functions, a `ScratchList` behaves as a `BaseList<T>`.

`ScratchList` is also useful for constructing lists at compile time from an explicit list of elements. Selecting elements 1 and 2 of a list `A` requires clumsy syntax of the form

New!

```
size_t rawinds[]=1,2;
cout << A(BaseList<size_t>(2,rawinds));
```

Note that `rawinds` is not `const` since creating lists (or matrices) of `const` objects is problematic (it is simpler to declare the container object itself as `const`).

This awkward two step initialisation can be replaced by `A(ScratchList<size_t>(1,2))`.

The number of elements in such a `ScratchList` initialisation is limited to between 2 and 10 (inclusive). `ExplicitList` can be used for an arbitrary number of explicit elements, but is limited to "simple" objects e.g. `ExplicitList<2,size_t>(1,2)`. Note that the size of the list must be passed as a template parameter so that the constructor knows how many values to read.

5.2 The `MultiMatrix<T,N>` class

`MultiMatrix` is a functional multi-dimensional matrix type which permits the storage of multi-dimensional data and core mathematical operations. The dimensionality is currently restricted to 4; this will be extended but does involve a lot of typing... It is used internally in some parts of the library, but not in any visible interfaces.

As with the `Matrix` type, the data is stored in a "flat" memory allocation and the overheads are relatively small, though greater than for `Matrix`. It is efficient, wherever possible, to "reduce" the multi-dimensional object to a simple matrix slice or vector rather than working with the full multidimensional object. It is important to note that the dimensionality is fixed at compile time; a `MultiMatrix` cannot change its dimensionality. This means that `MultiMatrix` fits naturally into the compile-time algebra[15].

`MultiMatrix()` creates an undefined matrix (dimensions are defined to be zero).

`MultiMatrix(A)` creates a new matrix from an object of matching dimensionality, copying the structure and contents from `A`. A special case is made for initialising a `MultiMatrix<T,3>` from `BaseList< Matrix<T> >` (which would otherwise appear to involve initialising a 3D object from a 1D one).

[15] The `MultiHolder<T>` type provides a somewhat basic object which can switch its dimensionality.

`MultiMatrix(`n`,`m`,...)` creates an empty matrix with the supplied dimensions, $m \times n \times \ldots$. The contents are undefined.

`MultiMatrix(`n`,`m`,...,T` v`)` creates a $m \times n \times \ldots$ matrix with contents initialised to v.

`MultiMatrix(`n`,`m`,...,T*` *data*`)` creates a $m \times n \times \ldots$ matrix with data address *data* (data must be stored with last index varying most rapidly).

`create(...)` versions of these last three constructors are also defined, allowing a previously initialised object to be (re)created.

`size()` return the number of data items.

`dimension(`n`)` returns the size of dimension n $(0 \ldots N-1)$ where dimension 0 is the first index (and varies most slowly). `rows()` and `cols()` can also be used for 2D objects.

`clear()` releases the memory allocation and returns the matrix to the undefined state.

`row()` returns the data array as a one-dimensional vector (`BaseList<T>`).

`back()` returns the last element, row, matrix slice etc. of 1, 2, 3 etc. dimensional objects. `A.back()` is effectively shorthand for `A(A.dimension(0)-1)`.

`front()` returns the first element, row, matrix slice etc. of 1, 2, 3 etc. dimensional objects. `A.top()` is equivalent to, but marginally more efficient than, `A(0)`.

The `()` operator returns individual elements and subsets of a `MultiMatrix`. If the arguments are N integers, a single element is returned e.g. `A(1,2,3)` returns slice 1, row 2 and column 3 etc. (if `A` is a 3D object). If fewer arguments are used, then the dimensionality of the object returned increases accordingly e.g. `A(1,2)` returns row 2 of slice 1. The types of the objects returned are: `T&` (0), `BaseList<T>` (1), `Matrix<T>` (2), `MultiMatrix<T,M>` (M). Note that references can only be returned for element access so it may be necessary to store the result in a temporary before it is passed as an output argument to other functions[16].

Arguments that cannot be treated as simple indices are taken as index ranges and the object returned is an `IndirectMultiMatrix<T,M>` where `M` is the number of index ranges. Hence `A(1,range(),slice(2,3))` returns a `IndirectMultiMatrix<T,2>` which refers to columns 2–4 of slice 1 (the empty `range` being equivalent to selecting the complete row range). `IndirectMultiMatrix` is extremely flexible, but this comes at the price of signficant overhead. In this case, for example it would be signficantly more efficient to first select slice 1 and then select the submatrix i.e. `cmatrix Aslice(A(1)); Aslice(range(),slice(2,3))` (this would be a `IndirectMatrix<T,range,slice>`).

[16] This is quite a subtle point... There is no difficulty in directly passing a temporary that refers to an original `const` object by reference to a function (e.g. `const BaseList<T>&`). This is not generally true for non-`const` objects. An exception can be made, with care, when passing simple vectors. Here input arguments can be passed using `BaseList<T>` as the overhead of creating a new `BaseList` is negligible. The new `BaseList` still refers back to the original data. In this case, however, references should not be made to the passed object as this will be out of scope when the function exits.

5.3 The `ListList<T>` data type

Using `List< List<T> >` is a very inefficient way to store a list of lists since separate memory allocations are made for each list. The `ListList<T>` type is an efficient way to implement a list of lists. In `ListList<T>`, the data entries are stored as a single `List<T>` together with an index allowing the individual sublists to be accessed.

These objects can be created in a variety of ways

Modified

`ListList<T>(BaseList<size_t>` *sizes*`,BaseList<T>` *list*`)` where the size of each block are given in the *sizes* vector while the members are listed in *list*. Effectively *list* is partitioned in blocks given by *sizes*. Omitting *list* creates the structure of the `ListList<T>`, without filling it.

`ListList<T>(BaseList<T>` *list*`,ListList<size_t>` *indices*`)` where the structure of the created object is given by that of *indices*, which is used to specify the elements of *list* that should be used to fill it. Hence `ListList(H,find_blocks(H,1e-10))` would find the block structure of the (diagonal) H matrix and use this to construct the blocked version of H.

`ListList<T>(BaseList< List<T> >` *list_of_lists*`)` creates a `ListList<T>` from an actual list of lists.

Elements are accessed with the `()` operator. *listlist*`(i,j)` is the j element of the i list (always numbered from 0). *listlist*`(i)` returns the i list (as a `BaseList<T>`). The elements can be manipulated through these operators. As usual, checking of the indices is disabled if the program is compiled with `NDEBUG`.

Other functions that return information on a `ListList<T>`:

`items()` returns the number of items in the object (the sum of the block sizes).

`size(`i`)` returns the number of elements in block i. Note that this is allowed to be zero.

`blocks() or size()` returns the number of blocks (lists) in the object. The presence of `size()` means that a `ListList<T>` can be treated as list of `BaseList<T>` objects by suitably generic algorithms.

`structure()` returns a `List<size_t>` object which can be used to construct a new `ListList<T>` with the same structure.

5.4 Blocked matrices

The `BlockedMatrix<T>` type is used to store Hamiltonians, propagators etc. in block diagonal problems. It is not described in detail as these objects are not generally used independently. Although `BlockedMatrix<T>` functions as a `BaseList<Matrix<T>>` in use, there are important differences: the dimensions of the different matrices are fixed when the object is created (and cannot be changed independently). This allows a single block of memory to be allocated, which reduces the overhead associated with dealing with many small matrices (especially 1×1). Purely diagonal blocked matrices should be stored in a `ListList<T>`.

New!

A `ListList<size_t>` is used to describe the block structure of a matrix (see below for a description of the `ListList` type). It is composed of a list of blocks, specified by a list of row/column indices, e.g. the list $\{0, 3, 4\}$ corresponds to a block consisting of rows/columns 0, 3 and 4 of the original matrix. Assuming that the original matrix spanned the full Hilbert space, these `List<size_t>` vectors can then be used to construct blocks of spin operators for spin-1/2 systems using `spinhalf_system`.

`ListList<size_t> find_blocks(`A`,double` *tol* `=0)` finds the blocks of a matrix A, taking any element that is larger than the given tolerance to be non-negligible. The tolerance will need to be greater than zero if there is any "noise" on the matrix elements.

`ListList<size_t> find_blocks(List<double>` *evalues*`,`*tolerance*`)` constructs a blocking list assuming that diagonal elements differing by at least *tolerance* correspond to different blocks of the Hamiltonian. Note that `find_blocks(diag(`A`))` effectively returns the block strcture using the (secular) approximation that the off-diagonal elements can be neglected.

6 NMR functions

6.1 Using spin systems

The `spin_system` class (from `spin_system.h`) is implemented as a `List` of a fundamental `spin` type which is specified by an "isotope name" e.g. "13C" Unlike GAMMA, no other nuclear properties are stored in the `spin_system`; its job is as an input to functions that create spin operators, rather than complete Hamiltonians.

The member functions of the `spin` type are

`double gamma()` returns the gyromagnetic ratio. Note that the gyromagnetic ratio of a known isotope can be found independently using, for example, `gamma("13C")`.

`char* isotope()` returns the isotope name e.g. "13C".

`isotope(char*` *nucleus*`)` sets the isotope from an isotope name.

`float qn()` returns the I quantum number.

`size_t nucleus()` returns an internal number identifying the type of the nucleus.

`size_t deg()` returns the number of spin states, $2I + 1$.

Comparison of `spin` objects returns `true` if they are of the same type.

Spin systems are created using

`spin_system(`n`,char*` *nucleus*`)` creates a system of n spins of type *nucleus*. *nucleus* must be one of the known nucleus types, otherwise an `InvalidParameter` exception is thrown.

The number of spins is fixed when the `spin_system` is created, but the identity of the spins can be altered using e.g. `sys(0).isotope("1H")`, or using:

`isotope(`m`,`n`,char*` *nucleus*`)` set spins m to n to given nucleus type.

Note that the isotope type is used by higher level functions to identify groups of nuclei e.g. pulses can be applied to the ^{13}C spins etc.

`()` is used to address individual spins e.g. `sys(1).isotope("13C")` sets the type of spin 1 to ^{13}C.

Other functions:

`size_t` *sys*`.nspins()` returns the number of spins in a spin system.

`size_t` *sys*`.size()` returns the dimensionality of the Hilbert space for the full spin system i.e. the product of `deg()` for all the spins. N.B. The spin system *may* have been set up to work on a sub-space of this full space, so don't assume that the spin operators it returns are square matrices of this size. Use the functions below.

`size_t` *sys*`.rows()` and *sys*`.cols()` returns the number of bra and ket states respectively.

`bool isdiagonal()` returns `true` if the bra and ket states are identical (always true for simple `spin_system` objects.

6.1.1 `spinhalf_system`

`spin_system` is actually a subclass of a `basespin_system` type. Another such subclass is `spinhalf_system` which creates spin operators for systems of spin-1/2 nuclei only. These can be created extremely efficiently by "bit twiddling" rather than using direct products of single spin operators. Another major advantage is that it can generate general submatrices of spin operators. Higher level functions are defined (whenever possible) in terms of `basespin_system` so that they will function with any class derived from it. Individual spin states are stored in `state_t` type, which is defined as `unsigned long` by default[17]. Note that on most systems this is different from the `size_t` type used to store indices e.g. into the Hilbert space.

`spinhalf_system` is used in the same way as `spin_system`, with the exception that only spin-1/2 nuclei nuclei are permitted. It is also possible to restrict the size of Hilbert used

`spinhalf_system(int` N`, const char*` *nucleus* `="1H")` specifies N spin-1/2 nuclei of the given type with full Hilbert space i.e. 2^N states.

`spinhalf_system(int` N`, const char*` *nucleus*`, BaseList<state_t>` *states*`)` specifies a sub-space of a N spin-1/2 system where the (symmetrical) sub-space of bra and ket states is given as a list of spin states (0 to $2^N - 1$). Note that the "blocking" functions address the Hilbert space in terms of offsets defined as `size_t`. These need to be converted to `state_t` in order to initialise a `spinhalf_system` i.e. `List<state_t>(size_t` *list*`)`.

`spinhalf_system(int` N`, const char*` *nucleus*`, BaseList<state_t>` *bra-states*`, BaseList<state_t>` *ke*
specificies a general sub-space of the 2^N Hilbert space.

[17] Depending on the system, this limits the number of spins that be considered to 32 or 64. The type used for `state_t` can be changed to increase this limit, but requires re-compilation of the library.

`rows()` and `cols()` return the number of bra and ket states (or rows and columns in the output operators) respectively.

`brastates() and ketstates()` returns a `BaseList<state_t>` containing the bra or ket states respectively.

`isdiagonal()` returns `false` if distinct bra and ket states were used to define the Hilbert space.

6.2 Creating spin operators

The next step in a simulation is to create the spin operators needed. These functions can be quite slow, so all the spin operators required are usually created at the start of a simulation. The input to the spin operator functions is some form of spin system and the output is a `cmatrix`, with the exception of functions that return `List<double>` for diagonal i.e. z spin operators. Note that spin operators for large spin systems are always highly sparse. It is often more efficient for multi-spin problem to create only submatrices of spin operators as they are needed.

6.2.1 Product operators

Simple spin operators and product operators are created using `I` and `F` functions (sum operators) using the `spin_system` type.

`cmatrix I(`*sys*`,`*n*`,`*op*`)` returns an operator for spin n of the specified spin system. *op* must be one of `'x'`, `'y'`, `'z'`, `'+'`, `'-'`, `'a'`, `'b'` or `'I'` (identity operator). The α, β polarization operators are only valid for spin-1/2 spins.

`List<double> diag_Iz(`*sys*`,`*n*`)` returns a z operator as a diagonal matrix. This is equivalent to, but substantially more efficient than `real(diag(I(`*sys*`,`*n*`,'z')))`.

`cmatrix I(`*sys*`,char*` *ops*`)` can be used to construct product operators. *ops* is a string that list that gives the spin operator for each spin in the system. The length of the string must equal the number of spins in the spin system. For example, `I(sys,"xyz")` would return the operator $I_{0x}I_{1y}I_{2z}$. It is somewhat more efficient than taking products of full spin operators e.g. `I(`*sys*`,0,'x')*I(`*sys*`,1,'x')`. Note that this function cannot create operators of the form $I_{1x}I_{1y}$ (see below).

`cmatrix I(`*sys*`,`*m*`,char` *op_m*`,`*n*`,char` *op_n*`)` is an interface to the routine above for bilinear operators.

`cmatrix I(`*sys*`,`*m*`,char*` *ops*`)` constructs a product operator for a *single* spin i.e. `I(sys,1,"xy")` will return $I_{1x}I_{1y}$.

`List<double> diag_Iz(`*sys*`,`*m*`,`*n*`)` returns the product operator $I_{mz}I_{nz}$ as a real diagonal matrix.

`cmatrix F(`*sys*`,`*op*`)` returns a sum spin operator for the complete spin system, F_x etc.

`List<double> diag_Fz(`*sys*`)` returns the F_z operator for the complete system as a real diagonal matrix.

`cmatrix F(`*sys*`,`*type*`,`*op*`)` returns a sum operator for a given nucleus type e.g. `F(sys,"13C",'z')`. The type can be specified as a name, `"13C"` or the internal nucleus number.

`List<double> diag_Fz(`*sys*`,`*type*`)` returns the F_z operator for a group of spins.

