Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Magnetic Resonance 162 (2003) 269–283
JMR Journal of Magnetic Resonance
www.elsevier.com/locate/jmr

BlochLib: a fast NMR C++ tool kit

Wyndham B. Blanton*

Materials Sciences Division, Lawrence Berkeley National Laboratory, University of California, Mail stop 66-208, 1 Cyclotron Rd., Berkeley, CA 94720, USA
Department of Chemistry, University of California, Berkeley, CA 94720, USA

Received 27 September 2002; revised 10 January 2003

Abstract

Computational power, speed, and algorithmic complexity are increasing at a continuing rate. As a result, scientific simulations continue to investigate more and more complex systems. Nuclear magnetic resonance (NMR) is no exception. NMR theory and language is extremely well developed, that simulations have become a standard by which experiments are measured. Nowadays, complex computations can be performed on normal workstations and workstation clusters. Basic numerical operations have also become extremely optimized and new computer language paradigms have become implemented. Currently there exists no complete NMR tool kit which uses these newer techniques. This paper describes such a tool kit, *BlochLib*. *BlochLib* is designed to be the next generation of NMR simulation packages; however, the basic techniques implemented are applicable to almost any problem. *BlochLib* enables the user to simulate almost any NMR idea both experimental or theoretical in nature. Both classical and quantum mechanical techniques are included and demonstrated, as well as several powerful user interface tools. The total tool kit and documentation can be found at http://waugh.cchem.berkeley.edu/blochlib/.

1. Introduction

Simulations in nuclear magnetic resonance (NMR) are often essential towards the development and understanding of the physical nature of any NMR experiment. The theory for NMR is so well established [1] that if a simulation does not agree with the experiment, typically, the experiment has an error and not the simulation. As a result there is a large amount of feedback between simulation and experiment. For example, in a recent paper [2] a unique theory was presented to separate the anisotropic dipolar couplings from the isotropic couplings via control of the average Hamiltonians over a series of experiments. There are numerous sub-pulse sequences that could have been used to generate the desired average Hamiltonians. Experimentally, however, there are fewer able to adhere to the hardware limitations of the NMR spectrometer. Simulations were invaluable as a method to determine both working pulse sequences and the best ones for experiment.

These types of simulations, designated experimental evolutions (EEs) (see Fig. 1a), are some of the simplest to construct and generalize. Programs such as *Simpson* [3] have generalized these forms of simulations into a simple working structure not unlike programming a spectrometer itself. EEs typically require only a few algorithms to solve the dynamics of the systems, the rest of the program is simply a user interface to input experimental parameters (e.g., pulse sequences, rotor angles, etc.). EEs are essential to understand or discover any anomalies in experimentally observed data. Another common usage of EEs is to give the experimenter a working picture of what to expect from the experiment. Surprisingly, there are very few complete NMR EE packages. In fact, up until this tool kit, *Simpson* seems to be the only one publicly available.

The other class of simulation, designated theoretical evolutions (TEs) (see Fig. 1b), are used to explore theoretical frameworks and theoretical modeling. Of course there can be much overlap between the EEs and TEs, but the basic tenet of a TE simulation is they are a

* Fax: 1-510-486-5744.
E-mail address: magneto@dirac.cchem.berkeley.edu.

1090-7807/03/$ - see front matter © 2003 Published by Elsevier Science (USA).
doi:10.1016/S1090-7807(03)00035-1

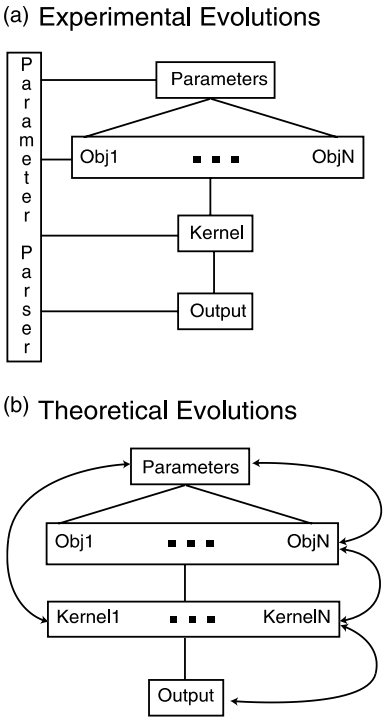

Fig. 1. Many basic simulations can be divided into two main subgroups: (a) experimental evolutions (EEs), and (b) theoretical evolutions (TEs). The main object of both is some sort of generated data and both typically require some input of parameters. EEs tend to use a solid Kernel driver, whereas the TEs can use many different Kernels, feedback upon itself, use the generated data in other kernel, and so forth. For this reason EEs can be developed to a large degree of generality on the input parameters (e.g., various different types of pulse sequences). Their main function is a parameter parser where much attention is given to the interface. TEs, on the other hand, are usually complex in the kernels and transparent interfaces are not necessarily their primary goal.

designed to explore the physical properties of the system, even those not assessable by experiments, to develop an understanding and intuition about the systems involved. Simulations of singular interactions (e.g., including only radiation damping in a spin system) to see their effect is one such example. Development of a master TE program proves near impossible simply because of the magnitude of different methods and ideas used to explore an arbitrary model. The best one can do today is to create a tool kit that provides the most common algorithms, structures, and ideas used for the theoretical modeling. These tool kits should be a simple starting places for more complex ideas (see Fig. 1). A good overview of the methods desired in NMR can be found in [4].

Currently there is only one such package available to the NMR spectroscopists, Gamma [5]. Gamma made an initial leap forward in terms of definitions of many of the necessary ideas to stimulate TEs. The main focus of Gamma is liquid state NMR (the solid state practicalities are becoming developed in later versions). However, NMR experimentation is evolving past the basic high field liquid experiment. Complex interactions like the demagnetizing field and radiation damping are becoming important and are best treated classically (see [6] and references there in). Solid state NMR (SSNMR) is being used more frequently and with better and better resolution and techniques. Ex situ NMR is a new branch currently under exploration [7–9] requiring detailed knowledge of magnetic fields in the sample. Low field experiments (see [10] and references there in) are also becoming more common. Pulse shaping [11] and multiple rotor angle liquid crystal experiments [12] are also becoming more frequent. Not only have new developments been encountered in NMR, but relatively new developments in computer science and computer power have recently opened new avenues towards numerical computations.

The need for code efficiency in numerical computation is self evident. However, to create fully optimized numerical code involves a deep understanding of the microprocessor architecture, instructions sets, and general memory handling. But the time it takes to write machine code in a fully optimized form is daunting, thus programming in another programming language is preferable for complex projects. Several computer programming languages exist, each with theirs own benefits and challenges. The easiest language to use for almost any task is a high level language (scripting languages which run non-compiled code). Matlab (Mathworks), Mathematica (Wolfram), Python, and Tcl/Tk offer a wide variety of numerical algorithms, complex data types, and easier syntax. Typically, however, whatever is gained in ease of use, is lost in either speed or extendibility. The interfaces and numerics can be built for these scripting languages in a lower level language (usually C), but building the interfaces for types of languages tends to make the lower level language harder to code and manage.

The most common language used for numerical speed is Fortran. The compiler is fast, but it is still up to the programmer to create the optimized code. The basic problem with the Fortran language is its syntax. Functions, the basic drivers of any program, are very hard to read, creating complex data structures is next to impossible, and separating the user interface from the numerics is also very difficult. The next choice for a language/compiler is C. It extends Fortrans ability for creating complex data types and character and string handling. However, to obtain highly optimized code, one must still write it out by hand as in Fortran (see Fig. 2).

C++ allows the same basic syntax as C but invokes the idea of objects, object oriented programming (OOP), inheritance and templates [13,14]. These additions alter the way one thinks about programming in general, especially in scientific programming. The ob-

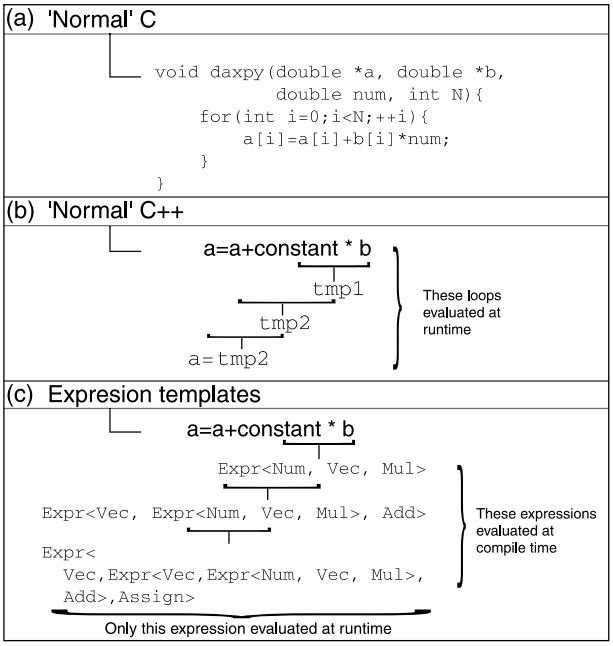

Fig. 2. A comparison of the various methods to write the DAXPY (Double precision A times X Plus Y) algorithm where (a) shows how to program the algorithm in C using the simplest method. (b) shows how a non-expression template C++ vector object code would interpreted by the compiler, and (c) shows how an expression template C++ vector object code would be interpreted by the compiler. The unrolling of (c) is performed at compilation time, not at runtime, thus increasing the speed by approximately a factor of 3.