The `coherencematrix` functions return a `Matrix<bool>` matching the Hilbert space in which selected coherences are marked by `true` values.

`Matrix<bool> coherencematrix(`*spin_sys*`,`*coher(s)*`)` determines the F_z operator for the given spin system and uses this to calculate the coherence mask for the given coherence (specified as a single integer) or list of coherences (`BaseList<int>`) e.g. `coherencematrix(sys,ExplicitList<` would return a matrix in which the ± 1 coherences were flagged.

`Matrix<bool> coherencematrix(`*spin_sys*`,`*spins*`,`*coher(s)*`)` calculates F_z for the selected group of spins (by nucleus name).

`Matrix<bool> coherencematrix(List<double>` F_z`,BaseList<int>` *cohers*`)` is allows a predetermined F_z operator to be supplied.

Masks involving mixed heteronuclear coherences e.g. $+1$ on `"1H"`, -1 on `"1H"` can be obtained by combining masks with the logical operations, `|`, `&` etc.

The following primitive functions are also defined

`cmatrix kronecker(`A`,`B`)` returns the direct product $A \otimes B$, where A and B can `cmatrix` or integers. An integer represents an identity matrix of the corresponding size. If either A and B is undefined, the result is B or A respectively. These functions are also defined for real matrices.

`cmatrix spinhalfop(char` *op*`)` returns the simple spin-1/2 operators (pre-defined for speed). *op* must be one of `'x'`, `'y'`, `'z'`, `'a'`, `'b'`, `'+'`, `'-'` or `'I'`.

`void spinop(cmatrix &`*dest*`,int` m`,char` *op*`,double` *scale* `=1.0)` is a primitive function returning a spin operator for a spin with m nuclear states, i.e. $m = 2I + 1$. The matrix can be optionally scaled.

`void expandop(cmatrix &`*dest*`,`*sys*`,`m`,`*matrix*`)` returns the spin operator for spin m, *matrix*, "expanded up" to the full Hilbert space of a spin system. An exception is thrown if the size of *matrix* does not match the number of energy levels of the spin.

6.2.2 Tensor operators

A limited number of tensor operators are declared in `tensorop.h`. They are simply rank 1 and rank 2 tensor operators constructed from combinations of product operators e.g. `sqrt(6)*T2(sys,0,1,2,0)` gives the same matrix as

```
2*I(sys,0,'z',1,'z')-I(sys,0,'x',1,'x')-I(sys,0,'y',1,'y')
```

In other words, the basis set of the matrices is still the product basis $\langle\alpha\alpha\rangle$ etc., rather than a tensor operator basis. `libcmatrix` is not tied to any particular basis, but the functions that return spin operators invariably give results in this conventional product basis.

`cmatrix T1(`*sys*`,`*i*`,`*m*`)` returns the single spin tensor operators $T_{1,m}$

$$T_0 = I_z \qquad T_\pm = \mp\sqrt{2}I_\pm \tag{3}$$

`cmatrix T2(`*sys*`,`*i*`,`*j*`,`*l*`,`*m*`)` returns the rank 2 spin tensor operators spanning the two spins i and j of the given spin system; $l = 0, 1, 2$ and $m = -l \cdots l$. Note that i and j can be equal e.g. for quadrupolar spins.

$$T_{0,0} = -\sqrt{\frac{1}{3}}(I_{ix}I_{jx} + I_{iy}I_{jy} + I_{iz}I_{jz}) \tag{4}$$

$$T_{1,0} = -\frac{1}{2\sqrt{2}}(I_{i+}I_{j-} - I_{i-}I_{j+}) \qquad T_{1,\pm1} = \frac{1}{2}(I_{iz}I_{j\pm} - I_{i\pm}I_{jz}) \tag{5}$$

$$T_{2,0} = \sqrt{\frac{1}{6}}(2I_{iz}I_{jz} - I_{ix}I_{jx} - I_{iy}I_{jy}) \tag{6}$$

$$T_{2,\pm1} = \mp\frac{1}{2}(I_{i\pm}I_{jz} + I_{iz}I_{j\pm}) \qquad T_{2,\pm2} = \frac{1}{2}I_{i\pm}I_{j\pm} \tag{7}$$

6.3 Spin state permutations

When dealing with problems of chemical exchange and symmetry, it is often necessary to deal with spin permutations, that is changing the labelling of spins as a result of an exchange process or symmetry operation. A permutation is described using the `Permutation` class which holds the new index of the spin after the "exchange" e.g. {`0, 2, 1`} would correspond to leaving spin 0 unchanged and interchanging spins 1 and 2. Similarly {`1, 2, 0`} would be a cyclic permutation of the spins.

`Permutation(const BaseList<size_t>&)` initialises a permutation from a list of spin indices.

`Permutation(const char*)` allows permutations for small (<10 spin) systems to be initialised from the permutation expressed as a string e.g. `Q''021"`.

`order()` returns the permutation order i.e. the number of times the permutation can be repeated before the original sequence is returned.

`apply(List<T>&` A'`, BaseList<T>` A`)` applies the permutation to a vector of quantities A (which must be of the same length as the permutation vector).

`apply(Matrix<T>&` A'`, Matrix<T>` A`)` applies the permutation to a both the column and row indices of a matrix, A.

The permutation is checked for validity when created; each index should be usd exactly once and the permutation order must be well defined e.g. (21)(453) would not be valid since the

two subcycles have different orders (2 and 3 respectively). The `Permutation` type can be manipulated as a `List<size_t>` although elements should not be written to in case this breaks the preconditions.

The following functions calculate how the permutation of spins affects the spin states

`state_t` *sys*`.permute(state_t` *state*`,Permutation` *permvec*`)` returns the state generated by applying the permutation specified by *permvec* to *state* (which does not actually need to be within the Hilbert space of *sys*!).

sys`.permutation_vectorH(List<state_t>&` *newstates*`,Permutation` *permvec*`)` creates the list where *newstates(i)* is the *index* of the state within the Hilbert space (NOT the state itself!) resulting from applying `permute` to all the states of the Hilbert (sub)space of *sys*. A `Failed` exception is thrown is any of the states created fall outside the subspace specified by *sys*.

sys`.permutation_vectorL(List<state_t>&` *newstates*`,Permutation` *permvec*`)` does the same but for the Liouville states (of which there are n^2 for a Hilbert space of size n).

Thse functions can only be applied to spin systems with "diagonal" Hilbert subspaces. They are very efficient for `spinhalf_system`, considerably less so for `spin_system`.

The list of permuted states contains all the necessary information on the permutation, but the following convenience functions generate matrices, T, which can be applied as unitary transforms to matrices in the Hilbert or Liouville space:

`rmatrix` *sys*`.permutation_matrixH(`*permvec*`)` returns a matrix T such that THT' applies the spin permutation to a matrix H in the Hilbert space.

`rmatrix` *sys*`.permutation_matrixL(`*permvec*`)` returns a matrix T such that $T\hat{\hat{H}}$ performs the permutation on the superoperator H.

6.4 NMR functions

The functions declared in `NMR.h` are specific to NMR simulations. It is possible to write simulations using only the data types and functions described above, and `NMR.h` is an optional higher-level package of functions.

The first group of functions are used to construct the separate space and spin parts that characterise Hamiltonians in solid-state NMR. The Hamiltonians are all in their high-field forms and so many of the Hamiltonians are diagonal (with the z axis oriented conventionally along B_0). Corresponding `diag_` functions provide the real diagonal Hamiltonians directly and more efficiently.

`double dipolar_coupling(double` γ_1`,double` γ_2`,double` r`)` returns the dipolar coupling (in Hz) between two spins of specified gyromagnetic ratio and internuclear distance r. SI units are used for consistency, i.e. r is in metres.

`space_T spatial_tensor(double` δ`,double` η `=0)` returns the pure rank 2 tensor for an interaction of strength δ (e.g. as returned by `dipolar_coupling`) and anisotropy η. It is

important to note that there is more than one convention for the normalisation of irreducible spin tensors. Using `spatial_tensor` together with, say, `spin_dipolar` is always "safe", even if the normalisation convention is changed. *N.B.* `spatial_tensor` *is only appropriate when working within the high-field approximation*

`space_T spatial_tensor(double` *iso*`,double` δ`,double` η`)` as above, but including the isotropic value corresponding element (0,0) of the resulting tensor. Note that it is often more convenient to treat this isotropic component separately.

`cmatrix spin_dipolar(`*sys*`,`i`,`j`)` returns the spin operator for the dipolar coupling between spins i and j. If i and j have different nucleus types, the coupling is truncated to its heteronuclear form i.e. $2I_{iz}I_{jz}$.

`cmatrix spin_J(`*sys*`,`i`,`j`)` returns the Hamiltonian of the scalar coupling ($\boldsymbol{I}_i \cdot \boldsymbol{I}_j$) between spins i and j. Note that the weak coupling Hamiltonian is given by `I(`*sys*`,`i`,'z',`j`,'z')` or in its diagonal form by `diag_Iz(`*sys*`,`i`,`j`)`.

`cmatrix spin_quadrupolar(`*sys*`,`i`)` returns the spin operator for the quadrupolar coupling of spin i to first order. A `Failed` exception is thrown if i is not a quadrupolar nucleus.

$$(3I_z^2 - I(I+1)\boldsymbol{I})/3 \tag{8}$$

`List<double> diag_spin_quadrupolar(`*sys*`,`i`)` returns the quadrupolar Hamiltonian for spin i as a real "diagonal matrix".

Once the Hamiltonian has been constructed, propagators can be calculated. The propagator for time period t is given by $U(t,0) = \exp(-2\pi i H t)$ where H is independent of time. Note that H must be expressed in Hz (assuming t is in seconds) and *not* angular units.

`cmatrix propagator(cmatrix` H`,double` t`)` returns the propagator for Hamiltonian H (expressed in Hz) applied for a given time (in seconds). H must be Hermitian since `hermitian_eigensystem` is used for diagonalisation. Other forms: SD.

`List<complex> propagator(List<double>` H`,`t`)` calculates the propagator for a diagonal Hamiltonian. The result is complex diagonal. Other forms: SD.

`complex propagator(complex` a`,double` t`)` returns the dynamic phase factor for a 1×1 Hamiltonian (a) applied for a given time. This is trivially $\exp(-2\pi i a t)$, but is included for completeness.

In the high field approximation, the system Hamiltonian commutes with the sum magnetisation operators, F_z. Under these circumstances, the Hamiltonians (and propagators) for different phases of the RF can be quickly evaluated from the propagator/Hamiltonian with a given phase:

$$U(\phi) = e^{-i\phi F_z} U(0) e^{i\phi F_z} \tag{9}$$

Since the Hamiltonian is purely real for x phase RF, $U(0)$ can be evaluated efficiently. Although Eqn. (9) is easy to evaluate (since F_z is diagonal), it is efficient to store the $\exp i\phi F_z$ factors if the same rotation is used frequently. The relevant functions are

`cmatrix rotatez(cmatrix` U`, List<double>` F_z`,` ϕ`)` performs a "z-rotation" through angle ϕ on U. F_z is for the nucleus (nuclei) involved.

`rotatez_ip(cmatrix&` U`, List<double>` F_z`,` ϕ`)` performs the rotation "in-place"

`rotatez_ip(cmatrix&` U`, List<complex>` *facs*`)` performs the in-place rotation given a previously-evaluated $\exp(i\phi F_z)$.

`rotatezfacs(List<complex>&` *facs*`, List<double>` F_z`,` ϕ`)` calculates $\exp(i\phi F_z)$ given F_z and ϕ. The function is trivial but helps to ensure sign conventions are consistent.

`rotatezfacs(cmatrix&` *facs*`, List<double>` F_z`, List<double>` ϕ`)` calculates $\exp(i\phi F_z)$ for a set of rotation angles, storing the results in the rows of *facs*

The following functions diagonalise the propagator, $U(t,0)$ and convert the propagator eigenvalues, ω_i, into those of corresponding effective Hamiltonian:

$$U(t,0) = e^{-2\pi i H_{\text{eff}} t} \tag{10}$$

$$H_{\text{eff}} = V\Lambda V^\dagger \qquad \text{where } \lambda_i = \frac{\arg(\omega_i)}{-2\pi t} \tag{11}$$

`diag_propagator(cmatrix&` V`,List<double> &`*eigs*`,cmatrix` U`,`t`)`

`diag_propagator(List<double> &`*eigs*`,List<complex>` U`,`t`)` for a diagonal propagator

`double diag_propagator(complex` U`,`t`)` returns the effective frequency for a "single-state" propagator

It is important to note that the values of *eigs* are periodic with frequency $1/t$; by definition there are an infinite number of effective Hamiltonians which will give the same propagator.

As well as the propagators for evolution periods, it is also possible to calculate propagators for delta pulses:

`cmatrix Upulse(`*sys*`,`θ`,`ϕ`)` returns the propagator for an ideal pulse of a given tip angle, θ, and phase, ϕ, applied to all the spins. Phases can be expressed as numbers or as `"x"` (0°), `"y"` (90°), `"-x"` (180°) etc. It returns $\exp(i\theta H_{\text{RF}})$.

`cmatrix Upulse(`*sys*`,`*type*`,`ϕ`,`*phase*`)` returns the propagator for an ideal pulse of a given tip angle and phase applied to nucleus type specified as a string e.g. `"13C"`.

`cmatrix Upulse(cmatrix&` U`,rmatrix` Fx`,BaseList<double>` Fz`,`*tip*`,`*phase*`)` is an efficient version which uses pre-calculated F_x and F_z operators. Other forms: SD.

The `Upulse` functions are useful for occasional use, but are relatively inefficient when several propagators are needed for only slightly different pulses. The `PulseGenerator` object defined in `Sequence.h` provides an alternative, more efficient approach.

New!

`PulseGenerator(`*spin_sys*`,`*nuclei*`)` creates an object which generates pulses for a given set of nuclei (generally specified as a nucleus name e.g. `"13C"`) from a given spin system. *spin_sys* is any object that generates spin operators e.g. `spinhalf_system`. The pulses affect all nuclei if *nuclei* is omitted.

Pulses are returned from the `()` operator, which takes different arguments depending on whether the pulse is soft or hard:

`cmatrix` *obj*`(`*tip*`,`*phase*`)` returns a hard pulse of given tip angle and phase. Also SD.

`cmatrix` *obj*`(`*H*`,`*t*`,`*offres*`,`*tip*`,`*phase*`)` returns the propagator for a soft pulse of duration t (in s). The tip angle is that experienced by the on-resonance spins. *offres* is the resonance offset, $\Delta\nu$. H is the Hamiltonian of the system (without RF!) during the pulse (if undefined, it is taken as zero). The Hamiltonian during the pulse is

$$H_{\text{eff}} = H + \frac{\theta}{2\pi} H_{\text{RF}} - H_{\text{off}} \tag{12}$$

where $H_{\text{off}} = \Delta\nu F_z$ is the off-resonance term. The propagator is then $\exp(-2\pi i H_{\text{eff}} t)\exp(-2\pi i H_{\text{off}} t)$. Note how the propagator is corrected for the time spent in the frame rotating at the off-resonance frequency. Also SD.

6.5 Spatial tensors

Spatial tensors can be represented in Cartesian form using a normal 3×3 `cmatrix` with elements R_{xx}, R_{xy} etc. Tensors in terms of irreducible spherical tensors are represented using the `space_T` type which is defined in `space_T.h`.

A `space_T` has a maximum rank which sets the number of (complex) elements for which space is reserved. So a `space_T` of rank 2 has space for a single rank 0 component, 3 rank 1 components and 5 rank 2 components. It is not necessary, however, for all the ranks to be active, although it is most economical of memory if the maximum rank in use matches the maximum rank for which space is reserved. Attempting to access a component in an inactive rank is an error, and will generate a `BadRank` exception if access checking is enabled. Activating or deactivating a rank is rapid since the memory has already been allocated, but elements in previously inactive ranks *cannot* be assumed to be zero. On the other hand, it is not possible to add extra ranks without re-`create`-ing the `space_T`.

It is important to note that the components of (symmetric) spatial tensors should satisfy the symmetry relation $A_{l,m} = (-1)^{l+m} A^*_{l,-m}$. It is easy to create spatial tensors where this is not true by manipulating individual components, but many of the routines that use spatial tensors, e.g. in `MAS.h`, will give undefined results if passed "corrupted" tensors[18].

The various creation and deletion operations are:

`space_T(int` *rank*`,int` *flag* `=mxflag::maximum)` creates a tensor of given maximum rank. The activation of the ranks is determined by the optional *flag*; `mxflag::none`—no ranks are activated, `mxflag::maximum`—only the maximum rank is activated (default), `mxflag::all`—all ranks activated. The *contents* are undefined both for active and inactive ranks.

`bool have_rank(`*rank*`)` returns true if the specified rank is active (`false` is returned if *rank* exceeds the maximum rank).

[18] This is a design weakness; the obvious solution is to create a "symmetric tensor" type which enforces this symmetry

`void ensure_rank(`*rank*`)` activates the specified rank (which cannot be greater than the maximum rank). The contents are unchanged by activation or deactivation.

`void clear(`*rank*`)` disactivates the given rank.

`int rank()` returns the maximum rank. This is negative if the `space_T` is undefined.

`int max_rank()` returns the number of the largest *active* rank. A return value of -1 indicates that the tensor contains no active ranks.

The tensor type is unusual in that copy by `=` and copy by initialisation are subtly different. A copy by initialisation, e.g. `space_T A(B)`, ensures that `A` is identical to `B`. When `A=B` is used, however, `A` is already a valid `space_T`. Since resizing the tensor is expensive, this will only be done if the maximum *active* rank of `B` is larger than the maximum *physical* rank of `A`. For example, if `B` was a rank 4 tensor with ranks 3 and 4 inactive and `A` was a rank 2 tensor, the superfluous ranks 3 and 4 would be ignored in the copy, and `A` would remain a rank 2 tensor. If rank 4 of `B` was active, however, `A` would be 'expanded' up to rank 4. This distinction is rarely important in ordinary usage since the same maximum rank is usually used for all tensors in a given problem.

Elements are accessed as usual by the `()` operator e.g. `A(2,0)` returns the rank 2, component 0 element. `(`*l*`)` returns a `BaseList<complex>` referring to the specified (active) rank. The elements are stored in the order $A_{l,-1}$, $A_{l,-l+1}$ etc. This allows tensor contents to be passed as vectors to other functions. Access checking is controlled, as usual, by `NDEBUG`. Attempting to access a rank that does not exist, or is inactive, generates a `BadRank` exception. An invalid order index, e.g. `A(2,3)=4`, generates a `BadIndex` exception. Modified

The operators `*`, `*=`, `/=` and `/` only allow tensor objects to be scaled by real scalars i.e. `double`. The only other mathematical operations defined are the addition and subtraction of two tensors: `+`, `-`, `+=` and `-=`. In this case, inactive ranks *are* taken to be zero, allowing the addition of tensors with different patterns of active and inactive ranks. When a new object is being created (`+` and `-`), the maximum rank of the output is the smaller of the two maximum inputs ranks consistent with the largest *active* rank of new object. The situation is slightly simpler for the in-place operations, `A+=B` and `A-=B` (which should be used if possible); `A` will only be resized if absolutely necessary i.e. the maximum active rank of `B` exceeds the maximum physical rank of `A`.

Naturally space tensors can be rotated. The three angles required to express a general rotation in 3-space are always expressed in radians, and grouped together into the simple `Euler` structure (Section 8.2). Often we need to apply the same rotation to a number of tensors. Calculating the Wigner rotation matrix elements each time is inefficient, so an alternative is to supply the rotation matrix directly.

`space_T rotate(space_T` A`,Euler` Ω`)` returns tensor A rotated by the specified Euler angles.

`space_T rotate(space_T` A`,cmatrix` D`)` rotates a rank L tensor using the Wigner rotation matrix D. This must be a square matrix of order $2L+1$ with element $m+L, n+L$ corresponding to $D(l,m,n,\Omega)$. This form is only applicable if there is a single active rank (L) apart from any isotropic component.

`space_T rotate(space_T` A`,List<cmatrix>` D`)` is a general form of the above which will work with an arbitrary tensor; the Wigner matrix for rank L is stored in element L of the list, which should contain a matrix of the correct dimensions for each active rank of A. Note that element 0 of the list (corresponding to the isotropic component) is ignored.

`complex rotate(space_T` A`,int` l`,int` m`,Euler` Ω`)` returns the (l, m) element of tensor A after rotation. Often we are only interested in a single component of a tensor after a change of reference frame e.g. $A(2, 0)$.

`complex rotate(space_T` A`,int` m`,cmatrix` D`)` returns the (l, m) element of tensor A after rotation defined by the Wigner rotation matrix D. l is determined by the order of D e.g. 5 for a rank 2 tensor.

Irreducible spherical tensors can be constructed from their Cartesian counterparts using the `A1` and `A2` functions for first and second rank tensors respectively:

`space_T A1(double` x`,double` y`,double` z`)` returns a rank 1 tensor

`complex A1(double` x`,double` y`,double` z`,int` l`,int` m`)` returns the (l, m) component of the rank-1 tensor specified by the Cartesian components x, y, z. m can only take the values $0, \pm 1$. The values returned are

$$A^{(1)}_0 = z \qquad A^{(1)}_{\pm 1} = \mp\sqrt{\frac{1}{2}}(x \pm iy) \tag{13}$$

`space_T A2(rmatrix` R`)` constructs a tensor of maximum rank 2 from the matrix of Cartesian components R_{xx}, R_{xy} etc. A `Mismatch` exception is thrown if R is not 3×3. All ranks (0, 1 and 2) are active in the output tensor, although the rank 1 components will be zero if the input matrix is symmetric.

`space_T A2(rmatrix` R`,int` l`)` constructs only one rank, l, of the maximum rank 2 tensor specified by the input matrix, R. This avoids the activation of unwanted components of the tensor.

`complex A2(rmatrix` R`,`l`,`m`)` returns a single component of the spherical tensor. The values are calculated using

$$A^{(2)}_{0,0} = -\sqrt{\frac{1}{3}}(R_{xx} + R_{yy} + R_{zz}) \tag{14}$$

$$A^{(2)}_{1,0} = -i\sqrt{\frac{1}{2}}(R_{xy} - R_{yx}) \qquad A^{(2)}_{1,\pm 1} = -\frac{1}{2}(R_{zx} - R_{xz} \pm i(R_{zy} - R_{yz})) \tag{15}$$

$$A^{(2)}_{2,0} = \sqrt{\frac{1}{3}}(2R_{zz} - R_{xx} - R_{yy}) \tag{16}$$

$$A^{(2)}_{2,\pm 1} = \mp\frac{1}{2}(R_{xz} + R_{zx} \pm i(R_{yz} + R_{zy})) \qquad A^{(2)}_{2,\pm 2} = \frac{1}{2}(R_{xx} - R_{yy} \pm i(R_{xy} + R_{yx})) \tag{17}$$

`space_T A2(double` *xx*`,double` *yy*`,double` *zz*`)` constructs a rank 2 tensor in its PAS from the principle components. By definition the rank 1 component is null and therefore inactive.

`space_T A2(`*xx*`,`*yy*`,`*zz*`,`*rank*`)` constructs only one component of the rank 2 tensor. *rank* must between 0 or 2, otherwise a `BadRank` exception is thrown.

`double A2(`*xx*`,`*yy*`,`*zz*`,`*l*`,`*m*`)` returns a single component of the rank 2 tensor specified by the principal values of the Cartesian representation.

`space_T` is in fact the complex instantation of a general irreducible spherical tensor type `Tensor<T>` i.e. `Tensor<complex>`. This can be used to store "tensors" of arbitrary types, for example `Tensor<double>` can be used to store the Wigner elements $d^{(l)}_{m0}(\beta)$. It is even possible to use a "tensor" of matrices, `Tensor<cmatrix>`.

6.6 Sample spinning

Simulation of NMR for spinning samples is considerably more involved due to the time-dependence of interactions such as the CSA, dipolar interaction etc., but they divide into two basic categories; inhomogeneous vs. homogeneous Hamiltonians. The calculations in the two cases are quite different and so are handled separately:

6.6.1 Inhomogenous Hamiltonians

In this case, the Hamiltonian (and propagator) is defined uniquely by a so-called dynamic phase for each interaction (which must all commute). The propagator for a given time interval is determined from the integrated dynamic phase over the interval, which can be evaluated analytically:

$$U(t_2, t_1) = \exp iH\Phi(t_1, t_2) \tag{18}$$

$$\Phi(t_1, t_2) = \sum_{m=-L}^{L} B_m \int_{t_1}^{t_2} e^{im(\omega_r t+\gamma)} \tag{19}$$

$$B_m = \sum_{l=-L}^{L} d^L_{m0}(\beta_{\mathrm{RL}}) A_{\mathrm{RF}}(l, m) \tag{20}$$

Since H is generally diagonal (i.e. pure z operators), the exponential is trivial. The $m = 0$ component is the time-independent "isotropic phase" (chemical shift etc.), while the $m \neq 0$ terms can be designated the "anisotropic phase". The Φ are purely real *if* no odd rank components are present in the spatial tensors. Since odd-rank terms are only present in the legendary anti-symmetric J-coupling, complex dynamic phases are not allowed (it is not clear what this would mean physically anyway). As a result, all tensors used must contain even ranks only.

The `DynamicPhase` object can be used to calculate the $\Phi(t)$. It is particularly efficient if values are only required at regular steps within the rotor cycle.

`DynamicPhase(`*rotor_speed*`,` *rotor_phase*`,` *N*`,space_T` *A_RF*`,`*rotor_info* `=MASRotorInfo)` sets up the object. The rotor speed is expressed in Hz and may have a sign (setting the sign of the rotation). *rotor_phase* is the rotor phase at $t = 0$. N is the number of observations per rotor cycle. A value of zero leaves this unspecificied (calculation will then be slower). The orientation etc. of the rotor is specified by a `RotorInfo` object which defaults to rotation about the magic angle if omitted. Modified

`DynamicPhase(`*rotor_speed*`,` *rotor_phase*`,` *N*`,`*rotor_info* `=MASRotorInfo)` constructor leaving spatial tensor unspecified.

`double rotor_phase()` and `rotor_phase(`*phase*`)` read and set respectively the rotor phase.

`double rotor_speed()` and `rotor_speed(`*speed*`)` ditto for the spin rate.

`size_t observations()` returns the number of observations per rotor cycle or 0 if unspecified.

`complex component(`m`)` returns the value of B_m (in Hz).

`double component0()` returns B_0, which is purely real by definition. This is the "isotropic" part that is not refocused over the full rotor cycle, consisting of the isotropic chemical shift plus second order quadrupolar shifts etc.

`tensor(space_T` A`)` allows the interaction tensor to be changed. This is useful in powder loops.

`double anisotropic(double` t_1`,double` t_2`)` returns the "anisotropic" component, Φ_{aniso}, of the integrated dynamic phase for the time period specified.

`double isotropic(double` t_1`,double` t_2`)` returns the isotropic component of the integrated dynamic phase, $B_0(t_2 - t_1)$.

`double (double` t_1`,double` t_2`)` returns the total integrated dynamic phase (isotropic + anisotropic) for the time period.

`double instant_phase(`t`)` returns the dynamic phase *at* t rather than an integrated phase.

The following functions are only valid if N has been specified:

`double anisotropic(int` n`)` returns the integrated anisotropic phase for $t = 0$ up to point n in the rotor cycle (numbered from 0). The instantaneous rotor phase *at the end point* is $\gamma + 2\pi n/N$.

`double isotropic(int` n`)` returns the integrated isotropic phase for point n.

`double (int` n`)` returns the total (anisotropic + isotropic) integrated dynamic phase.

`List<complex> propagator(List<double>` H`,`n`)` returns the propagator for $t = 0$ to point n given a Hamilitonian (in its eigenbasis). Other forms: SD.

6.6.2 Homogeneous (system) Hamiltonians

The system Hamiltonian is again described in terms of its decomposition in terms of irreducible spherical tensors:

$$H(t) = \sum_{\lambda} \sum_{l}^{L} \sum_{m=-l}^{l} A^{\lambda}_{\mathrm{RF}}(l,m) d^{(l)}_{m0}(\beta) e^{im(\omega_r t+\gamma)} H^{\lambda} \tag{21}$$

We can sum over the interactions λ and rank l to give

$$H(t) = \sum_{m=-l}^{l} e^{im(\omega_r t+\gamma)} H_m \qquad \text{where} \qquad H_m = \sum_{l}^{L} d^{(l)}_{m0}(\beta) \sum_{\lambda} H_{\lambda} \tag{22}$$

Although this is straightforward to program explicitly, it can be done for a general rotor orientation using the `SpinningHamiltonian` type. The object is set up by adding the different interaction Hamiltonians, specified by their spin and spatial components to the object one by one.

`SpinningHamiltonian(`*rotor_info* `=MASRotorInfo)` creates an empty Hamiltonian using information on the rotor orientation etc. from the supplied `RotorInfo` objects (defaults to MAS). The spinning speed must be set before the object can be used.

`SpinningHamiltonian(`*speed*`,`*phase* `=0,` *rotor_info* `=MASRotorInfo)` creates an empty Hamiltonian including the rotor speed and initial rotor phase.

`add(space_T` A`,`H`)` adds an interaction to the system Hamiltonian, where H can be a full matrix, a "diagonal matrix", or a single element.

`add(space_T` A`,List<cmatrix>` H`)` adds a non-secular Hamiltonian of rank 2. A must be of rank 2, and the contribution to the component m of the periodic Hamiltonian is

$$\sum_{n=-L}^{L} D^2_{mn}(0,\beta_r,\gamma_r) A_{2,-n} H_n \tag{23}$$

L is determined by the number of matrices in H which should be 5, 3 (or 1), where H is a list of $2L+1$ matrices corresponding to $H_{-L} \ldots H_L$.

`add(`H`)` adds an isotropic Hamiltonian e.g. J. This can also be specified using a "diagonal matrix".

`rotor_speed(`*speed*`) and double rotor_speed()` set and return the rotor speed (in Hz). Negative values will effectively reverse the direction of rotation.

`rotor_phase(`*phase*`) and double rotor_phase()` set and return the rotor phase at $t = 0$.

`clear()` resets the Hamiltonian to an empty state.

`cmatrix component(`m`)` returns the mth component of the Hamiltonian Fourier series, H_m.

In-place addition i.e. `+=(`H`)` is used to add in isotropic componenents (full matrices or "diagonal").