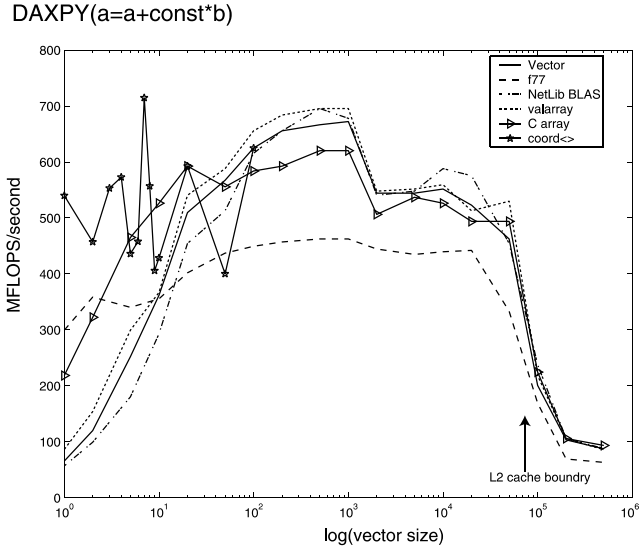

Fig. 3. This figure shows speed test for the common vector operation $a = a + b * \ constant$ (daxpy: Double precision A times X Plus Y) for doubles in Millions of Floating point operations (MFLOPS) per second performed on a 700 MHz Pentium III Xeon processor running Linux (Redhat 7.2). Each code sample was compiled using the GNU compiler (gcc) using the same optimizations (-O3-finline-functions-funrool-loops). Several different data types were tested. The most common data types, C and Fortran, show faster initial speeds because there is no overhead in the algorithm (no optimization see Fig. 2a). The remaining data are for the optimized algorithms. The Netlib F77 daxpy algorithm was taken from the Netlib website (http://netlib.org/). The `valarray` type is available for some C++ standard libraries and is an expression template vector class. The `coord<>` object is in *BlochLib* and is defined as a fixed size vector, thus its speed can be greatly improved due to loop unrolling. The `Vector` data are for the *BlochLib* vector object. As the figure shows, the *BlochLib* data containers are comparable in speed to those of the highly optimized `valarray` and Netlib versions. The L2 cache (2 Mb) boundary is visible for the Xeon processor.

ject is not only a complex data type (a set of other data types) but also defines a set of allowed operations and functions that the object can perform. The object (referred to in C++ as a *class*) in scientific programming should embody the total ability of the theoretical idea. For example a vector consists of an array numbers. A vector has specific rules about addition, multiplication, etc. that are easily contained in the objects definition. Unlike C or Fortran, operators (like '+', '–', '*', etc.) can be defined specifically for these objects making code readability and program interfaces much easier to use. Users can simply write the code as one thinks of the problem analytically (e.g., $a = a + b * 3$), rather then as a function (e.g., daxpy$(a, b, 3)$, as in Fig. 2). Given M data types, and N functions, templates can in principle reduce the $O(M \times N)$ number of procedures in a Fortran environment to $O(N + M)$ procedures in a C++ environment.

C++ has typically been avoided for scientific programming because of certain speed issues using the operator formalism [4,15]. However, a relatively new technique called *expression templates* [16,17] removes this older problem. Fig. 2c shows how the compiler treats a expression template object. As Fig. 3 shows, the speed in MFLOPS (millions of floating point operations per second) of a standard vector operation is comparable to that of the hand coded version and the Fortran versions. If one knows the length of the vector (in Fig. 3 called `coord<>`) then we can effectively unroll the loops to allow for machine instruction optimizations using *meta programming* techniques [18–20].

In this paper we wish to present *BlochLib* designed to be the next generation NMR simulation tool kit. It incorporates modern computational techniques, data structures and algorithms available. Because the tool kit is quite large (the documentation alone is over 1000 pages), this paper will serve as an introduction to the techniques used as well as a general overview of library itself. The following section will discuss some of the computational techniques and generic classes of NMR simulations that drive the basic design of the *BlochLib*. Following the design overview, several example programs will be discussed. They will attempt to demonstrate both the generality of the library as well as how to set up a basic program flow from parameter inputs to data output.

2. Design

The design of a given tool kit relies heavily on the objectives one wishes to accomplish. These objectives then determine the implementation (code language, code structure, etc.). The key objectives for *BlochLib* are, in order of importance, speed, ease of use, the incorporation of existing numerical techniques and implementations, and the ability to easily create both TEs and EEs for NMR in almost any circumstance. Below, several issues are addressed before the major design of *BlochLib* is discussed.

2.1. Existing numerical tool kits

For the quantum mechanical aspects of NMR, the basic operation is matrix multiplication. The same expression template methodology can also by applied to matrices. However, there are certain matrix operations that will always require the use of a temporary matrix. Matrix multiplication is one such operation. One cannot unroll these operations because an evaluated point depends on more then one element in the input. So the task becomes one of optimizing a matrix–matrix multiplication. This task is not simple; in fact it is probably one of the more complex operations to optimize because it depends dramatically on the systems architecture. A tool kit called ATLAS (automatically tuned linear algebra software) [21] performs these optimizations.

The introduction of the fast Fourier transform made possible another class of simulations. Since that time several fast algorithms have been developed and implemented in a very efficient way. The fastest Fourier transform in the west (FFTW) [22] is one of the best libraries for the FFT.

Another relatively recent development in scientific simulations is the movement away from supercomputers to workstation clusters. To use both of them effectively one needs to know how to program in parallel. The message passing interface (MPI) [23] provides a generic interface for parallel programming.

Most any scientific endeavor eventually will have to perform data fitting of experimental data to theoretical models. Data fitting is usually a minimization process (usually minimizing a χ^2 function). There are many different types of minimization routines and implementations. One used fairly frequently for its power, speed, multiple types of algorithms is the CERN package MINUIT [24].

2.2. Experimental and theoretical evolutions for NMR simulations

As stated above TEs tend to require more possible configurations then an EE program. EEs tend to be heavily parameter based using a main driver kernel, while a TEs are basically open ended in both parameters and kernels (a better assumption about a TE simulation is that one cannot really make any assumptions). Fig. 1 shows a rough diagram of an NMR simulation for both types (of course it can be applied to many simulation types).

EEs are easily parsed into four basic sections: *Parameters*, *Parameter parser*, *Main Kernel*, and *Data Output*. The *Parameters* define a program's input, the *Parameter parser* decided what to do with the parameters, the *Main Kernel* performs the desired computation, and the *Data Output* decides what to do with any generated data. *BlochLib* is designed to make the *Parameters*, *Main Kernel* and *Data Output* relatively simple for any NMR simulation. The *Parameter Parser* tends to be the majority of programming an EE. *BlochLib* also has several helper objects to aid in the creation of the parser. The objects `Parameters`, `Parser` and `ScriptParse` are designed to be extended. They serve as a base for EE design. With these extendable objects almost any complex input state can be treated with minimal programming effort.

The *Main Kernel* drivers need to be able to handle two distinct classes of NMR simulation the quantum mechanical and the classical. The quantum mechanical aspect involves solving the Heisenberg form of the Schrödinger equation

$$\frac{\mathrm{d}U}{\mathrm{d}t} = \frac{-\mathrm{i}}{\hbar} H(t) U. \tag{1}$$

The propagator, U, and Hamiltonians, $H(t)$, are matrices with the solution to Eq. (1) given by

$$U = T \exp\left[-\mathrm{i} \int H(t)\,\mathrm{d}t \right], \tag{2}$$

where T is the Dyson time ordering operator. This evolves a density matrix, ρ, as

$$\rho(t) = U * \rho * U^{\dagger}. \tag{3}$$

Thus the two most important algorithms for the kernel are the matrix exponentiation and the matrix multiplication. In our simulation framework, we can only approximate the integral in Eq. (2) as

$$U = \prod_{j=t_1}^{t_2} \exp\left[-\mathrm{i} H(t_j) \Delta t_j \right], \tag{4}$$

where Δt_i is a small time step, where small means approximately an order of magnitude less then the inverse of the largest eigenvalue of the Hamiltonian or the inverse of the rate of variation of the Hamiltonian, which ever is smaller. Several approximations can be made if $H(t)$ is not a function of time

$$U(t) = \exp(-\mathrm{i} H * t) \tag{5}$$

or if the Hamiltonian is periodic.

Table 1
Numerical ordinary differential equation solver algorithms available in *BlochLib*

Algorithm	Class name	Engine_T functions[a]
Cash–Karp–Runge–Kutta fifth order method	`ckrk`	Function
Bulirsch–Stoer extrapolation method	`bs`	Function
Semi-implicit Bulirsch–Stoer extrapolation method	`stibs`	Function, Jacobian

[a] Functions that must be defined in the input object class (see text).

The classical aspects of NMR are treated by solving the Bloch equations,

$$\frac{d\vec{M}}{dt} = -\gamma\vec{M} \times \vec{B}(t), \tag{6}$$

where $\vec{M}$ is the magnetization and $\vec{B}(t)$ is the magnetic field at a given time. Both $\vec{M}$ and $\vec{B}(t)$ are 3-vectors in Cartesian space and the equations therefore represent ordinary differential equations (ODE). There are several algorithms to solve ODE [25]. *BlochLib* contains three different solvers (see Table 1).

With these basic ideas of a TE and EE, the basic design of *BlochLib* will be described in the next section.

2.3. BlochLib layout

BlochLib is written entirely in C++. Fig. 4 shows the basic layout of the tool kit. It is designed to be as modular as possible with each main section shown in Fig. 4 treated as separate levels of sophistication. The first levels are the main numerical and string kernels, the second levels utilize the kernels to create valid mathematical objects, the third levels uses these objects to perform complex manipulations, and the fourth levels creates a series of modules specific to NMR for both the classical and quantum sense.

It uses C++ wrappers to interface with MPI, ATLAS, FFTW, and MINUIT. *BlochLib* uses MPI to allow for programming in parallel and to pass the parallel objects to various classes to achieve a seamless implementation in either parallel or serial modes. It also allows the user to put and get the libraries basic data types (vectors of any type, matrices of any type, strings, coords of any type, vectors of coords of any type) with simple commands to any processor. The ATLAS library provides the backbone of the matrix multiplication for *BlochLib*. Fig. 5 shows you some speed tests for the basic quantum mechanical NMR propagation operations. You may notice that *BlochLib*'s speed is slower then ATLAS's even though the same code is used. The reason for this discrepancy is discussed in Section 2.4. *BlochLib* uses FFTW to perform FFTs on its vectors and matrices, and allows the usage of the MINUIT algorithms with little or no other configuration.

The containers are the basic building blocks. It is critical that the operations on these objects are as fast as possible. The optimizations of vector operations are critical to performance of classical simulations as the solving of differential equations take place on the vector level. Matrix operations are critical for quantum mechanical evolutions and integration. For this reason the `coord`, `Vector`, and `matrix` classes are all written using expression templates, with the exception of the matrix multiplication and matrix division which use the ATLAS and LU decompositions algorithms respectively. The `coord<>` object is exceptionally fast and should be used for smaller vector operations. The `coord<>` object is specifically made for 3-space representations, with specific functions like rotations and coordinate transformations which only function on a

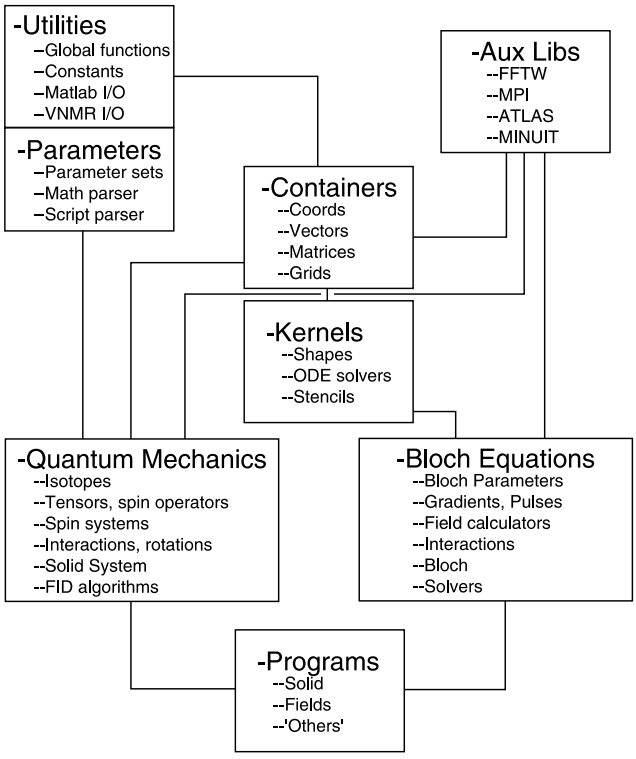

Fig. 4. The basic design layout of the *BlochLib* NMR tool kit. The *Utilities*, *Parameters*, *Aux Libs*, *Containers*, and *Kernels* sections comprise the basic beginning of the tool kit and have little to do with NMR. They form a basic generic data structure framework to perform almost any high performance scientific simulations. The *Quantum Mechanic* and *Bloch Equation* sections assemble objects that comprise the backbone for the NMR simulations. Finally the *Programs* section assembles the NMR pieces into functional programs which perform general NMR simulations (like *Solid*), calculate arbitrary fields from coil geometries, and a wide range of investigative programs on NMR systems (see Table 4).

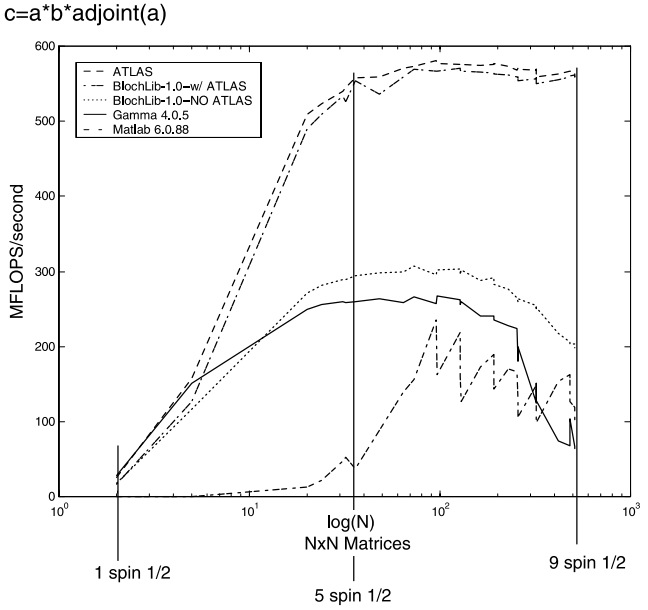

Fig. 5. This figure shows a speed test for the common NMR propagation expression $c = a * b * a^{\dagger}$ in Millions of Floating point operations (MFLOPS) per second performed on a 700 MHz Pentium III Xeon processor running Linux (Redhat 7.2). Each code sample (except for Matlab) was compiled using the GNU compiler (g++) using the same optimizations (-O3-finline-functions-funrool-loops). Both a, b, and c are full, square, complex matrices. ATLAS shows the fastest speed, but *BlochLib* using ATLAS as a base is not far behind. An existing C++ library, *Gamma*, shows normal non-optimized performance. Matlab's algorithm is slowed appreciably by this expression because the overhead on its use of temporaries is very high. It may be interesting to note that the speed of Matlab's single matrix multiply ($c = a * b$) is much better (and close to that of Gamma's) then the performance shown for ($c = a * b * a^{\dagger}$) because of this temporary problem. The matrix sizes are incremented in typical numbers of spin 1/2 particles. A '1 spin 1/2' matrix is a 2×2, a '5 spin 1/2' matrix is 32×32, and a '9 spin 1/2' matrix is 512×512.

3-space. However, any length is allowed, but as Fig. 3 shows, the `Vector` speed approaches the `coord<>` for large N, and with *much* less compilation times. The matrix class has several structural types available: Full (all elements in the matrix are stored), Hermitian (only the upper triangle of the matrix are stored), Symmetric (same as Hermitian), Tridiagonal (only the diagonal, the superdiagonal, and the subdiagonal elements are stored), Diagonal (only the diagonal is stored), and Identity (assumed ones along the diagonal). Each of these structures has specific optimized operations, however, the ATLAS matrix multiplication is only used for Full matrices. There are also a wide range of other matrix operations: LU decompositions, matrix inverses, QR decompositions, Gram-Shmidt ortho-normalization, matrix exponentials and matrix logarithms.[1] The Tridiagonal structure has an exceptionally fast LU decomposition. The `Grid` class consists of a basic grid objects and allows for creation of rectangular Cartesian grid sets.

The utilities/IO objects include several global functions that are useful string manipulation functions. These string functions power the parameter parsing capabilities of *BlochLib*. Several basic objects designed to manipulate parameters are given. The `Parser` object can be used to evaluate string input expressions. For instance if "$3 * 4/\sin(4)$" was entered, the `Parser` can evaluate this expression to be -15.85. The object can also use variables either defined globally (visible by every instance of `Parser`) or local (visible only by the specific instance of `Parser`). For examples, if a program registers a variable $x = 6$, the `Parser` object can use that variable in an expression, like "$\sin(x) * 3$", and return the correct value, -0.83. The `Parameters` object comprises the basic parameter input capabilities. Large parameter sets can be easily grouped into sections and passed between other objects in the tool kit using this object. The parameter sets can be nested (parameters sets within parameters sets) and separated. Creation of simple custom scripts can be performed using the `ScriptParse` object in conjunction with `Parser`. The `ScriptParse` object is used to define specific commands to be used in conjunction with any mathematical kernels.