If the Hamiltonians are purely real, then the more efficient `RealSpinningHamiltonian` type can be used. `SpinningHamiltonian` objects can be combined using the `+=` and `-=` operators.

The Hamiltonian at time t is read out using the `()` operator[19], either in new-object form *obj*`(double` t`)` or supplied-destination form, *obj*`(cmatrix& H,`t`)`.

Propagators are calculated using `XXXPropagator` objects e.g. for MAS

`MASPropagator<M>(`*obj*`,`ν_r`,`α_0`,`*dt*`)` creates a `Propagator` object from an object returns a Hamiltonian of type `M` as a function of rotor phase e.g. `SpinningHamiltonian` (`MASPropagator<cmatrix>` or `RealSpinningHamiltonian` (`MASPropagator<rmatrix>`). The rotor phase is given by $\phi(t) = \alpha_0 + 2\pi\nu_r$. Note the implied sense of rotation; some treatments use $\phi(t) = \alpha_0 - 2\pi\nu_r$. *dt* is the integration time-step over which the Hamiltonian is assumed to be effectively constant.

The `(`t_1`,`t_2`)` (new object) or `(cmatrix&` U`,`t_1`,`t_2`)` functions then calculate propagators for the time interval t_1 to t_2.

Multiple propagators can be calculated using

`propagators(List<cmatrix>&` *Us*`,`*Propagator_obj*`,`t_1`,`t_2`)` creates n evenly spaced propagators, $U(t_1 + \Delta t, t_1)$, $U(t_1 + 2\Delta t, t_1 + \Delta t)$ to $U(t_2, t_1)$, where $\Delta t = (t_2 - t_1)n$. n is determined by the length of the *Us* vector. This is useful for functions that require a set of propagators over a rotor cycle.

6.7 Propagation

Propagation (taking a set of propagators and calculating a NMR signal) is handled by specialised objects declared in `Propagation.h`. These share the following general characteristics:

- The general scheme is: initialisation of object, specification of propagator(s) or Hamiltonian(s) (where appropriate), specification of initial density matrix, σ_0 and detection operator,Q, read out of FID or spectrum (the last steps are combined in time-domain propagation).

- `set_U(`U`)` or `set_U(`U`,`Δt`)` sets the propagator for the dwell time, Δt, for time- and frequency-domain propagation objects respectively. Note that frequency domain propagation requires the diagonalisation of the propagator in order to determine the effective frequencies, $-\ln(U)/2\pi i\Delta t$. This should be avoided if possible. U can be a full matrix or a diagonal matrix (i.e. `List<complex>`).

- `set_H(`H`)` or `set_H(`U`,`Δt`)` allows the propagation to be specified in terms of the (effective) Hamiltonian over the dwell time for frequency- and time-domain respectively. If H is real (`rmatrix` or `List<double>`) then the eigenbasis is purely real.

[19] In earlier releases, `SpinningHamiltonian` etc. were expressed in terms of the rotor phase rather than time. Attempts to use the objects in this way (i.e. without setting the rotation rate) should be caught at run time

- σ_0 and Q can be complex matrices, real matrices or real diagonal matrices (`List<double>`). Propagation involving real operators is significantly faster if the eigenbasis is also real (see above). If σ_0 and Q are both hermitian, then the FID is necessarily purely real. Hence time-domain objects generally have distinct `add_FID` and `add_FID_hermitian` functions which accumulate the signal into complex and double `List` objects respectively. Inputs are *not* checked for hermiticity.

- Signals are implicitly calculated from $\mathrm{tr}(Q\sigma)$ i.e. if $\sigma_0 = F_+$ (+1 coherence), Q needs to be F_-.

- The fundamental `add_FID(`*FID*`,`*weight*`,`σ_0`,`Q`)` member functions in time-domain objects accumulate the time-domain signal multiplied by a (real) scaling factor, *weight*, (e.g. for powder averaging) to a FID stored as a real or complex vector, for the specified σ_0 and Q. The corresponding `FID(`n`,`σ_0`,`Q objects return a single unweighted FID of n points. If Q is omitted, it is taken as $\sigma_0^\dagger$. Where accepted, this approximately halves the number of similarity transforms required.

- If $\sigma_0 = Q^\dagger$ then the *spectrum* is purely real in most cases. These cases are handled by distinct "matched excitation detection" objects e.g. `StaticSpectrumED`.

- The functions `observe(`σ_0`,[`Q`])` set σ_0 and Q (taken as $\sigma_0^\dagger$ if omitted) immediately prior to frequency-domain propagation (after the Hamiltonian(s)/propagator(s) have been specified). The object is then used as an iterator: transitions (in the form of amplitude, frequency pairs) are read out of the object until the transitions are exhausted:

  ```
  double amp,freq;
  object.observe(σ0)
  while (object(amp,freq))
      std::cout << "Amp:  " << amp << " Freq:  " << freq << std::endl;
  ```

  The resulting transitions can be saved to a file, added to a spectral histogram or FID. The `add_FID(List<complex>&` *FID*`,`*amp*`,`*freq*`)` function can be used to add a signal to a FID, but is generally slow compared to using a histogram (cf. the `Histogram` type) or direct time-domain calculation. It is important to note that the order and number of transitions returned is not defined. Transitions with zero intensity will often be skipped. Amplitudes are real for `ED` objects, complex in the general case.

Static propagation can either be done in the conventional eigenbasis or, more efficiently, in the eigenbasis of the Hamiltonian.

As a general rule, the time-domain methods tend to be more efficient for FIDs of moderate length, while the frequency domain methods are more flexible.

In many important cases, most notably evolution under a free precession, it is only necessary to calculate the evolution within blocks of the density matrix corresponding to certain coherences. Rather than consider the full density matrix and propagators spanning the complete Hilbert space, $\sigma(t) = U(t,0)\sigma(0)U(t,0)^\dagger$, we can consider $\sigma_{RC}(t) = U_R(t,0)\sigma_{RC}(0)U_C(t,0)^\dagger$ where U_R and U_C are propagators for the "row" (bra) and "column" (ket) states of the block,

RC. In general we need to sum the signals from a series of blocks, and the "rows" of one block often make up the "columns" of the next, or vice versa.

This "off-diagonal" propagation is achieved by qualifying Hamiltonians/propagators to indicate whether they are associated with "rows" (`'R'`) or "columns" (`'C'`) of subsequent σ_0 and Q matrices e.g. `set_H('R',`H`)` sets the "row" Hamiltonian to H. `shuffle('R'/'C')` moves any existing "row" (or "column") data into the "column" (or "row"), ready for the "row" (or "column") data to be updated. This allows the off-diagonal blocks to be easily stepped through:

```
obj.set_H('R',Hs(0));
for (int blk=1;blk<blks;blk++) {
    obj.shuffle('R');
    obj.set_H('R',Hs(blk));
    obj.add_FID(FID,1.0,σ0,Q);
}
```

Other points to note

- A single propagator can be used for both diagonal and off-diagonal propagation e.g. subsequently using `obj.set_H(`H`)` will force "diagonal mode".
- The Hamiltonian/propagators for off-diagonal propagation involving a Hilbert subspace of a single dimension can be passed efficiently as single scalar elements i.e. `double` or `complex`.
- The `swap()` member function interchanges row and column designations which is useful if working with blocks both above and below the diagonal.
- Off-diagonal propagation introduces the special case of $Q = \sigma_0 = \mathbf{1}$. This is useful for calculation of the free-precession of a spin S coupled to I spins. Hamiltonians and propagators are blocked according to the spin state of S, e.g. for evolution of the $+1$ coherence for S spin-1/2:

$$S(t) = \mathrm{tr}(S_- U(t,0) S_+ U(t,0)^\dagger) \tag{24}$$

$$U(t,0) = \begin{pmatrix} U_\alpha(t,0) & 0 \\ 0 & U_\beta(t,0) \end{pmatrix} \qquad S_+ = \begin{pmatrix} 0 & \mathbf{1} \\ 0 & 0 \end{pmatrix} \tag{25}$$

  This case is specified by `observe()` and `add_FID(...)` functions that omit σ_0 and Q.

The `StaticSpectrum` and `StaticSpectrumED` objects are used to calculate spectra from systems characterised by a single propagator (or effective Hamiltonian); this applies to simple static problems, or periodic systems which are sampled once per period[20].

There are two time-domain propagation objects for time-independent Hamiltonians, `StaticFID_U` and `StaticFID_H`. The former uses the simple method of repeated similarity transformation of the density matrix by the supplied propagator (from `set_U`) to determine the evolution. `StaticFID_H` uses propagation in the eigenbasis of the (effective) Hamiltonian, specified by either the Hamiltonian or propagator over the dwell time.

[20] If dealing with powder samples under MAS, it is necessary to integrate over the γ angle which can only be done efficiently using the MAS-specific objects

Method	general	hermitian	ED	ED + hermitian
`GammaPeriodicFID`	0.9 (0.7)	0.9	0.6 (0.4)	0.6
`GammaPeriodicSpectrum`	1.0 (0.30)	0.9	0.6	0.7 (0.3)

Table 6: Time taken (ms) for 2 spin-1/2 propagation using time- and frequency-domain methods for: general problem $\sigma_0 = F_x$, $Q = F_+$, hermitian ($\sigma_0 = F_x$ and $Q = F_y$), matched "excitation-detection" ($\sigma_0 = F_+$, $Q = F_-$), ED + hermitian ($\sigma_0 = Q = F_x$). See `runtesthomo` for arguments to `testhomo`. Brackets show times for propagation "blocked" by F_z i.e. propagating the 2×1 $F_z = -1 \leftrightarrow F_z = 0$ and $F_z = 0 \leftrightarrow F_z = -1$ blocks. For comparison, the baseline propagation method (using periodicity of the propagators, explicit γ angle integration) took 3.8 ms. Note that the time taken to calculate the propagators in this case was 3.6 ms, so the differences are not particularly significant overall!

6.7.1 Propagation under a homogeneous periodic Hamiltonian

If the Hamiltonian is periodic, the NMR signal can be calculated efficiently if the sampling is synchronised with the periodicity. In the event where the sampling occurs only once per rotor cycle (or integer multiple of a rotor cycle), the only propagator required is $U(\Delta t, 0)$ (where Δt is the dwell time) and the problem is essentially identical to the static case i.e. the `StaticSpectrum` objects can be used.

Often, however, we are interested in the case where the signal is sampled more than once during the rotor cycle. In general, we can calculate the density matrix at arbitrary time from the set of propagators $U(n\Delta t)$ where $0 < n \leq N$ and $N\Delta t = \tau_r$ (the rotor period)[21]. The specification of the propagators defining the density matrix evolution is done by functions of the form `set_Us(List<cmatrix> Us)`.

The objects `PeriodicSpectrum`, `PeriodicSpectrumED` and `PeriodicFID` are used for frequency domain propagation with general and "matching" σ_0 and Q, and time-domain propagation respectively. The transitions can be restricted to those associated with a single sideband with the member function

`sideband(`m`)` where $-N/2 \leq m \leq N/2$ ($N/2$ and $-N/2$ are equivalent for N even). The restriction is maintained until if and when `sideband` is called again.

The corresponding `GammaPeriodicSpectrum`, `GammaPeriodicSpectrumED` and `GammaPeriodicFID` objects perform the same function, but include integration over the γ powder angle. In this case, the propagators are calculated for the M steps of the powder integration, while the number of observations per rotor cycle is specified when the object is initialised e.g. `GammaPeriodicFID obj(`N`)`. M must be a multiple of N.

Table 6 shows comparisons of time and frequency domain propagation for different pairs of σ_0 and Q. Note how the matched "excitation-detection" is significantly faster, as is the propagation "blocked" by F_z (the difference would be even greater for larger problems).

[21] Usually the periodicity is assumed to be due to sample rotation. It can equally well be periodic RF or even periodic RF + sample rotation (providing they are synchronised).

6.7.2 Propagation under inhomogeneous Hamiltonians

Although purely inhomogeneous Hamiltonians are relatively rare, they can be computed very efficiently and are an important special case. The role played by the propagator for homogeneous Hamiltonians is replaced by a simple integrated dynamic phase $\Phi(t, 0)$. These can be calculated efficiently, especially for rotor-synchronised sampling, using the `DynamicPhase` object. The input to the inhomogeneous propagation functions is simply a `List<double>` of the phases $\Phi(n\Delta t, 0)$, $0 < n \leq N$ ($N\Delta t = \tau_r$). Often the Hamiltonian is made up of a number of interactions with potentially different time-dependencies (and hence dynamic phases) but with mutually commuting Hamiltonians (so the overall Hamiltonian is still inhomogeneous).

The objects `InhomogeneousSpectrum`, `InhomogeneousSpectrumED`, `GammaInhomogeneousSpectrum` or `GammaInhomogeneousSpectrum`, `InhomogeneousFID` and `GammaInhomogeneousFID` work in the same basic fashion as described above, but with the following differences:

- Rather than providing a set of propagators, the evolution is specified by passing a set of (integrated) dynamic phase(s) for the Hamiltonian(s).
- `set_phases(List<double>` *phases*`,[`*period*`])` specifies the set of N or M dynamic phases (with or without gamma integration). The *period* of Hamiltonian must be specified for time-domain methods. Multiple interactions are included by passing a `rmatrix` whose rows correspond to the different interactions.
- `set_phases(DynamicPhase` *obj*`)` provides the dynamic phases in terms of a `DynamicPhase` object. A list of objects is used for multiple-interaction Hamiltonians.
- `set_H(`H`)` or `set_H('R'/'C',`H`)` sets the Hamiltonian for "diagonal" or "off-diagonal" propagation. H can be a real or complex matrix, a diagonal matrix or single element (off-diagonal propagation). `set_Hs(`*Hs*`)` is used for multi-interaction Hamiltonians—the rows of the matrix correspond to the different (diagonal) Hamiltonians.
- The Hamiltonians and dyanmic phases can be updated independently e.g. within a powder integration loop.

The observation and detection operators are set as for `StaticSpectrum` or `StaticSpectrumED`. In addition to their iterator mode of motivation, transitions can then be read out as sideband manifolds from frequency-domain objects:

`double frequency(`r`,`s`,` n `=0)` returns the frequency of sideband n of the r–s transition: $\nu_{\text{iso}} + n\nu_r$. If n is omitted (or 0), the position of the centreband is returned. Assuming ν_{iso} is independent of the powder angle, this is only required outside any powder loop.

`add(List<double>&` *amps*`,`r`,`s`)` accummulates scaled sideband intensities for the r–s transition. This is most useful when the sideband positions (determined by ν_{iso}) are independent of powder angle. Note that some transitions may be "forbidden" i.e. the amplitude is exactly zero. In this case *amps* will be unchanged.

The `add` functions cannot be used simultaneously with the normal iterater behaviour since they both modify internal buffers. They can be used sequentially, however, if the iterator is used *first*.

The transition numbers refer to the eigenbasis of the Hamiltonian, which does not necessarily correspond to the standard Zeeman eigenbasis. Rather than call `add` for each possible transition, the `amplitudes` member function can be used to return the matrix of transition probabilities. The matrix may be real (`double`) or complex, depending on the type of the object. The type used for transition amplitudes is *obj*`.transition_type`.