Data output can be as complicated as the data input. The `Parameters` object can output and update specific parameters. Any large amount of data (like matrices and vectors) can be written to either Matlab (5 or greater) format. One can write matrices, vectors, and coords, of any type to the Matlab file, as well as read these data elements from a Matlab binary file.[2] Several visualization techniques are best handled in the native format of NMR spectrometer software, so a VNMR (Varian) reader and writer of 1D and 2D data are available. XWINNMR (Bruker) and Spinsight (Chemagnetics) data readers are also available as well as a WAVE audio format reader and writer. Any other text or binary formats can be constructed as needed using the basic containers.

The next level comprises the function objects, meaning they require some other object to function properly. The `XYZshape` objects require the `Grid` objects. These combine a set of rules that allow specific Cartesian points to be included in a set. It basically allows the construction of non-rectangular shapes within a Cartesian grid. For instance the `XYZcylinder` object will remove all the points not included in the cylinder dimensions. Similar shapes exist for slice planes and rectangles, as well as the capability to construct other shapes. The shapes themselves can be used in combination (e.g., you can easily specify a grid to contain all

[1] The algorithms used for matrix exponentials and matrix logarithms are based on those in GAMMA [5].

[2] Much of the Matlab source is based on those in GAMMA [5].

the points within a cylinder and a rectangle, using normal operators `and` (`&&`) and `or` (`||`), "`XYZcylinder && XYZrect`").

The ODE solvers require function generation objects (Table 1 lists the available ODE solver algorithms). The solvers are created as generically as possible, allowing for various data types (`double`, `float`, `complex`) and containers (Vectors, coords, matrices, and vectors of coords). The ODE solver requires another object that defines a `function`. All the algorithms require the same template arguments, `template<class Engine_T, class ElementType_T, class Container_T>`. `Engine_T` is another class which defines the function(s) required by the solver shown in Table 1 column 3. `ElementType_T` is the precision desired or another container type (it can be things like `double`, `float`, `coord<>`, `Vector<>`, etc.). The `ElementType_T` is the type inside the container, `Container_T`. For instance if `ElementType_T = double`, then `Container_T` will usually be `Vector<double>`or `coord<double, N>`. The Cash-Karp-Runge-Kutta fifth order method (the `ckrk` class) is a basic work horse of medium accuracy. It is a good first attempt for attempting to solve ODEs [25,26]. The Bulirsch–Stoer extrapolation method (the `bs` class) is of relatively high accuracy and very efficient (minimizes function calls). However, stiff equations are not handled well and it is highly sensitive to impulse type functions. The `BlochSolver` object uses the `bs` class as its default ODE solver [25,27–29]. The semi-implicit Bulirsch–Stoer extrapolation method is base on the Bulirsch–Stoer extrapolation method for solving stiff sets of equations. It uses the Jacobian of the system to handle the stiff equations by using a combination of LU decompositions and extrapolation methods [25,30]. All the methods use adaptive steps size controls for optimal performance.

Finally, the stencils perform the basic finite difference algorithms over vectors and grids. Because there is no array greater then two dimensional in *BlochLib* yet, the stencils over grid spaces (which are treated as vectors of coordinates, not 3D arrays) are treated much differently then they would be over a standard three dimensional array. Determination of nearest neighbors is a harder procedure using this vector representation, and the class `StencilPrep` attempts to determine the neighbors. This procedure then allows for an apparent 3D finite difference algorithm to be applied to the vectorized grid data. The 3D algorithms are included in this version of *BlochLib* for completeness and are slow due to the preparation step. The *N*-dimensional array and tools should be included in later versions.

At this point the tool kit is split into a classical section and a quantum section. Both sections begin with the basic isotropic information (spin, quantum numbers, gamma factors, labels, mass, momentum).

The quantum mechanical structures begin with the basic building blocks of spin dynamics: the spin and spatial tensors, spin operators, and spin systems. Much of the naming and design for the quantum mechanical parts are based on GAMMA's [5] initial ideas. Spatial tensors are explicitly written out for optimal performance. The spin operators are also generated to minimize any computational demand. There is a `Rotations` object to aid in optimal generation of rotation matrices and factors given either spherical or Cartesian spaces. After the basic tensor components are developed, *BlochLib* provides the common Hamiltonians objects: chemical shift anisotropy (CSA), dipoles, scalar couplings, and quadrupoles (see Table 2). These objects use the `Rotations` object in the Cartesian representation to generate rotated Hamiltonians. The `HamiltonianGen` object allows for string input of Hamiltonians to make arbitrary Hamiltonians or matrix forms more powerful. For example, the input strings "$45 * \pi * (Ix\_1 + Iz\_0)$" ($Ix\_1 + Iz\_0$ are the x and z spin operators for spin 1 and 0, respectively), and "$T21\_0, 1 * 56$" (where $T21\_0, 1$ is the second rank ($l = 2, m = 1$) spin tensor between spins 0 and 1), can be parsed by the `HamiltonianGen` much like the `Parser` object. The `SolidSys` object combines the basic Hamiltonians, rotations, and spin operators into a combined object which generates entire system Hamiltonians and provides easy methods for performing powder averages and rotor rotations to the system Hamiltonian. This class can be extended to any generic Hamiltonian function. In fact, using the inheritance properties of `SolidSys` is imperative for further operation of the algorithm classes `oneFID` and `compute`. The Hamiltonian functions from the `SolidSys` object, or another derivative, act as the basis for the `oneFID` object that will choose the valid FID collection method based on rotor spinning or static Hamiltonians. It uses normal eignvalue propagation for static samples and the γ-compute [31] algorithm for spinning samples. If the FID is desired over a powder, the algorithm is parallelized using a `powder` object. The `powder` object allows for easy input of powder orientation files and contains several built in powder angle generators.

For classical simulations the relevant interactions are offsets (magnetic fields), T_2 and T_1 relaxation, radiation damping, dipole–dipole interactions, bulk susceptibility, and the demagnetizing field [39] (see Table 3 for more details). These interactions comprise the basis for the classical simulations. Each interaction is treated separately from the rest, and can be either extended or used in any combination to solve the system. The grids and shapes interact directly with the Bloch parameters to creates large sets of configured spins either in gradients or rotating environment. New interactions can be added using the framework given in the library. The interactions are optimally collected using the `Interactions`

Table 2
Quantum mechanical high field hamiltonian objects and definitions

Interaction (class)	Hamiltonian[a]
CSA (`Csa`)	$\left(\omega^i_{\text{iso}} + \left(R_t \cdot \omega^i_{\text{ani}} \begin{pmatrix} \frac{\eta-1}{2} & 0 & 0 \\ 0 & \frac{-\eta+1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot R_t^\dagger\right)_{(3,3)}\right) \times I_{z,i}$
Dipole (`Dip`)	Let
	$C^{ij} = \left(R_t \cdot \omega^{ij}_{\text{ani}} \begin{pmatrix} -\frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot R_t^\dagger\right)$
	Homonuclear $\left\{C^{ij}(3,3) \times \frac{1}{\sqrt{6}}(2I_{z,i} \cdot I_{z,j} - I_{x,i} \cdot I_{x,j} - I_{y,i} \cdot I_{y,j})\right.$
	Heteronuclear $\left\{C^{ij}(3,3) \times \frac{2}{\sqrt{6}} I_{z,i} \cdot I_{z,j}\right.$
Quadrupole (`Qua`)	Let $C = \left(R_t \cdot \begin{pmatrix} \frac{\eta-1}{2} & 0 & 0 \\ 0 & \frac{-\eta+1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot R_t^\dagger\right)$
	and $\omega_Q = \frac{\omega_{\text{ani}}}{4I_i(2I_i-1)}$.
	then the first order is $\left\{\omega_Q C(3,3) \times 3I_{z,i} \cdot I_{z,i}\right.$
	and the second order[b] is proportional to $\omega_Q^2/(\gamma_i B_0)$
Scaler coupling (`J`)	Weak coupling $\left\{J^{i,j} \times I_{z,i} \cdot I_{z,j}\right.$
	Strong coupling $\left\{J^{i,j} \times (I_{z,i} \cdot I_{z,j} + I_{x,i} \cdot I_{x,j} + I_{y,i} \cdot I_{y,j})\right.$

[a] The Hamiltonians are given in terms of isotropic frequencies (ω_{iso}), anisotropic frequencies (ω_{ani}), and asymmetry (η) as represented in the principle axis system (PAS). $J^{i,j}$ is the isotopic scalar coupling constant. Each interaction object can have a relative orientation, R_{ori}, specified by three Euler angles (ψ, χ, ξ). Anisotropic and asymmetry terms can be rotated (using the `Rotations` class) via two separate Euler rotations. The first $R_{\text{pow}} = R(\phi, \theta, \gamma)$ is the powder average rotation, the second $R_{\text{rot}} = R(2\pi t * \omega_{\text{rotor}}, \beta_{\text{rotor}}, 0)$ is the rotation associated with mechanical rotation. R_t will signify the total rotation. $I_{(x,y,z),i}$ are the spin, i, associated spin operators. I_e is the identity matrix, B_0 is the magnetic field strength, γ_i is spin i's gamma factor, and I_i is spin i's quantum number.

[b] See [36] for more detailed second order Hamiltonians.