6.8 Powder averaging

Routines for powder averaging are declared in `powder.h`. The different methods are derived from the base class `PowderMethod`, which can be used either as an iterator or obtaining orientations by their index:

`int next(Euler&` *powder*`,double&` *weight*`)` retrieves the next powder angle in the sequence, updating the Euler angles and the weighting factor for the orientation. The function returns 0 if no more orientations were left.

`void reset()` resets the iterator to the start of the sequence.

`void orientation(Euler&` *powder*`,double&` *weight*`,size_t` *index*`)` returns the Euler angles and weighting factor for orientation *index* where *index* runs from 0 to `orientations()`-1. A `BadIndex` exception is thrown if *index* is out of range.

`size_t orientations()` returns the number of orientations in the sequence.

The `PowderMethod` weighting factors must be normalised i.e. they sum to unity over all the orientations.

The powder averaging methods declared are:

`PlanarGrid(int` *alpha_steps*`,int` *beta_steps*`,` *range* `=sphere)` samples α and β linearly. The weighting factor is $\sin(\beta)$. The *range* can be set to `sphere` (default) to sample the complete set of polar angles, *range* can be one of `sphere`, `hemisphere` or `octant` to average over the ranges:

`sphere` $0 \leq \alpha < 2\pi$, $0 < \beta < \pi$

`hemisphere` $0 \leq \alpha < 2\pi$, $0 < \beta < \pi/2$

`octant` $0 \leq \alpha < \pi/2$, $0 < \beta < \pi/2$

Obviously the problem must have the necessary symmetry before the reduced ranges can be used! A single polar angle can be integrated by setting *alpha_steps* or *beta_steps* to 1. In this case, the unsampled angle is unchanged and the weighting of the points is equal.

`SphericalGrid(int` *alpha_steps*`,int` *beta_steps*`,` *range* `=sphere)` samples the α linearly while β is stepped non-linearly such that the size of the volume element is constant. Although this rarely works better than `PlanarGrid`, it is useful for the cases where the volume element needs to be constant. Note that, unlike `PlanarGrid`, an `InvalidParameter` exception is thrown if either `alpha_steps` or `beta_steps` are one (or less). Use `PlanarGrid` to integrate over a single polar angle.

`PlanarZCW(int` n`,`*range* `=sphere)` uses "ZCW" sampling of α and β treated as planar (rather than spherical variables). The number of points in the sampling is the nth Fibonacci number. $n = 10$, for example, gives 144 points. The weighting factor remains $\sin(\beta)$. Odd as it may seem, this apparently gives better results than the ZCW sampling which treats α and β as angles (`SphericalZCW`)! Again the integration can be altered using *range*.

`SphericalZCW(int` n`,`*range* `=sphere)` uses the ZCW spherical treating α and β as spherical angles i.e. the volume element is constant. Counterintuitively, this is often less effective than `PlanarZCW`, but does have uses when constant volume elements are required.

`ExplicitSampling(rmatrix&` *sampling*`)` allows arbitrary sampling patterns to be used. The sampling matrix must be a 3 column `rmatrix`, with the columns being the values of α, β and the weighting factor in that order. This is convenient for complex sampling patterns stored in files. Note that the contents of *sampling* are *not* copied and so its contents should not be changed while the sampling is being used.

`PowderSingle(Euler` Ω`)` returns the single orientation Ω.

These powder methods only apply for integration over two (or one) Euler angles, but can easily be combined with an independent sampling for the third γ angle either "manually" or using the `WithGamma PowderMethod`:

`WithGamma(PowderMethod` *method*`,` N`)` create a new `PowderMethod` based on a two-angle method but adding averaging over N equally spaced γ values. The total number of orientations is the product of N with the number of steps in the base method.

`WithGamma` copies the `PowderMethod` argument so it is not necessary for this object to persist over the lifetime of the `WithGamma` object e.g. `WithGamma(PlanarZCW(5),8)` will work as expected, even though the original `PlanarZCW` object is destroted at the end of constructing `WithGamma`.

6.9 Superoperators

A very limited number of functions are provided for superoperator arithmetic. Superoperators can be represented by matrices (in Louiville space) and there is little reason to provide a complete new data type. One of the rare times that the superoperator nature of the matrices appears is when they interact with matrices expressed in the Hilbert space e.g. $\hat{\hat{U}}\sigma$ where σ and $\hat{\hat{U}}$ are $n \times n$ and $n^2 \times n^2$ matrices respectively. Rather than "flatten" σ into a 1 by n^2 vector for the purposes of multiplication, the following function is provided

`superop_multiply(cmatrix&` C`,cmatrix` A`, cmatrix` B`)` where $C = AB$. If $n_A = n_B^2$, A is assumed to be the superoperator and B the ordinary matrix, and vice versa if $n_B = n_A^2$ (right and left multiply respectively).

The following functions generate operators in Louiville space

`cmatrix commutator(cmatrix` H`)` creates the commutator $[H,]$ $(H \otimes I - I \otimes H^T)$.

`cmatrix double_commutator(cmatrix` H`)` creates the double commutator $[H,[H,]]$ ($H^2 \otimes I + I \otimes (H^T)^2 - 2H \otimes H^T$).

These functions are also defined for real matrices.

See also Section 6.3 for functions related to chemical exchange and other "spin permutations".

6.10 `Sequence.h`

`Sequence.h` contains data types and code for the definition of general pulse sequences, together with the code for calculating propagators during pulse sequences for both static and spinning experiments. For simple pulse sequences, it is simpler to use the `Upulse` and `propagator` functions to calculate sequence propagators. For complex sequences and/or MAS pulses sequences, this becomes too cumbersome and this more sophisticated approach is required.

The steps involved are

1. Create the elements of the pulse sequence, hard/soft pulses, decoupling fields etc.
2. Assemble elements into a pulse sequence e.g. WHH4, MREV8 etc.
3. A `Propagator` object is derived from the sequence to evaluate propagators.

The parameters and timing of the pulse elements can be modified without recreating the whole sequence. Thus makes it straightforward to optimise sequences.

6.10.1 Creating the pulse sequence elements

The elements of pulses sequences such as hard pulses, CW RF periods etc., are created as objects (`HardPulse` and `CWPulse` respectively) e.g. a 90 degree x pulse on ^{1}H. The objects contain a reference to a supplied `PulseGenerator` object which is used to calculate the propagators required. The `PulseGenerator` objects must remain in scope through the calculation.

The `RFEvent` objects currently defined are:

`HardPulse(PulseGenerator` *pgen*`,double` *duration*`,double` *vRF*`,double` *phase*`)` creates a hard pulse object. The nutation angle is specified in terms of an RF nutation rate, ν_{rf}, and nominal pulse duration (i.e. $\theta = 2\pi t/\nu_{\mathrm{rf}}$). The duration and nutation rate have no other significance for a hard pulse, i.e. the pulse is always of zero duration as far as timing is concerned.

`SoftPulse(PulseGenerator` *pgen*`,double` *duration*`,double` *vRF*`,double` *phase*`,double` *offset*`)` creates a soft/non-delta function pulse of given duration and resonance offset. An `InvalidParameter` exception is thrown if the duration is negative.

`CWPulse(PulseGenerator` *pgen*`,double` *duration*`,double` *vRF*`,double` *phase*`,double` *offset*`)` creates an "RF field" object, expressed in terms of resonance offset and power (in Hz). `CWPulse` and `SoftPulse` are basically different ways of expressing a finite RF field. *The duration of a* `CWPulse` *is expressed in terms of the cycle time scaling (see below) i.e. its duration may scale as the overall duration of the sequence changes. The duration of* `SoftPulse`*, however, is expressed directly in seconds and is not scaled.*

A sequence consists of a series of "timed events" i.e. `RFEvent` objects together with a time parameter that specifies when the event occurs.

`vRF`, `phase`, `duration` and `offset` can be freely modified after the object has been created e.g.

`duration(`t`)` sets the duration object to t. Note again the different units for the duration parameter of `CWPulse`.

`duration()` returns the duration of *object*.

6.10.2 Creating the pulse sequence

A sequence object is a list of `TimedEvent` objects (`RFEvent`s plus a timing) created with

`Sequence(`*timescale*`)` where *timescale* is an optional 'time scaling' factor (e.g. the rotor period for rotor sychronised sequences). If omitted, this defaults to 1 i.e. the timings are expressed directly in seconds.

Events are added using

`TimedEvent& push_back(RFEvent` ev`,`t`,char` *sync* `='|')` adds a single event, returning a reference to the `TimedEvent` object created. t is the time origin of the event. This will be either an absolute time (in s) or as a fraction of the "timescale". The synchronisation character (`'+'`, `'-'` or `'|'`) dictates how the start and end points of `SoftPulse` events are related to the time parameter. With `'+'`, the rise of the pulse is synchronised with the time origin; `'-'` synchronises with the end of the pulse, while `'|'` means that the centre of pulse is synchronised. Synchronisation is not applicable to `CWPulse` (effectively always `'+'` sychronised). In the case of `HardPulse` events, the synchronisation flag is a 'hint' indicating whether a pulse should be included within an integration interval. For instance, if a `'-'` pulse is encountered at the end of an integration interval, it will be included, while a `'+'` event is assumed to belong to the next interval (e.g. rotor cycle) and will not be included. A `'|'` event exactly at an interval boundary is ambiguous and will trigger a warning.

`push_back(List<RFEvent>` *evs*`,`t`,char` *sync* `='|')` adds a windowless sets of events. The first element is at time t, the second $t + \Delta t_0$ where $\Delta t + 0$ is the duration of event 0, etc. The same synchronisation is used for all elements.

The `Sequence` thus consists of an ordered list of `TimedEvent` objects which combine an `RFEvent` with a time point and sychronisation indicator. This time point can be altered subsequently using the `time(double` t`)` member function.

Note that the `TimedEvent` objects contain a reference to the original `RFEvent` object. Modifying this object, e.g. the tip angle of a pulse object, will therefore change this pulse wherever it has been used. In most situations this is exactly what is needed. If, for example, the 90° pulse is mis-timed, this should be reflected in all the 90° pulses etc. If, on the other hand, two 90 degree pulses have very different functions e.g. one is sets the initial polarisation, while another is part of a synchronised pulse sequence, then it is better to create two distinct `Pulse` objects.

`period(double` *period*`)` sets (or changes) the period of the sequence. This allows, for instance, the rotor speed to be changed easily. A period of 0 corresponds to no scaling (static sequence). A negative period is illegal.

`period()` returns the period of the sequence, `0` if none.

A `SequencePropagator` object is used to evaluate propagators. The object is created with one or more `Sequence` objects together with a "period". A non-zero period indicates that the sequence is cyclic and should be should be repeated with the given period. This period is independent of any period parameter used to establish the timing of the pulses, although these would normally be the same e.g. for rotor synchronised sequences. If the period is zero, the sequence is not repeated. Using more than one pulse sequence is necessary when there is more than one "channel", particularly if the pulses on different channels are not synchronised. It is not strictly necessary to dedicate a sequence to a particular nucleus; the only requirement is that the elements within any given sequence be time-ordered.

`SequencePropagator(`H`,` *intdt*`,` *sequence*`,`*period*`,`*tol* `=1e-9,` *verbose* `=0)` initialises the object with a single sequence and associated period. H is the system Hamiltonian. *indt* is the integration for time-dependent Hamiltonians (omit for static Hamiltonians). *verbose* controls how much information is output during evaluation (0 corresponding to none). The *tol* parameter is described below.

`SequencePropagator(`H`,` *intdt*`, BaseList<Sequence>` *seqs*`,`*periods*`,`*tol* `=1e-9,` *verbose* `=0)` initialises the object with multiple sequences. *periods* can either be a single, common period or a `BaseList<double>` of individual periods for the different sequences.

Note that H should normally be one of the `Blocked` Hamiltonians described in `MetaPropagation`[22].

The optional tolerance parameter, *tol*, allows for imperfect pulse timing. If, for instance, an RF event is scheduled at exactly 5 ms, and the propagator is calculated up to a nominal $t = 0.005$, we cannot be certain whether the pulse is included or not, since the accumulated time, `time()`, may not be exactly 0.005000 due to rounding errors. Often this is not important; if the pulse has not been included in this time interval, it will be in the next one. An exception is when the end of the propagator calculation coincides with the final pulse which *must* be included if the calculation is to be meaningful. Under these circumstances *tol* should be set to a small (positive) non-zero value. Any `'-'` synchronised events occurring in the "buffer" period $time+incr \pm tol/2$ are included in the propagator. `'+'` synchronised events encountered in this interval are *not* included and force the calculation to stop, since this indicates that another segment has started. `'|'` events are included, but generate warnings. The synchronisation flags should be used whenever possible to help the `SequencePropagator` decide whether an event is to be included.

Most `Propagator` objects calculate propagators for arbitrary time intervals (t_1–t_2). `Sequence`'s, on the other hand, are best evaluated in strict time order, as it is relatively "expensive" to

[22] An internal interface "describes" the different Hamiltonians for the benefit of `SequencePropagator` and other high-level objects. Compilation will fail with rather cryptic error messages if you try to pass an object that is not compatible with this interface.

"break into" to a sequence. The `SequencePropagator` object stores the previous state so that sequence evaluation i.e. t_1–t_2 followed by t_2–t_3 etc. is evaluated efficiently. If the following time interval does not follow from the stored state, the object must be "reset", making the evaluation much less efficient.

6.11 Data processing

`Histogram.h` creates a number of histogram types that can be used to accummulate spectra. These are all derived from the `BaseHistogram` type which declares the (virtual) function `add(T` a`,double` f`)` which adds an amplitude a (of type `T`) to the histogram. The histogram objects are created from a `List<T>` (or `BaseList<T>`) and essentially simply define a method for adding data to the list which is created and can be manipulated outside of the the histogram. Obviously though the underlying `List` should not be destroyed or have its length changed while the histogram is active! The following histogram types are provided

`Histogram(BaseList<T>` *vector*`,`$f_{\mathtt{max}}$`)` creates a histogram whose "x" scale runs from 0 to Δf. The number of bins, N is determined by the length of data list, *vector*. *vector* is *not* reset to zero. The bins are *centred* at 0.0, $f_{\max}/N$, $2f_{\max}/N$ etc. and run from 0 to $N-1$. So small negative values, for instance, will fall into bin 0, while values close to $f_{\max}$ will fall outside the histogram. Values that fall outside the histogram are ignored.

`FoldingHistogram(BaseList<T>` *vector*`,`$f_{\mathtt{max}}$`)` creates a histogram whose "x" scale has a period $f_{\max}$, that is, values are folded into the range 0 to $f_{\max}$ by adding/subtracting multiples of $f_{\max}$. Values cannot be "lost" from such a histogram.

`InterpHistogram(BaseList<T>` *vector*`,`$f_{\mathtt{max}}$`)` creates a histogram for the x range 0 to $f_{\max}$, but in which intensity is distributed between the two bins centred at x_0 and x_1 which straddle a given data item, x. The fractions of the intensity a which are added to the two bins are given by

$$f_0 = \frac{x - x_0}{x_1 - x_0} \qquad f_1 = 1 - f_0 = \frac{x_1 - x}{x_1 - x_0}$$

The interpolation of intensity helps to reduce the rounding error caused by "quantising" the continuous x scale on to the discrete number of bins. It does not, however, address the root of this rounding error and it is generally more satisfactory to increase the number to bins to reduce the error.