Table 3
Available classical interactions in *BlochLib* on the ith spin

Interaction	Field equation $(\vec{M}_i(t) \times \vec{B}(t))$[a]
Magnetic fields[b]	$\gamma_i M_i(t) \times (B_x, B_y, B_z)$
T_1 relaxation[c]	$\left(0, 0, \frac{M_0 - M_z(t)}{T_1}\right)$
T_2 relaxation[d]	$\left(\frac{-M_x(t)}{T_2}, \frac{-M_y(t)}{T_2}, 0\right)$
Bulk susceptibility[e]	$D\gamma\langle M_z\rangle(M_y(t), -M_x(t), 0)$
Radiation damping[f]	$\frac{1}{\tau_r\lvert\vec{M}\rvert}(-\langle M_x(t)\rangle M_z(t),$ $-\langle M_y(t)\rangle M_z(t),$ $\langle M_x(t)\rangle M_x(t) + \langle M_y(t)\rangle M_y(t))$
Dipole–dipole[g]	$\vec{M}_i(t) \times \frac{\mu_0}{4\pi} \sum_{i\neq j}^{\text{spins}} \frac{\vec{M}_j(t) - \vec{M}_j(t)\cdot\hat{r}}{\lvert r_i - r_j\rvert^3}$
Demagnetizing field[h]	$\vec{M}_i(t) \times \frac{\mu_0}{4\pi} \sum_{i\neq j}^{\text{spins}} \frac{(\vec{M}_j(t) - \vec{M}_j(t)\cdot\hat{r})\Delta r^3}{\lvert r_i - r_j\rvert^3}$
Modulated demagnetizing field[i]	$M_i(t) \times \frac{3(\hat{s}\cdot\hat{B}-1)}{6\tau_d}((\vec{M}_i(t) - \langle\vec{M}\rangle) - ((\vec{M}_i(t) - \langle\vec{M}\rangle)\cdot\hat{r})\hat{r})$

[a] $M_i(t)$ is the magnetization of spin i, $\langle M\rangle$ is the average magnetization over all the spins, and M_0 is the equilibrium magnetization.

[b] γ_i is the ith spin gamma factor.

[c] T_1 is a time constant and $T_1 \geqslant 0$.

[d] T_2 is a time constant and $T_2 \geqslant 0$.

[e] D is the sample shape factor $D = 0$ for a sphere, $D = 1$ for a cylinder, $D = 1/3$ for a flat disk, and $0 \leqslant D \leqslant 1$ for all other cases.

[f] τ_r is the radiation damping constant and is equal to $(2\pi\gamma_i\eta Q M_0)^{-1}$ where η and Q are the probes filling factor and Q factor, respectively.

[g] μ_0 is the permittivity of a vacuum, and $|r_i - r_j|$ is the distance between the two spins.

[h] Δr^3 is the volume of a grid cell.

[i] $\hat{s}$ is the direction of modulation, $\hat{B}$ is the direction of the high magnetic field, and τ_d is the time constant and is equal to $(\gamma\mu_0 M_0)^{-1}$. See [37] for derivation.

object, which is a crucial part of the `Bloch` object. The `Bloch` object is the master container for the spin parameters, pulses, and interactions. This object is then used as the main function driver for the `BlochSolver` object (a useful interface to the ODE solvers).

As magnetic fields are the main interactions of classical spins, there is an entire set of objects devoted to calculating magnetic fields for a variety of coil geometries. The basic shapes of coils, circles, helices, Helmholtz, lines, and spirals, are built-in. These particular objects are heavily parameter based, requiring positions, turns, start and end points, rotations, centering, lengths, etc. One can also create other coil geometries and add them to the basic coil set (examples are provided in the tool kit). The magnetic fields can be added to the offset interaction object to automatically create a range of fields over a grid structure, as well as into other objects to create rotating or other time dependant field objects.

No toolkit would be complete without examples and useful programs. Many programs come included with *BlochLib* (see Table 4). Also included are several Matlab visualization functions (see Table 5) that interact directly with the data output from the magnetic field generators `plotmag`, the trajectories from solving the Bloch equations, `plottraj`, and generic FID and data visualization, `plotter2D` and `Solidplotter`.

2.4. Drawbacks

As discussed above, the power of C++ lies within the object and templates that allow for the creation of generic objects, generic algorithms, and optimization. There are several problems inherent to C++ that can be debilitating to the developer if they are not understood properly. The first three problems revolve around the templates.

Because templated objects and algorithms are generic, they cannot be compiled until used in a specific manner (the template is *expressed*). For example to add two vectors, the compiler must know what data types are inside the vector. Most of the main mathematical kernels in *BlochLib* cannot be compiled until expressed (matrices, vectors, grids, shapes, coords, and the ODE solvers). This can leave a tremendous amount of overhead for the compiler to unravel when a program is actually written and compiled.

The other template problem arises from the expression template algorithms. Each time a new operation is

Table 4
Key examples and implementation programs inside *BlochLib*

Catagory	Folder[a]	Description
Classical	`bulksus`	Bulk susceptibility interaction
	`dipole`	Dipole–dipole interaction over a cube
	`echo`	A gradient echo
	`EPI`	an EPI experiment [38]
	`magfields`	Magnetic field calculators
	`rotating_field`	Using field calculators with the offset interaction
	`splitsol`	Using field calculators for coil design
	`mas`	Simple spinning grid simulation
	`raddamp`	Radiation damping interaction
	`relaxcoord`	T_1 and T_2 off the z-axis
	`simple90`	Simple 90° pulse on an interaction set
	`yylin`	Modulated demagnetizing field example [6]
Quantum	`MMMQMAS`	A complex MQMAS program
	`nonsec`	Nonsecular quadrupolar terms exploration
	`perms`	Permutations on pulse sequences
	`shapes`	A shaped pulse reader and simulator
	`Solid-2.0`	General solid state NMR simulator
Other	`classes`	Several 'How-To' class examples
	`data_readers`	Data reader and conversion programs
	`diusion`	1D diffusion example
	`mpiplay`	Basic MPI examples

[a] These folders correspond to the folders inside the distribution.

Table 5
Available Matlab visualization functions in *BlochLib*

Matlab function	Description
`Solidplotter`	A GUI that plots many of the NMR file formats
`plotter2D`	A sub function of `Solidplotter` that performs generic data plotting
`plotmag`	Provides many visualization functions for the magnetic field calculators
`plottrag`	Visualization of magnetization trajectories from classical evolutions

performed on an expression template data type (like the vectors), the compiler must first unravel the expression, then create the actual machine code. This can require a large amount of time to perform, especially if the operations are complex. The two template problems combined require large amounts of memory and CPU time to perform, however, the numerical benefits usually overshadow these constraints. For example the `bulksus` example in *BlochLib* takes approximately 170 Mb of memory and around 90 s (using gcc 2.95.3) to optimally compile one source file, but the speed increase is approximately a factor of 10 or greater. Compiler's themselves are getting better at handling the template problems. For the same `bulksus` example, the gcc 3.1.1 compiler took approximately 100 Mb of memory and around 45 s of compilation time.

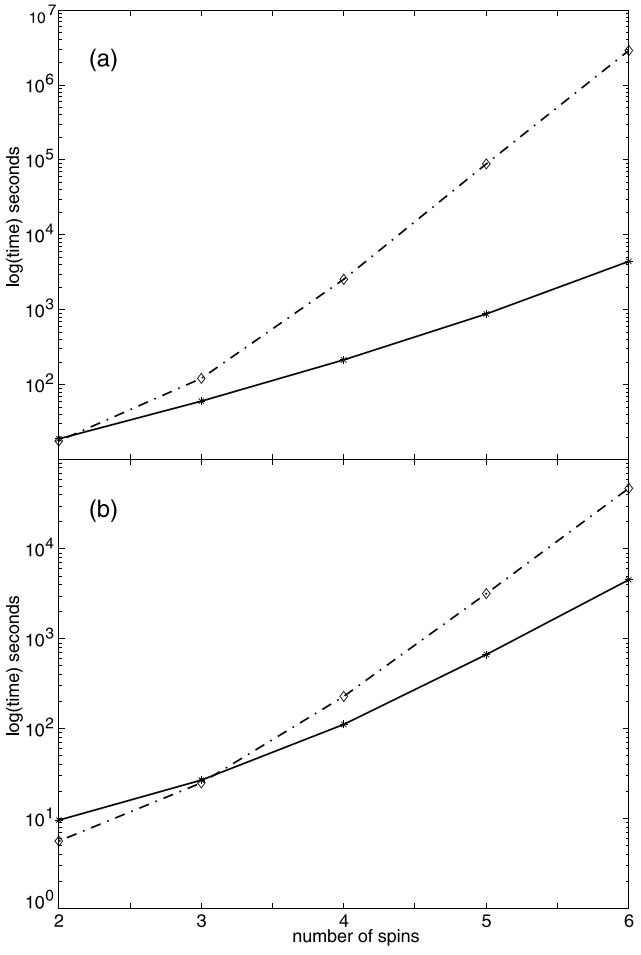

Fig. 6. Time for simulations of *Solid* (solid line) and *Simpson* (dashed–dotted line) as a function of the number of spins. (a) shows the simulation of a rotary resonance experiment on set pair of spins. Conditions are the same as those shown in [3, Fig. 5]. (b) shows the speed of the simulation of a C7 with simulations conditions the same as those shown in Figure 6e of [3]. In both cases the extra spins are protons with random CSAs that have no interactions between with the detected ^{13}C nuclei. *Solid* tends to be slower for small spin sets as explained in Section 2.4. All simulations were performed on a 700 MHz Pentium III Xeon (Redhat 7.3), compiled with gcc 2.95.3.

The final template problem arises from expression template arithmetic, which require a memory copy upon assignment (i.e., $A = B$). Non-expression template data types can pass pointers to memory rather than the entire memory chunk. For smaller array sizes, the cost of this copying can be significant with respect to the operation cost. The effect is best seen in Fig. 5 where the pointer copying used for the ATLAS test saves a few MFLOPS as opposed to the *BlochLib* version. However, as the matrices get larger the relative cost becomes much smaller.

The last problem for C++ is one of standardization. The C++ standard is not well adhered to by every compiler vendor. For instance Microsoft's Visual C++ will not even compile the most basic template code. Other compliers cannot handle the memory requirements for expression template unraveling (CodeWarrier (Metrowerks) crashes constantly because of memory problems from the expression templates). The saving

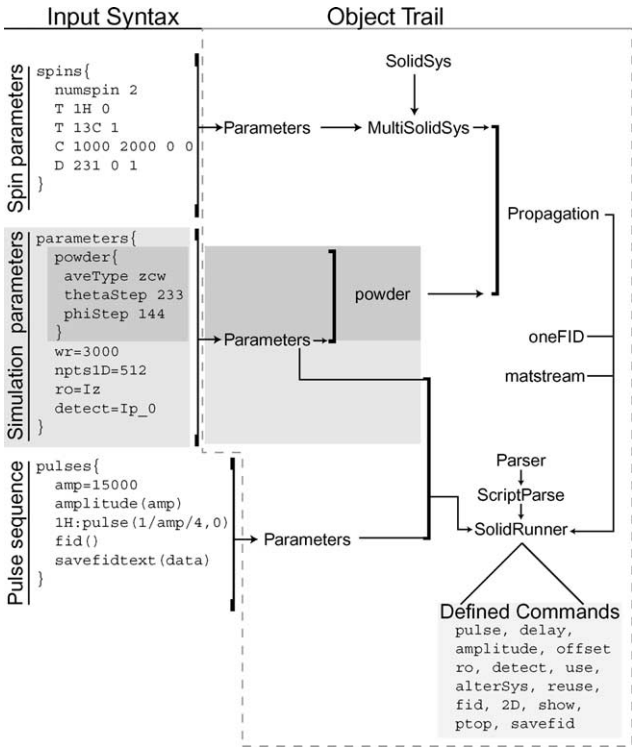

Fig. 7. The design of the EE program *Solid* derived from the input syntax. Three basic sections are needed. Definitions of a solid system (`spins`), definition of powder average types, other basic variables and parameters (`parameters` and the subsection `powder`), and finally the definition of a pulse section where spin propagation and fid collection is defined (`pulses`). The `pulses` section contains the majority of *Solid*'s functionality. Based on this input syntax, a simple object trail can be constructed. `MultiSolidSys` contains at least one (or more) `SolidSys` objects. This combined with the powder section/object defines the `Propagation` object where the basic propagation algorithms are defined. Using the extendable `ScriptParse` object, the `SolidRunner` object defines the new functions available to the user. `SolidRunner` then combines the basic FID algorithms (in the `oneFID` object), the `Propagation` object, and the output classes to perform the NMR experimental simulation.

grace for these problems is the GNU compiler, which is a good optimizing compiler for almost every platform. GNU g++ 3.2.1 adheres to almost every standard and performs optimization of templates efficiently.

3. Various implementations

This section will describe a basic design template to create programs from *BlochLib* using the specific example of the program *Solid*. *Solid* is a generic NMR simulator. Several other programs are briefly described within the design template. The emphasis will not be on the simulations themselves, but more on their creation and the modular nature the tool kit.

There is potentially an infinite number of programs that can be derived from *BlochLib*, however, the tool kit comes with many of the basic NMR simulations programs already written and optimized. These programs serve as a good starting place for many more complex programs. In Table 4 is a list of the programs included and their basic function. Some of them are quite complicated while others are very simple. Describing each one will show a large amount of redundancy in how they are created. A few of the programs which represent the core ideologies used in *BlochLib* will be explicitly considered in the following sections.

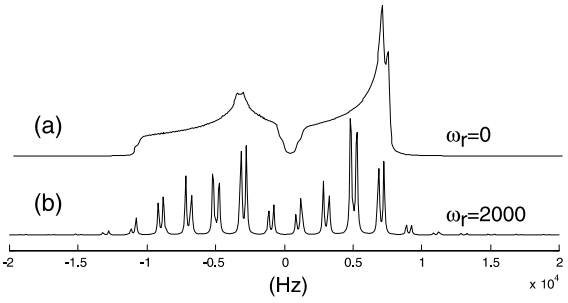

Fig. 8. A two spin system as simulated by *Solid* where (a) is with no spinning, and (b) is with a rotor speed of 2000 Hz at the magic angle (54.7°. The spins system included 2 CSA's with the first spins parameters as $\omega_{iso} = 5000 * 2\pi$, $\omega_{ani} = 4200 * 2\pi$, and $\eta = 0$, the second spin's parameters as $\omega_{iso} = 5000 * 2\pi$, $\omega_{ani} = 6012 * 2\pi$, and $\eta = 0.5$, with a scalar J coupling of 400 Hz between the two spins. For (a) and (b) 3722 and 2000 powder average points were used respectively. See http://waugh.cchem.berkeley.edu/solid/ for the input configuration file.

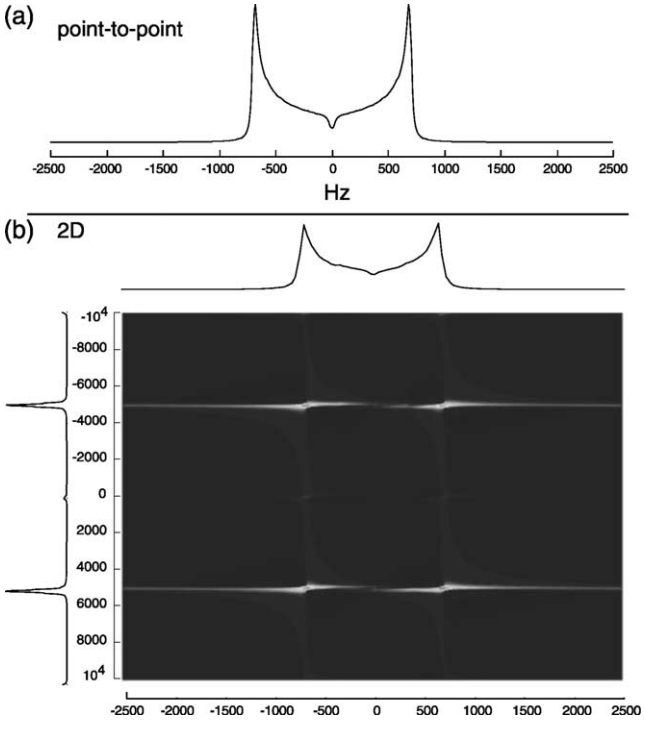

Fig. 9. A two spin system as simulated by *Solid* of the post-C7 sequence where (a) is collected using a point-to-point acquisition, and (b) is a full 2D collection. The spins system includes a dipole coupling of 1500 Hz. For both (a) and (b) 233 powder average points were used. See http://waugh.cchem.berkeley.edu/solid/ for the input configuration file.

3.1. Solid

The program *Solid* represents the basic EE quantum mechanical simulation program. *Solid*s basic function is

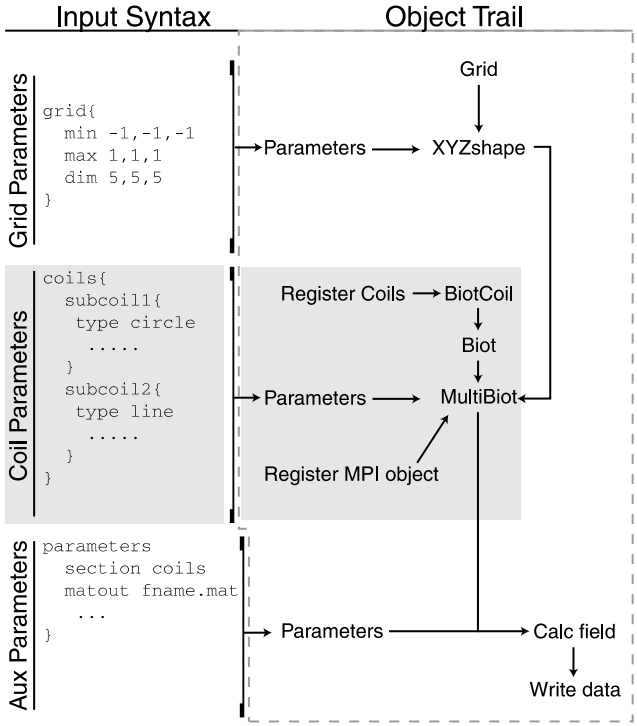

Fig. 10. The basic design for the Field Calculator program. There are two basic parameters sections needed. The first describes the coil geometry using the basic elements (see text) and any user written coils geometries. The second describes the Cartesian grids where the field will actually be calculated. Again once the parameter sets are known a simple object trail can be developed. Initially the user must register their own geometries into the `BiotCoil` list. The parameters then feed into the `XYZshape` and `MultiBiot` objects. Parallelization can be implemented simply by defining the `MPIworld` object and passing it to the `MultiBiot` object. The data are written in two formats; into one readable by Matlab for visualization and into a file readable by the `MultiBiot` object.