`FoldingInterpHistogram(BaseList<T>` *vector*`,`$f_{\mathtt{max}}$`)` combines a periodic x-scale with linear interpolation.

The following functions, declared in `cmatrix_utils.h`, implement Fast Fourier Transforms for one and two-dimensional data sets:

`void fft_ip(BaseList<complex> &`*vector*`,int` *sign* `=FT_FORWARD,double` *scale* `=1)` performs the Fast Fourier Transform of a vector (passed as a `List` of complex numbers. The *sign* should be ± 1. To improve code readability and minimise inconsistencies the constants

FT_FORWARD (−1) and FT_BACKWARD (1) can be used to express "normal" Fourier transformation of a time-domain signal and "back" Fourier transformation of a spectrum respectively. The first data point is multiplied by the *scale* factor before transformation. For signals that decay towards zero (as opposed to true periodic signals) this should normally be set to 0.5 to avoid putting an offset in the spectrum

`List<complex> fft(BaseList<complex>` *vector*`,`*sign* `=FT_FORWARD,`*scale* `=1)` is the corresponding new-object form.

`void fft_ip(cmatrix &`*matrix*`,`*sign* `=FT_FORWARD,`*scale* `=1)` Fourier transforms each row of data matrix in place. Note this is not a full 2D transform. Also: `fft`.

`void phasefft_ip(cmatrix &`*matrix*`,`*sign* `=FT_FORWARD,`*rscale* `=1,`*cscale* `=1)` performs a 2D Fourier transform on a "phase-modulated" data set (i.e. one with no sign discrimination in the t_1 dimension). This is equivalent to (but significantly more efficient than) the sequence `fft_ip`, `transpose`, `fft_ip`, `transpose`. The scaling factors for the rows and columns can be set separately e.g. for data sets with both imaging and spectroscopic dimensions. Other forms: new-object (`phasefft`).

`void ampfft_ip(cmatrix &`*cos*`,cmatrix &`*sin*`,`*sign* `=FT_FORWARD,`*rscale* `=1,`*cscale* `=1)` Fourier transforms an "amplitude modulated" (States) data set described by a set of "cosine FIDs" and "sin FIDs" (there is little need for a true hypercomplex matrix type). The result is returned "in place" with the complex component in the first argument and the (generally uninteresting) hypercomplex component in the second matrix argument.

`void ampfft_ip(cmatrix &`*cos*`,`*sign* `=FT_FORWARD,`*rscale* `=1,`*cscale* `=1)` performs an amplitude-modulated FT assuming a purely cos modulated data set. Only the "normal" complex part of the result is retained. Other forms: new-object (`ampfft`).

As usual, the in-place functions should be used if possible. Note that only the simple base-2 Fast Fourier Transform is used, i.e. the number of data points must be exactly a power of two otherwise an `InvalidParameter` exception is thrown. The function `ispowerof2(int` n`)` returns a true value if n is a (positive) power of two. Note that $n = 1$ is now considered a power of 2 !

It is also possible to perform "ordinary" Fourier transforms, although these are significantly slower:

`List<complex> ft(BaseList<complex> &`*vector*`,int` *sign* `=FT_FORWARD,double` *scale* `=1)` performs the "slow"Fourier transform of a vector (passed as a `List` of complex numbers. The *sign* should be ±1. The first data point is multiplied by the *scale* factor before transformation. Other forms: SD.

`List<complex> real_ft(BaseList<complex> &`*vector*`,int` *sign* `=FT_FORWARD,double` *scale* `=1)` is useful when only the real part of the transform is required. It is equivalent to (but quicker than) `real(ft(`A`))`.

`cmatrix ft(cmatrix` *matrix*`,`*sign* `=FT_FORWARD,`*scale* `=1)` performs a Fourier transform on each row of the matrix. Note that this is relatively efficient since the Fourier coefficients are calculated once and re-used for each row. Other forms: SD.

It is usually necessary to damp calculated Free Inductions decays by exponential apodisation functions in order to have a finite linewidth (Fourier transformation of signals with an infinite linewidth invariably leads to strange output):

`void (rc)matrix exponential_multiply( (rc)matrix` A`,double` *rcons*`,double` *ccons*`)` returns the result of appliying an exponential damping to a 2D data set, A. *rcons* and *ccons* are the damping factors for the row and column dimensions respectively. The damping factor is expressed as the number of time constants covered by the FID i.e. a value of 3.0 means the damping factor at the tail of the FID will be $\exp(-3)$. A zero time-constant implies no damping in the dimension.

`void exponential_multiply_ip( (rc)matrix` A`,double` *rcons*`,double` *ccons*`)` is an in-place version of the function i.e. A is replaced by its damped version.

6.12 "Meta Propagation"

The objects defined in `Propgation.h` encapsulate the messy business of calculating an individual NMR signal. Even using these objects, complex problems involving, for example, block diagonal Hamiltonians, symmetry factorisation of the Hamiltonian are still rather tedious. The objects created in `MetaPropagation.h` allow simulations to be expressed at a yet higher level, allowing complex problems to be expressed in a compact fashion.

In overview, a typical simulation will involve

1. Creating a `HamiltonianStore` object that defines what NMR interactions are active.
2. Creating an "operator generator" to create the Hamiltonians. This will normally be a `SpinOpGenerator`, but different generators may be defined e.g. `CrystalOpGenerator` for problems with periodic symmetry.
3. Creating a "blocked" Hamiltonian e.g. `BlockedSpinningHamiltonian` that defines the (system) Hamiltonian. "Blocking" refers to any block-diagonal structure e.g. "m_z" blocking for free precession Hamiltonians, or "eigenvalue" blocking when the Hamiltonian is blocked by some additional symmetry. `BlockedOperator` objects store other spin operators e.g. density matrices.
4. Creating a `MetaPropagator` object which generates propagators for a given system Hamiltonian combined, if required, with RF (defined by `Sequence`).
5. Using a `MetaSpectrum` object or `add_MetaFID` to generate the spectrum or FID.

Note how the "encapsulation" of the NMR simulation allows steps such as the creation of the Hamiltonian to be "automated". The price is loss of low-level control; you can only do with the objects what they allow you to do.

Considering each of these steps in turn...

6.12.1 `HamiltonianStore<T>`

is used to store the values of NMR interactions. `T` can be either `double` (static problems) or `space_T` (spinning problems). It is created using

`HamiltonianStore(`M`,`N`)` creates an empty interaction list for M spins repeated N times. N is 1 is omitted; other values of N are used for periodic problems.

`HamiltonianStore(HamiltonianStore<space_T>` A`, Euler` Ω`)` creates a new `HamiltonianStore` from A after rotating all interactions through Ω. A must be defined in terms of `space_T`, although the destination can be either a `HamiltonianStore<double>` or `HamiltonianStore<space_T>`.

`HamilonianStore` understands periodic systems. All N spin systems are defined to be identical, so the range of spin indices for shifts, δ_i and couplings $d_{i,j}$ is $0 \leq i < M$, $0 \leq j < MN$.

Interactions are added / read using

`set_dipole(`i`,`j`,T` d`)and T& get_dipole(`i`,`j`)` sets or reads out the magnitude of the dipole coupling (as a `double` or `space_T`) between spins i and j (indexed from 0). The couplings are always mutual, so it not possible (or necessary) to set the coupling between j and i independently.

`set_jcoupling(`i`,`j`,T` J`)and T& get_jcoupling(`i`,`j`)` Ditto for J coupling.

`set_shift(`i`,T` δ`)and T& get_shift(`i`)` Sets / returns the chemical shift (Hz rather than ppm) for spin i.

`set_quadrupole(`i`,T` δ`,` *order*`)and T& get_quadrupole(`i`)` Sets / returns the quadrupole coupling for spin i. *order* is the order to which the quadrupole Hamiltonian is treated. Currently this is limited to 1!

`clear()` removes all interactions.

`verify(ostream&, double` *tol*`)` returns true if the coupling network is properly periodic, otherwise output on the supplied stream indicates couplings that are incompatible with the periodic structure.

Interactions that are not defined or set to zero are ignored when Hamiltonians are created.

An "operator generator" is used to construct spin operators. Unlike the basic `spin_system` objects etc., these objects understand block structure, periodicity, and can interface into other objects such as `MetaPropagator` and `MetaSpectrum`. The default generator is `SpinOpGenerator`, which is constructed from

`SpinOpGenerator(`*sys*`,` N`)` creates an operator generator for a set of spins, repeated N times for periodic systems. *sys* is any `basespin_system` object and is used to create the underlying spin operators.

`SpinOpGenerator(`*sys*`,` N`, List<nuclei_spec>` *blockingnucs*`)` imposes an "m_z" block structure. *blockingnucs* is a list of nuclei for which F_z is a good quantum number. Usually this will correspond to any nucleus that is not being irradiated.

`SpinOpGenerator(`*sys*`, `*N*`, char* `*label*`)` is a simplification for the case when only one nucleus is "blocked" e.g. `SpinOpGenerator(`*sys*`,1,"13C")`.

Static and spinning Hamiltonians are stored in `BlockedHamiltonian<OpGen>` and `BlockedSpinningHamilt` respectively, where `OpGen` is the type of an operator generator (defaults to `SpinOpGenerator`). These are created using

`BlockedHamiltonian(`*opgen*`,`*Hstore*`)` where *opgen* is an operator generator, and *Hstore* is a `HamiltonianStore<double>`.

`BlockedSpinningHamiltonian(`*opgen*`,`*Hstore*`,`*rotor_speed*`,`*rotor_phase*`,`*RotorInfo*`)` creates a spinning Hamiltonian on this basis of a set of interactions (stored in a `HamiltonianStore<space_T>`) and a spinning specification: spin rate, rotor phase at $t = 0$ and rotor orientation (MAS by default).

Density matrices, spin operators etc. are stored in the `BlockedOperator` type. These are created using

`BlockedOperator(`*opgen*`, `*opspec*`)` where *opgen* is an operator generator and *opspec* is an "operator specifier" which defines the spin operator. This is turn is created from `operator_spec(`n`,char `*op*`)` where n is a spin indexer e.g. `operator_spec(1,'x')` would refer to I_{1x}. Sum operators are created from `operator_spec(char* `*nuclabel*`,`*op*`)` e.g. `operator_spec("1H",'x')` corresponds to F_x for the ^{1}H spins.

Note that the `BlockedOperator` type is effectively "closed"; unlike operators stored in lower level structures such as `cmatrix`, it cannot be arbitrarily manipulated. Only a limited number of operations are currently defined for `BlockedOperator`. It is essentially a means of packaging a block diagonal operator in a single package with all the work being done on the individual blocks.

For time-dependent problems, it is then necessary to create a `MetaPropagator` object to generate propagators for specified time intervals. These are created from

`MetaPropagator(`H`,`*dt*`)` where H is any valid Hamiltonian type e.g. `BlockedHamiltonian` and *dt* is the integration timestep for time-dependent Hamiltonians. This should be omitted (or set to zero) for time-independent problems.

Propagators are calculated using *propgen*`(BlockedMatrix<complex>& `*dest*`,`t_1`,`t_2`)`. `SequencePropagator` should be used instead of `MetaPropagator` if time-dependent RF is required.

Frequency domain propagation is performed with `MetaSpectrum<`*obj*`>` or `MetaSpectrumED<`*obj*`>` where *obj* is one of the `Spectrum` or `SpectrumED` objects from `Propagation.h`. These iterators, which used in the same way as the basic `Spectrum` objects, are created using

`MetaSpectrum(`H`,`σ_0`,`Q`,`*spectrum_obj*`)` where H is any suitable time-independent Hamiltonian type e.g. `BlockedHamiltonian` and σ_0 and Q are corresponding objects, e.g. `BlockedOperator`, containing the initial density matrix and detection operator. The optional *spectrum_obj* provides a means of passing arguments into the basic iterator objects since this argument is copied to create the necessary objects. For instance, γ-integrating objects require the number of sampling points to be specified when the object

is created e.g. `GammaPeriodicSpectrum(`*nobs*`)`. The default object constructor will be used if this argument is not present.

`MetaSpectrum(`*propgen*`,`n`,`t_1`,`t_2`,`$\sigma 0$`,`Q`,`*spectrum_obj*`)` is used for time-dependent problems. n propagators are calculated using one of the propagator-generator objects e.g. `SequencePropagator` over the time interval t_1 to t_2.

The `add_MetaFID` function is used for time-domain propagation. Note this is a function rather than an object as all the input arguments are passed in one step.

`add_MetaFID(`*FID_obj*`, List<complex>& `*FID*`,` *scale*`,`H`,`σ_0`,`Q`)` is used for time-independent propagation. *FID_obj* is one of the `FID` objects. A concrete object is passed so that any arguments to the `FID` object can be supplied. Q can be omitted for matched detection-excitation ($Q = \sigma_0^\dagger$).

`add_MetaFID(`*FID_obj*`, List<complex>& `*FID*`,` *scale*`,`*propgen*`,`n`,`t_1`,`t_2`,`$\sigma 0$`,`Q`)` is the time-dependent equivalent.

7 Optimisation and Data fitting

Various functions for data fitting and functional minimisation are declared in `optim.h`. `libcmatrix` does not provide functions itself for functional optimisation beyond a basic slow-and-steady simplex optimisation, but does provide an interface to the MINUIT routines.

7.1 Data fitting

The `fitdata` function uses the Levenburg-Marquardt algorithm (adapted from Numerical Recipes) for least-squares fitting of experimental data. Although least-squares fitting can be treated as a particular case of functional optimisation, where the function being minimised is the χ^2 difference between the trial and calculated data, it is more efficient to take advantage of the particular properties of the χ^2 function. For some problems, however, e.g. when several parameters are very strongly correlated, the Levenburg-Marquadt method is insufficiently robust. In these situations `simplex_fitdata` can be used.

The `fitdata` functions require "function objects" of the form:

Modified

```
class myfunction :  public BaseFitFunction { Must be derived from BaseFitFunction
    ...                                      Any private data
public:
    void operator()(BaseList<double>& dest, const BaseList<double>& paras) const;
                                             return trial function corresponding to parameters paras in dest
}
```

The advantage of using a function object rather than a function is that extra data can be passed cleanly within the object itself.

The `fitdata` function itself takes the following arguments

`double fitdata(`*covar*`,`*paras*`,`*funcobj*`,`*data*`,`*which*`,`*errs*`,`*noise*`,`*verbose*`,`*chistop*`,`*maxsteps*`)`

The input parameters are

`BaseList<double>` *paras* are the initial values of the parameters. This vector is updated the fitted parameters when the fitting is finished.

funcobj is the model function object

`BaseList<double>` *data* is the "experimental" data.

`BaseList<size_t>` *which* is a vector specifying which parameters are to be adjusted. e.g. if *which* is (0,2,3), then parameters 0, 2 and 3 will be fitted, while parameter 1 will be fixed. Note that items are numbered as usual from 0. This scheme makes it easy to change which parameters are being fitted without having to modify the model function.

`BaseList<double>` *errs* is a vector containing estimated errors for each of the parameters. This is only used to set the "scale" for each parameter i.e. whether the parameter should be adjusted by 1e6 or 1e-6, so the estimates need only be order of magnitude.

`BaseList<double>` σ or `double` σ sets the noise level for each point in the data set. The least-squares function being minimised is

$$\chi^2 = \sum_i \frac{(\text{trial}_i - \text{data}_i)^2}{\sigma_i^2} \tag{26}$$

where σ_i is the error on data point i. In most circumstances, the noise level is constant across the data set and so only a single value of σ need be specified. It is important that estimate of σ is reasonable for the χ^2 statistic to have any meaning. Generally the fitting can be assumed to satisfactory is $\chi^2 \approx N$ at the minimum where N is the number of data points, but this is clearly only meaningful if σ is correct. Similarly the estimated errors of the parameters (returned in the covariance matrix) are dependent on σ being correct.

`int` *verbose* `=0` controls the amount of output from the routine; 0 being the minimum.

`int` *chistop* `=0.0` sets the change of χ^2 used as the stopping criterion. If χ^2 changes by less than this amount during an iteration, the fitting is assumed to have converged. If 0 is passed, a value of 1e-4 times the number of data points is used as a default; this is only valid if σ has been set correctly.

`int` *maxsteps* `=1000` sets the maximum number of iterations. A `Failed()` exception is thrown if *maxsteps* is exceeded.

On exit, `fitdata` returns the final value of χ^2 and the covariance matrix, V, in *covar*. This can be used to give error bounds on the parameters and correlations between them. The error on parameter i is simply $\sqrt{V_{ii}}$ while the correlation coefficient between parameters i and j is

$$\text{cor}_{ij} = \frac{V_{ij}}{\sqrt{V_{ii}V_{jj}}} \tag{27}$$

Again the error estimates are only meaningful if σ is correct. Note that *covar* only spans the *variable* parameters; row i of *covar* corresponds to parameter *which(i)*.

Data can also be fitted by simplex optimisation. `simplex_fitdata` is used in the same way as `fitdata`, cf. the `testsimplex` program. Unlike `fitdata` no exception is thrown if the maximum number of steps of exceeded; simplex optimisation usually benefits from being "restarted" a few times. When restarting, it is usually a good idea to reduce the error estimates to prevent the search volume being expanded too far.

7.2 Functional minimisation

The `simplex` function can be used for undemanding functional optimisations. The input arguments are essentially similar to those for `fitdata` and `simplex_fitdata`.

Here the function object is of the form

```
class myfunction :  public BaseMinFunction { Must be derived from BaseMinFunction
    ...                                      Any private data
public:
    double operator()(const BaseList<double>& paras, int when =0) const;
                                             return function value for parameters paras
}
```

The *when* parameter is used by some minimisation routines, notably `MINUIT` to indicate at what stage of the minimisation function is being called (see the MINUIT section for details). Other methods leave this value set at zero.

`double simplex(`*paras*`,`*funcobj*`,`*which*`,`*errs*`,`*verbose*`,`*stopval*`,`*maxsteps*`=50)` returns the minimum value of *funcobj* found, updating the initial values of the parameters stored in *paras*. *stopval* is the change in the *funcobj* value below which the optimisation stops (this parameter is now obligatory). *errs* is used to set the volume of the simplex centered on the initial parameter values.

More sophisticated functional optimisation can be done using the `MINUIT` routines. `MINUIT` is now developed as as a C++ library and can be accessed directly (see example in `test/testminuit.cc`). `optim.h` simply provides a function adaptor which takes a `BaseMinFunction` and adapts it to the form expected by `MINUIT`[23]. Modified

8 Miscellaneous

8.1 Random numbers and noise

The following functions used to return random numbers are declared in `cmatrix_utils.h`

`double random(double` x`)` returns a random number between 0 and x.

[23] This simply involves transferring between the `List` type used in `BaseMinFunction` to the `std::list` expected by `MINUIT`.

`double gauss(double` σ `=1.0)` returns a random number from a normal distribution of standard deviation σ.

`void set_seed(int` *seed*`)` is used to set the "seed" value for the random number generator. Once the seed is set to a particular value, the numbers generated will follow the same sequence. `set_seed()` without an argument will set the seed to a "random" value based on the system clock. Because the clock only changes every second, calling `set_seed()` too frequently will reset the generator to exactly the same point!

It is important to note that `libcmatrix` sets the seed to the same value each time it starts. Hence `libcmatrix` programs will always use the *same sequence* of noise values in each run unless the seed changed.

8.2 Euler angles and Wigner rotation matrix elements

The functions for calculating Wigner rotation matrix elements, and the declaration of the `Euler` structure are found in `"wigner.h"`. This can be used independently of the rest of `libcmatrix`. `Euler` is declared as

```
struct Euler {
  double alpha;
  double beta;
  double gamma;
  Euler(double alpha,double beta,double gamma);
};
```

Because `Euler` is simply a structure, its components can be accessed directly e.g. `PAStoMF.beta=M_PI/4`. The constructor allows compact initialisation e.g. `Euler PAStoMF(0,0,0)`. Note that output of an `Euler` structure (using `<<` Ω) prints the angles in *degrees* although α, β and γ are always stored and input in radians.

The Wigner elements are returned by

`double d1(int` m`,int` n`,double` β`)` returns $d^{(1)}_{mn}(\beta)$

`double d2(int` m`,int` n`,double` β`)` returns $d^{(2)}_{mn}(\beta)$

`double d(int` l`,int` m`,int` n`,double` β`)` returns $d^{(l)}_{mn}(\beta)$. For $l = 1, 2$, the functions `d1` and `d2` are called, otherwise the element is calculated *(code used without validation).*

`double D(int` l`,int` m`,int` n`,double` α`,double` β`,double` γ`)` returns the full matrix elements $D^{(l)}_{mn}(\alpha, \beta, \gamma)$, which are the products of the `d()` term and the `wigner_factor()`.

`complex D(int` l`,int` m`,int` n`,double` α`,Euler` Ω`)` returns the full matrix elements $D^{(l)}_{mn}(\Omega)$. In the interests of efficiency, these functions should be avoided in favour of factorisation into the reduced matrix element and a phase factor.

`complex wigner_factor(int` m`,int` n`,double` α`,double` γ`)` returns the phase factor for the full Wigner rotation matrix element: $\exp[-i(m\alpha + n\gamma)]$. This function is made accessible to allow these elements to calculated consistently, since there is more than one convention.

`cmatrix D(`l`,`Ω`)` returns the complete rank l rotation matrix. Useful if rotating several tensors through the same angle.

`double legendre(int` n`,double` β`)` returns value of the Legendre polynomial of order n, $P_n(\beta) = d^{(l)}_{n0}(\beta)$. $n = 0, 1, 2$ are treated as special cases, otherwise the value of $d^{(l)}_{n0}$ is calculated explicitly.

8.3 3D geometry

`geometry.h` defines the simple data types `vector3` and `spherical` for storing 3D vectors and position in spherical coordinates respectively.

The `vector3` type is simply

```
struct vector3 {
  double x,y,z;
  vector3(double,double,double =0);
};
```

The components are accessed directly, e.g. `A.x=5.0`. The constructor allows initialisations of the form `vector3(x,y,z)` or `vector3(x,y)` (where the z component is set to zero).

Apart from the obvious addition, subtraction and scaling operations, the following functions are also defined

`double` *vector*`.length()` returns the magnitude of the vector, $|v|$.

`void` *vector*`.rotate(Euler` Ω`)` applies the rotation specified in terms of Euler angles to the vector. Also exists as a new-object function `rotate(`*vector*`,`Ω`)`.

`void` *vector*`.rotate(double` R`[3][3])` applies a rotation specified as a 3×3 rotation matrix. Also exists as a new-object function `rotate(`*vector*`,R)`.

`void rotation_matrix(double` R`[3][3],Euler` Ω`)` returns the 3×3 rotation matrix corresponding to rotation Ω. A simple double array is used to store the matrix.

`double dot(vector3` v_1`,vector3` v_2`)` returns the dot product $v_1 \cdot v_2$

`vector3 cross(vector3` v_1`,vector3` v_2`)` returns the cross (outer) product $v_1 \times v_2$.

`double vector_angle(vector3` v_1`,vector3` v_2`)` returns the angle (in radians) between two vectors.

When rotating a series of vectors through the same angle, it is much more efficient to first calculate the rotation matrix and pass this to `rotate` rather than calling `rotate` with the Euler angles.

The `spherical` type is simply

```
struct spherical {
  double r,theta,phi;
  spherical(double r,double theta,double phi =0);
};
```

`spherical` and `vector3` are converted simply by creating (casting) one type from the other e.g. `spherical sphco(myvector);` .

Modified

8.4 `timer`

`timer` (declared in `timer.h`) is a simple stopwatch. The timer starts from the construction of the object, is read out (in seconds) using the `()` operator and can reset by the `reset()` member function. e.g.

```
timer stopwatch;                              create and start stopwatch
...                                           code to time
cout << "That took " << stopwatch() << " seconds";
```

The closely related `wtimer` object displays elapsed "wall clock" time rather than CPU time.

8.5 Parallel computation with MPI

Currently undocumented: see `test/testMPI.cc` *for an example.*

8.6 Parameter input

`ttyio.h` declares a collection of functions intended to simplify the input of program parameters.

`int getint(char*` *prompt*`)` displays the prompt string and reads the (newline) terminated input from the user. If the resulting string cannot be read as an integer, the user is reprompted. Note that the integer parsing stops at the first non-digit character i.e. `"5.9"` and `"5 hundred"` will both be read, without complaint, as 5.

`int getint(char*` *prompt*`,int` *default*`)` as simple `getint`, but the *default* value is returned if the user returns an empty string (i.e. presses Return).

`int getint(char*` *prompt*`,int` *min*`,int` *max*`)` as simple `getint`, but the response is only accepted if it lies between *min* and *max* (inclusive). An `InvalidParameter` exception is thrown if *max*<*min*.

`int getint(char*` *prompt*`,int` *min*`,int` *max*`,int` *default*`)` combines a default response with a range. An `InvalidParameter` exception is thrown if *default* is not in the range *min* to *max*.

`float getfloat(char*` *prompt* `[,int` *min*`,int` *max* [,float *default*]`)]` behave as `getint` but return floating point numbers. `"5.9"` and `"5 hundred"` will be returned as 5.9 and 5 respectively.

`bool getlogical(char* prompt)` returns `false` or `true` depending on whether the response to the prompt was positive (1, or beginning with `"y"`), or negative (0, or beginning with `"n"`).

`int getlogical(char* prompt,int default)` as above, but with default (non-zero default value means default result is 1).

`int getoption(char* prompt,char* options)` requires the response to start with one of the letters of the *options* string. The result is the index of the response in the option string i.e. 0 for first letter etc. This numbering of the options is consistent with `enum` which, by default, numbers its members from 0. As an example:

```
enum output_type { ASCII, BINARY, MATLAB };
printf("A - ASCII\nB - Binary\nM- Matlab\n");
output_type format=output_type(getoption("Output format? ","ABM"));
```

which prompts the user to specify an output format after printing a menu of possible responses with the character for each response clearly indicated.

`int getoption(char* prompt,char* options,int default)` as above, but with default response.

`void getstring(char* prompt,char* dest,int max,char* default =NULL)` prompts the user for a string which is returned in *dest*. *max* specifies the maximum number of characters in *dest*. The *default* string is returned if the pointer is non-null and the user presses return. Note that an empty string (or one consisting entirely of white space) is a valid response.

Responding to a large number of questions each time the program is run is tedious, and so it is useful to be able to specify the responses in the command line. The functions used to implement this are identical to those above, with the addition of three arguments e.g.

`getint(int argc,char* argv[],int& count,char* prompt)` tries to read an integer from `argv[`*count*`]` if *count*<`argc`. If this is not the case, the user is prompted for the response. *count* is an argument counter which should initially be set to 1 and is incremented as arguments are parsed.

For example, `myprogram 3 - no output B` gives the responses to the first five `get` questions. `"-"` is used to denote the default response. The user will be asked the remaining questions, if more than five responses are needed. If a command line response cannot be parsed, the program exits with the non-zero error code `BADINPUT`. Similarly, if a file given in the command line cannot be opened, the program aborts with an `OPENFAILED` error condition. Each response read from the command line is printed (together with its prompt) on the standard output. Hence the output contains the information necessary to re-run the calculation.

A program will typically start

```
int main(int argc,const char* argv[])
{
    int count=1;                              Initialise counter
    int nspins=getint(argc,argv,count,"Number of spins?  ",2);
    ...                                       Get number of spins (default 2).
```

These routines are not particularly elegant, but are simple to use. Consider using existing public domain routines if something more sophisticated is required.
Miscellaneous handy functions

`bool isreadable(const char*` *fname*`)` returns `true` if the given file is readable, `false` otherwise. This does not imply that the user necessarily is able to *write* to the file; a directory would give a positive response.

`const char* getbasename(const char*` *fname*`)` returns (a pointer to) the "leafname" of a filename. Hence `getbasename(argv[0])` will returns the name of the program even if is it being run via a command such as `"./program"`.

9 Evolution

Changes that may involve modifications to user code are highlighted.

9.1 Changes from release 2

- The most significant change has been changing the storage of matrix objects for C-like pointer-to-pointer to a "flat" FORTRAN-like vector. The improves the efficiency of operations on larger matrices since the cache-misses on the pointer index table are avoided. It also reduces the overhead for allocation and de-allocation of matrices, so there is no significant *efficiency* advantage in using lists to store one-dimensional data. This change should not be visible to user programs. *The balance is less clear for smaller matrices; some operations could run more slowly.*

- Support for multidimensional matrices (`MultiMatrix`).

- The mathematical machinery has been rewritten in a cleaner fashion, making it easier to re-use code for existing types to new forms of e.g. matrices, vectors. The downside is an increased number of "layers" making *compilation* somewhat slower and some very obscure error messages if things go wrong in compilation or at run time. When faced by a screed of incomprehensible error messages, you should still be able to identify which line of your program triggered the problem. Then look for some form of logic error e.g. trying to copy a vector into a matrix, or a floating point matrix into an integer matrix etc.

- Some interfacing to external numerical analysis libraries (e.g. ATLAS, ACML) is provided. These routines can give major efficiency improvements for operations such as matrix multiplication on larger matrices. The "glue" defined in `cmatrix_external.h`

ought to be useful for applying other BLAS/LAPACK-style functions to `libcmatrix` matrices.

- The memory allocation for `List` and `Matrix` objects has been modified to make it possible to create containers for objects without default constructors (as in the Standard Library). Some changes to the `ListList` interface have been made as a result. The `ExplicitList` type allows mixed lists to be constructed, removing the need for some objects to have default constructors. Some objects, e.g. `spin_system` must now have arguments supplied when constructed.

- Function naming is aligning on Standard Library usage e.g. containers should be emptied by calling the `clear` member function (rather than `kill`), `empty` should be used in preference to test whether container objects are empty etc.

- The move towards function objects has been reinforced by the virtual elimination of pointers to functions (in `optim.h` and `cmatrix_threads.h`). Replacement by function objects has removed the need for inelegant pointers-to-void for supplementary information.

- The `read_simpson` and `write_simpson` allow the reading and writing of SIMPSON format files e.g. for display in `simplot`. The `coherencematrix` functions provide some functionality present in `SIMPSON` which was missing in `libcmatrix`. The Matlab V5 file format is now well supported.

- Additional support for Liouville space calculations and some support for Floquet techniques.

- The propagation code has been revised and extended. `xxxPropagator` objects replace `StepxxxPropagator` objects. Note that the former, as the name suggests, return the propagator for a specified time interval, *replacing* the destination argument rather than "propagating" it.