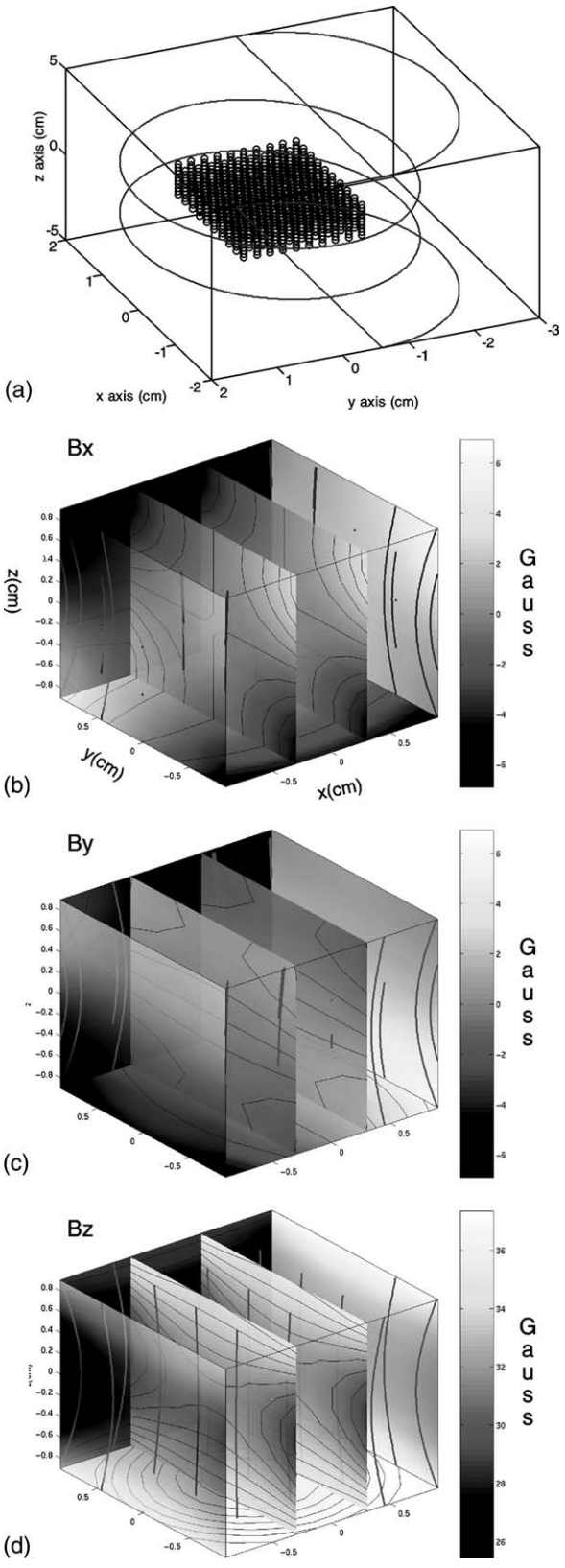

Fig. 11. The magnetic field calculated by the program shown in Fig. 10 given in the `magfields` folder of the distribution. The Matlab function, `plotmag`, was used to generate these figures (see Table 5). The coil and the sampled grid are shown in (a), the fields along the x, y, z directions are shown in (b)–(d), respectively.

to simulate most 1D and 2D NMR experiments. It behaves much like *Simpson* but is faster for large spin sets as shown in Fig. 6. It is essentially a parameter parser which then sends the obtained parameters to the main kernel for evaluation. The EE diagram (Fig. 1) can be extended to more specific object usage used in Solid (Fig. 7). *Solid* has three stages, parameter input, main kernel composition, and output structures. The EE normal section, parameter parser, was written to be the main interface to the kernel and output sections. It extends the `ScriptParse` object to add more simulation specific commands (spin evolution, FID calculation, and output).

There are three basic acquisition types *Solid* can perform: a standard 1D, a standard 2D, and a point-to-point (obtains the indirect dimension of a 2D experiment without performing the entire 2D experiment). Simple 1D simulations are shown in Fig. 8. The results of a 2D and point-to-point simulation of the post-C7 sequence [32] are shown in Fig. 9 [40].

3.2. Classical program: magnetic field calculators

Included in *BlochLib* is the ability to calculate magnetic fields over arbitrary coil geometries. The main field

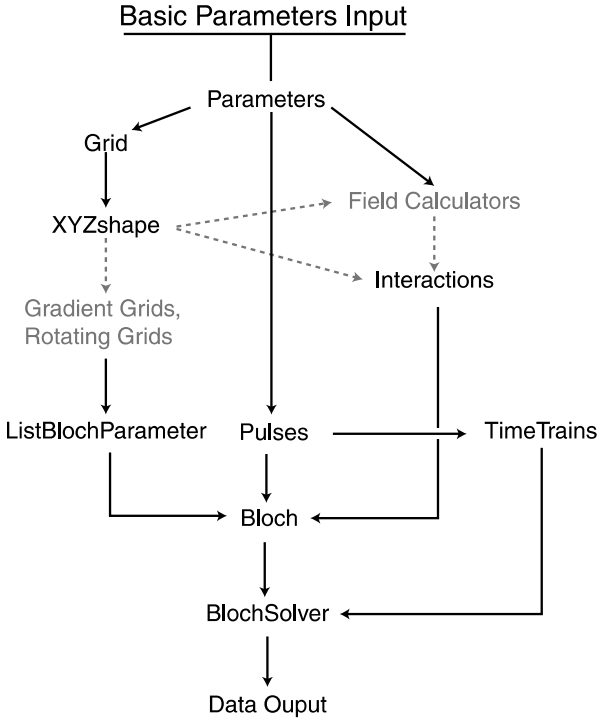

Fig. 12. A rough design for a classical Bloch simulation over various interactions. These programs typically need as much optimization as possible in order to function optimally over large spin sets. As a result, the parameter input is expected to be minimal, with the bulk of the design to aid in optimization of the interaction sets and pulse sequences used. Items in gray are optional objects, that can be simply added in the specific object chain to be used.

algorithm calculates a discretetized integral of Ampere's equation for the magnetic field.

$$B(r) = \frac{\mu_0}{4\pi} \int \frac{I(r') \times \mathrm{d}l(r')}{|r - r'|^2}, \tag{7}$$

where the magnetic field at the point r, $B(r)$, is the volume integral of the current at r', $I(r')$, crossed into a observation direction, $\mathrm{d}l(r')$, divided by the square of the distance between the observation point, r, and the current point, r'. One way to evaluate this integral numerically, the integral is broken into a sum over little lines of current (the Biot–Savart law). For this to function properly numerically, the coil must be divided into small line segments.

There are numerous coil geometries, but most of the more complex designs can be broken into a set of primitive objects. The geometric primitives included in *BlochLib* are lines, circles, spirals, helices, an ideal Helmholtz pair (basically two circles separated by a distance), a true Helmholtz pair (two sets of overlapping helices), input files of points, and constant fields. *BlochLib* also allows the user to create their own coil primitives and combine them along with the rest of the basic primitives. Fig. 10 shows the basic design of the field calculator using the `MultiBiot` object. This program is included in *BlochLib* under the `magfields` directory (see Table 4). Fig. 11 shows the data generated by the program. The input file for this program can be seen in the `magfields` folder in the distribution. It should be noted that the convergence of the integral in Eq. (7) is simply a function of the number of line segments you choose for the coils.