- "Higher order" functionality for defining pulse sequences, working with periodic systems, block-diagonal problems etc.

9.2 The future

The core functionality of the library works nicely and seems to fill its goal of combining the efficiency of a compiled language, the simplicity of *Matlab* programming, and the power of C++. Some types and functions that have not proved so useful have been removed in this release and this kind of experimentation followed by pruning is likely to continue at the "fringes".

The support for Liouville space calculations is rather limited (especially in comparison with GAMMA). Although a lot can be done with the existing types and functions, this is certainly an area for future work. Another interesting addition would be to package an existing (C) library for sparse matrices in terms of `libcmatrix` data types. This would be especially useful for Louiville space calculations. Otherwise there is little incentive to add further basic

+/-	0	1	2	n
0	0	1	2	n
1	1	1	2*	
2	2	2*	2	
n	n			n

Table 7: Dimensionality of results of addition or subtraction operations as a function of dimensionality of arguments. Blank entry indicates incompatible combination. *one-dimensional argument is interpreted as a matrix diagonal.

functionality e.g. additional numerical analysis routines, standard algorithms, since these can increasingly be supplied via external libraries and routines.

A The algebra

A major change in the current release is the "abstraction" of basic mathematical operations into a general algebra defined in `basedefs.h`. Rather than define operations on individual data types, such as `Matrix<T>`, the different data types are tagged with information, such as the dimensionality of the object, and an overall set of rules are applied to determine what function should be applied. This makes it easy to create new data types, such as "view" objects, with well-defined mathematical proporties; it would be very messy to create explicitly mathematical operations for each new data type and all possible combinations of data types.

It is important to realise that "rules" of the algebra are interpreted at *compile* time based on the properties of the data types involved. This means that there is no run-time overhead for this abstraction, but does mean that there is no mechanism for incorporating helpful error messages when things go wrong. The process relies heavily on templating and inlining, which lengthens *compile* times and implies that programs compiled without any optimisation (e.g. for debugging) will suffer from increased overheads.

Table 7 shows which object dimensionalities are compatible for addition and subtraction operations. For instance, addition of a scalar (dimensionality zero) is always valid, with the output having the same dimensions as the input. Other operations require the inputs to have the structure e.g. two matrices with the same shape (otherwise a `Mismatch` exception is thrown). Provided there is no ambiguity, one-dimensional objects can be interpreted as simple vectors (without orientation) or matrix diagonals. Adding a 1D object to a 2D object will create a 2D result. A `NotSquare` exception is thrown if the 2D input is not square and a `Mismatch` exception is thrown if the length of the vector does not match the order of the 2D object.

The same rules apply to in-place operations, `+=` etc., new-object and supplied destination forms, with the exception of that if the "target" matrix of the in-place function is undefined, it is treated as a zero matrix of appropriate dimensions. An undefined input for the binary additions generates an `Undefined` exception.

Table 7 shows the corresponding table for multiplication. Note the only only "scaling" operations are currently defined for multidimensional operations. Multiplication of a matrix

*	0	1	2	n
0	0	1	2	n
1	1	1	1/2*	
2	2	1/2*	2	
n	n			

Table 8: Dimensionality of results of multiplication as a function of dimensionality of arguments. *Interpretation of 1D argument depends on dimensionality of output: vector output implies input must be vector, matrix output implies it is diagonal matrix.

n(A)	n(B)	n(C)
n	0	n
1	1	1
2	0	1*

Table 9: Permitted dimensionality for `mla(A,B,C)` operation. *One-dimensional argument interpreted as matrix diagonal.

by a 1D object is unusual in the nature of the operation is deduced from the "output" e.g. for in-place operations and supplied-destination with matrix output, it is multiplication by a diagonal matrix. The output from the new-object form is undefined! An undefined "target" matrix to in-place multiplication is treated as an identity matrix.

Other operations, including division, have simpler rules: either the input arguments have exactly the same structure, in which case the operation is applied sequentially to corresponding elements, or one argument is a scalar, in which case the output has the same structure as the other argument and it formed by combining the constant argument with each element of the input. N.B. it is generally more efficient to write a scaling division in terms of multiplication i.e. `A*=1/x` rather than `A/=x`, but this is left to the user.

Points to watch

- When using supplied-destination functions, it is obviously necessary to ensure that the output has the correct type for the input arguments, otherwise the compilation will fail (somewhat obscurely).
- New-object functions create the following objects depending on the dimensionality of the output: `T` (0), `List<T>` (1), `Matrix<T>` (2), `MultiMatrix<T,`n`>` (n).

B Speed comparison of different operations

Table 10 shows the result of time trialing different operations on 8×8 complex matrices. Tests X–XI involve block matrices; `A` and `B` were each blocked into two 4×4 matrices. Tests XV–XVII involved `List<complex>` objects with an equivalent number of items (i.e. 64).

Points to note:

test	operation	coding	PC
I	matrix add	`A=B+C`	2.0
II	copy and add	`A=B; A+=C`	3.0
III	supplied-dest. add	`add(A,B,C)`	0.9
IV	matrix multiply	`A=B*C`	22
V	supplied-dest. multiply	`multiply(A,B,C)`	25
VI	multiply and add	`A+=2.0*C`	4.4
VII	multiply and add	`mla(A,double(2),C)`	1.5
VIII	multiply and add	`mla(A,complex(2),C)`	2.4
X	block copy and add	`BA=BB; BA+=BC`	1.9
XI	explicit block add	`add(BA(0),BB(0),BC(0)); add(BA(1),BB(1),BC(1))`	0.7
XII	similarity transform	`unitary_simtrans(A,B,C)`	55
XIII	inverse sim. trans.	`unitary_isimtrans(A,B,C)`	50
XIV	block sim. trans.	`unitary_simtrans(BA,BB,BC)`	25
XV	list add	`LA=LB+LC`	3.4
XVI	list copy and add	`LA=LB; LA+=LC`	1.8
XVII	supplied-dest. list add	`add(LA,LB,LC)`	0.9

Table 10: Timings (in us) of different implementations of basic operations. Machine used 433 MHz Celeron-based PC (cygwin/g++), `timetrial 150`, `timetriall 100 64`.

- Comparing the equivalent matrix additions, I, II and III, the operation involving a temporary matrix (I) is slightly more inefficient than the corresponding copy followed by in-place addition (II). The supplied destination form, III, is much faster.
- The elimination of temporary variables is much less significant for time consuming operations, such as matrix multiplication, IV vs. V. It is worth remembering, however, that the allocation of memory for square `cmatrix` is optimised, and where temporary objects are allocated from free store, the differences will be more significant.
- The difference between VI and VII is again the elimination of a temporary. Using a complex scaling factor (VIII vs. VII) approximately doubles the time required i.e. overheads are relatively small.
- Despite the reduced number of operations (32 vs. 64), the block addition, X, is slightly slower than the ordinary matrix addition, II. The explicit addition, XI, keeps overhead to a minimum and is slightly faster than III.
- The advantage of blocking is much clearer with more demanding operations e.g. similarity transformation, XVII vs. XV.
- XV–XVII are operations on lists with the same number of elements (64). The memory allocation is not optimised for `List` objects and so the performance of operations involving temporaries (XV) is noticably worse.

rank	external	internal	naïve
2	14	85	86
4	84	140	149
8	185	189	178
16	282	217	200
32	270	195	170
64	248	173	161
128	227	151	54
256	213	115	46
512	223	114	46

Table 11: Performance (in Mflops) of complex matrix multiplication as a function of matrix size for three algorithms: external (ATLAS), normal "internal" `libcmatrix` method, naïve implementation of matrix multiplication. From `testops 200 M y EIB n y y 2 512` on 433 MHz Celeron-based PC.

Optimising performance is not a simple matter of minimising the number of instructions required for a given operation. Particularly for large problems, the efficiency is determined by the retrieval of data from memory, while "overhead" becomes proportionately more significant for small problems. This is illustrated in Table 11 for matrix multiplication. The overall peaks for moderately sized matrices (rank 16). As the matrices becomes larger than the processor cache performance drops off sharply for the naïve implementation of matrix mutiplication. The more sophisticated, but generic, `libcmatrix` routine maintains a much better level of performance, while the processor-optimised ATLAS routine maintains a good efficiency except for small matrices where its performance is quite poor. Since small matrices need to handled efficiently too, `libcmatrix` switches between "external" and "internal" routines depending on the matrix size. This switch point for different algorithms can be modified before compilation. As a result, it is better *not* to compile with ATLAS support if dealing exclusively with small problems (Hilbert space $<\sim 10$).

C Errors and exceptions

In general, the `libcmatrix` library uses exceptions to indicate error conditions. The exception classes are all derived from the `MatrixException` class which can thus be used to catch any error arising from the library. The full list of exceptions is shown in Table 12. The `MatrixException` type include a character string which is used to describe the particular error detected, although this will often only give the name of the function where the error was first thrown. `std::cerr <<` *exception* can be used to print this message on the standard error output. Alternatively, the exceptions can be caught through the standard classes defined in `<stdexcept>` (where the member function `what()` returns the error description).

The throwing of an exception generally corresponds to a programming error e.g. attempting to add together matrices with different sizes. Under these circumstances it is best to abort the calculation with an exception rather than quietly return an error code. For some functions,

Exception	From	Why?
`ArgumentClash`	`invalid_argument`	Input argument cannot be used to store output.
`BadIndex`	`out_of_range`	Index out of range.
`BadRank`	`invalid_argument`	Attempt to access inactive or impossible tensor rank.
`Failed`	`runtime_error`	Algorithm failure due to unsuitable data e.g. singular matrix
`InternalError`	`logic_error`	Failure of internal consistency check. By definition, this indicates a bug in the library.
`InvalidParameter`	`invalid_argument`	Parameter value passed to function is not valid.
`Mismatch`	`domain_error`	Attempt to perform operation on matrices/lists with incompatible dimensions.
`NotSquare`	`domain_error`	Attempt to perform operation only defined for square matrices.
`Undefined`	`domain_error`	Attempt to perform operation on empty/uninitialised matrix.

Table 12: The exceptions generated by `libcmatrix`. These are all derived from the class `MatrixException` and from one of the standard exception classes defined in `<stdexcept>`. Note that the `BadIndex` exception will generally only be thrown if debugging has been enabled cf. Sec. 4.3

however, the input is effectively supplied directly by the user e.g. names for input files etc. In this case, it is more appropriate to return an error number if the operation fails rather than aborting or requiring the programmer to have set up a `catch` handler for whenever the user mistypes a filename.

The usual convention is used for error codes i.e. a zero return value indicates a successful operation. Error descriptions can be printed with the functions:

`const char* error_name(int` *errval*`)` returns a pointer to an error string. Note that an `InvalidParameter` *exception* is thrown if the error number does not exist!

`int error_filter(int` *errval*`)` is a convenience function which, given a function return value, prints the error string (if any) and passes through the return value.

D Multi-threading

`libcmatrix` can be used in shared-memory parallel applications via multi-threading ("pthreads"). Although light-weight, such approaches do not scale easily and additional precautions are required to ensure the executable code can be shared. `libcmatrix` now includes support for distributed parallel programming via the MPI library. This is generally simpler to use and should be used in preference to multi-threading.

The usual `libcmatrix` library is not entirely "safe" for use with multi-threaded applications. It is essential, therefore, to compile the entire programs with the correct options, in particular the symbol `_REENTRANT` must be defined, *and* to link with `libcmatrix_r.a`, which is the thread-safe version of the library.

In many situations, "parallelising" a program a simply a matter of splitting the main computational loop between processors. In the context of solid state NMR, powder averaging is the obvious candidate for such a division. `libcmatrix` contains some simple objects, declared

in the header file `"cmatrix_threads.h"`, which permit the multi-threading to be used in a reasonably straightforward manner for breaking up loops between multiple processors. This is, of course, only possible if the cycles of the loop are strictly independent, and is only beneficial if the loop involves many cycles and is relatively time-consuming.

The key data type is the `thread_controller` object. When this is created, a specified number of threads of execution are created. These execute a "work object" whose job is to process a given section of the loop. This must be an object (derived by `BaseThreadFunction` whose `()` operator takes the form `void` *obj*`(size_t` *start*`,size_t` *end*`,size_t` *nthread*`)`. *start* and *end* specify the section of the loop to be executed, *nthread* is an index running from 0 to $N-1$ which identifies the thread. This index can be used to indicate where in an output matrix the particular thread should write to. The per-thread result matrices would be added together once the calculation is finished. This is significantly more effective than using a single result matrix, which would need to be locked each time a thread needed to access it. Modified

The `ThreadFunction` type can be used to create such an object from a traditional pointer to a function of the type `void` *func*`(`*start*`,`*end*`,`*nthread*`)`.

The threads are only destroyed when the `thread_controller` object is destroyed and can readily be "re-used".

`thread_controller(`N`)` creates n threads.

`run(BaseThreadFunction&` *obj*`,size_t` *steps*`,size_t` *chunk*`)` is used to run the multi-threaded calculation. *obj* is the function object to be called, *steps* gives the number of steps in the loop and *chunk* the number of steps allocated to a thread per calculation step, and *ptr* is an optional generic pointer. `chunk` need not divide *steps* evenly and should be chosen so that each calculation "chunk" takes a reasonable length of CPU time (say half a minute). If the chunk size is too small, time is wasted in the synchronisation required each time a new chunk is handed out (this overhead is very small, however, compared to than, say, launching a new thread). If the number of chunks is too small, the calculation risks being blocked by a particularly slow-running thread.

`start(`*obj*`,`*steps*`,`*chunk*`)` allows the main program to continue while the threaded calculation is run. The `wait()` function must be called before any attempt is made to use the results of the calculation to make sure that the calculation has finished. Contrast with `run` where the main program "blocks" until the multi-threaded calculation is finished.

`size_t get_num_threads()` returns the number of *active* threads, N, or 0 if the calculation if not active[24].

`size_t get_max_threads()` returns the number of threads, N.

`bool in_parallel()` returns *true* if inside a parallel section.

`int get_thread_num()` returns the thread number, 0 to $N-1$, or -1 if the thread is not a "worker" thread.

[24] In OpenMP, the active threads includes the "master" thread, so the return value outside a parallel section is 1

It is, of course, vitally important that the "work function" is properly re-entrant. To ensure this, it is important to understand which objects can be safely shared between threads and which not.

`Read/write safe` objects can be shared between threads, including for write access i.e. write operations are designed to prevent problems from simultaneous writes. No user-visible `libcmatrix` objects fall into this category. Note that memory allocation is generally shared between threads and so forces the use of locks. It is therefore doubly important to avoid dynamic memory allocation within time critical sections.

`Read safe` objects can be shared for reading (but not writing). This applies to all objects unless indicated. Some objects may cache values (look for `mutable` variables), but these are protected *in their thread-safe versions*.

`Unsafe` objects should never be shared across threads. Typically such objects preserve "state" from one call to the next e.g. iterators, and it wouldn't normally make sense to try to share such objects across threads. Other objects in this category are: the propagation objects (`StaticSpectrum`, `MetaSpectrum`, `PeriodicFID`, etc.), some `Propagator` objects e.g. `SequencePropagator` and the "powder" objects, `PlanarZCW` etc.

Stick to these rules and `libcmatrix` programs should run close to the processor limit i.e. the speed should increase almost linearly with the number of processors.