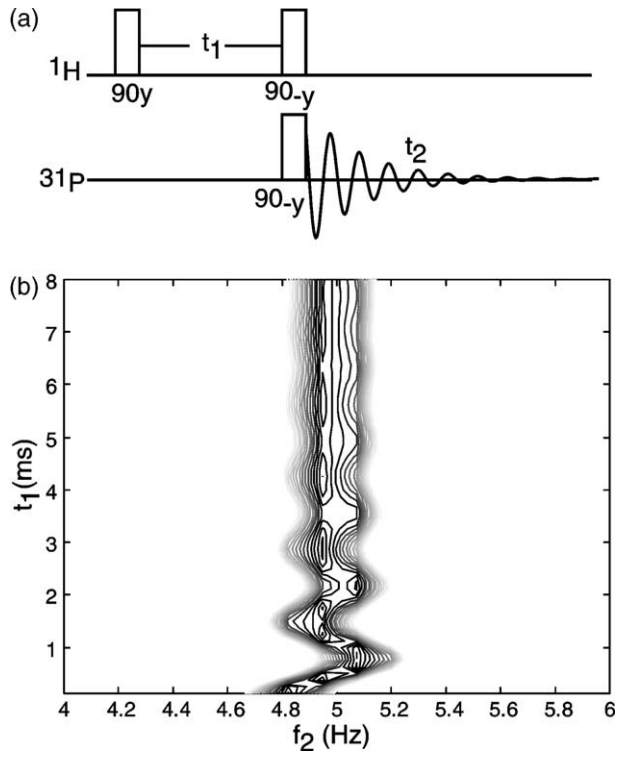

Fig. 13. Simulated data from a HETCOR (Heteronuclear Correlation) experiment showing the effect of bulk susceptibility on the offset of the ^{31}P. (a) shows the simulated pulse sequence and (b) shows the simulated data. This simulation is an attempt to replicate Fig. 2 from [33]. In order to correctly replicate the figure, the ^{1}H offset had to be changed to 722 Hz (the reference quotes 115 Hz as the offset, but it seems a factor of 2π was omitted ($722 = 2\pi * 115$). The T2 relaxation of the ^{1}H also had to be altered to 0.002 s (the reference quotes 0.015 s, however the diagram shows a much faster decay then this time).

3.3. Classical programs: Bloch simulations

Programs of this type are designed to function on large spin sets optimally based on the interactions present. The basic layout for these simulation can be see in Fig. 12. The `Grid` serves as the basis for much of the rest of the Bloch interactions and Bloch parameters. Grids also serve as the basis for gradients and physical rotations. The interactions are also a key part of the simulation and can rely on the grid structures as well as

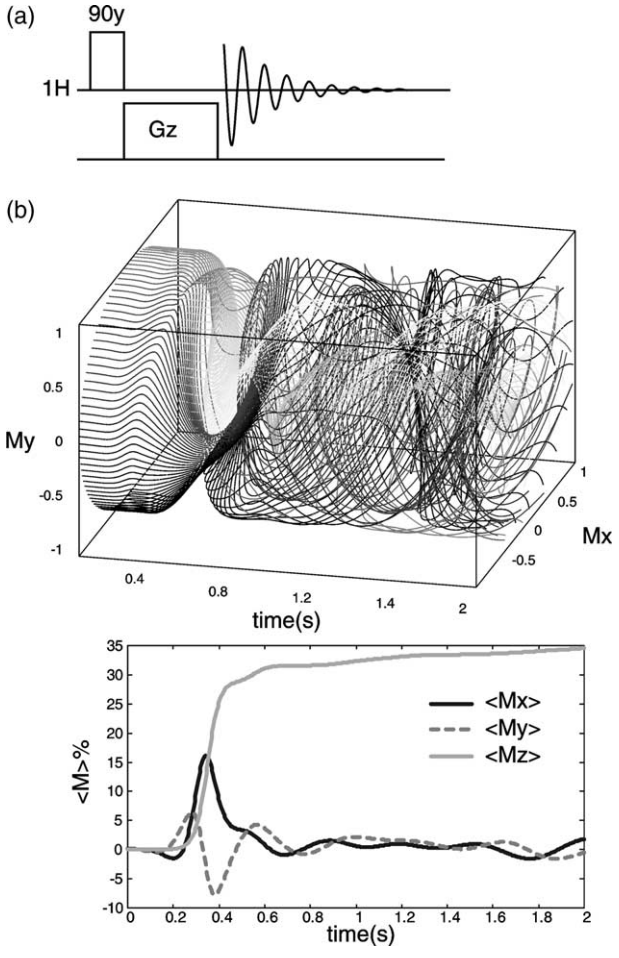

Fig. 14. Simulated resurrection of magnetization after a crusher gradient pulse in the sequence shown in (a). (b) shows the effect of radiation damping and the modulated demagnetizing field. The result is a non-linear partial resurrection of magnetization. The input parameters are those in [6]. The data were plotted using `plottrag` in the distribution.

any magnetic fields calculated. A pulse on a Bloch system represents a type of impulse function to the system. A pulse should be treated as a separate numerical integral step due to this impulse nature (such impulses can play havoc with ODE solvers). The pulses, Bloch parameters, and interactions are finally wrapped into a master object, `Bloch`, which is then fed into the `BlochSolver` object which performs the integration.

3.3.1. Bulk susceptibility

One such implementation attempts to simulate the result obtained in [33, Fig. 2]. This is a HETCOR (heteronuclear correlation) experiment between a proton and a phosphorous. The delay in the HETCOR sequence (see Fig. 13a) allows the offset of the ^{1}H to evolve. Next the ^{1}H magnetization is placed back on the z-axis. The z-magnetization of the proton will oscillate (its magnitude effected by the time evolved under the delay). If one then places the ^{31}P magnetization in the xy-plane and collects an FID, the ^{31}P will feel a slight offset shift due to the varying ^{1}H z-magnetization (effect of the bulk susceptibility). Thus in the indirect dimension an oscillation of the ^{31}P magnetization due to the protons will be observed. The results is shown in Fig. 13b and matches the result obtained in [33]. The code for this diagram is in the `bulksus` folder of the distribution.

3.3.2. Radiation damping

Another interesting implementation attempts to emulate the result obtained by Lin et al. [6]. In this simulation, the interplay between radiation damping and the demagnetizing field resurrect a completely crushed magnetization (a complete helical winding). Radiation damping is responsible for the resurrection as the demagnetizing field alone does not resurrect the crushed magnetization. The simulated data (Fig. 14b) matches the data found in [6, Fig. 2]. The code for this diagram is in the `yylin` folder of the distribution.

3.3.3. Probe coils

The final example involves analyzing an NMR probe coil design. Dynamic angle spinning (DAS) [34] experiments require the probe stator to move during the experiment. A solenoid coil moves with the stator, however, as the stator angle approaches 0 degrees (with respect to the high field), there would be little detected signal (or pulse power transmission) because the high static magnetic field and coils field are parallel (resulting in a 0 cross product). One can remove this shortcoming by removing the coil from the stator. But this represents its own problem if the coil is a solenoid, because the stator is large compared to the sample, and thus the solenoid would also have to be large thus reducing the filling and quality factor too much to detect any signal. A suitable alteration to the solenoid would be to split it. The entire probe design is the subject of a forth coming paper [35]. To optimize the split solenoid design one needs to see factors like inhomogeneities and effective power within the sample area. Fig. 15b shows a split solenoid design as well the inhomogeneity (Fig. 15d) profile along the xy-plane (the high field removes any need for concern about the z-axis). Compared with a normal solenoid, Fig. 15a, the field profile is much more distorted (Fig. 15c), also given the same current in the two coils, the solenoid has six times more field in the region of interest then the split-coil design. The figure also shows us a weak spot in the split-coil design. The wire that connects the two helices creates the majority of the asymmetric field profile, and is the major contrib-

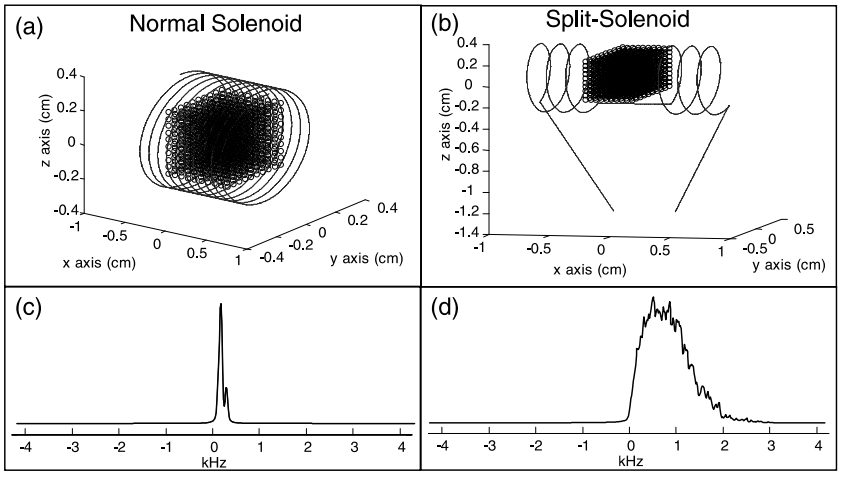

Fig. 15. The magnetic field distribution of two standard Solid State NMR probe detection coil. (a) shows a standard solenoid coil (10 turns/cm with a radius of 0.3175 cm) as well as the region of interest for the magnetic field (black points). (b) shows the split solenoid (3 turns/cm with a radius of 0.3175 cm and a splitting of 0.6 cm) as well as the region of interest for the magnetic field (black points). (c) shows the effective inhomogeneity of the coil in (a) for a proton given a 0.5 A current. The small peak to the right of the main peak is the edges of the sampled rectangle close to the coil. (d) shows the effective inhomogeneity of such the coil in (b) for a proton given a 3.0 A current. The majority of the inhomogeneity is due to the small line connecting the two helical segments. The average field of the coil was subtracted from the result in (c) and (d).

utor to the inhomogeneity across the sample. Correcting this by a U shape (or equivalent) should aid in correcting the profile.

4. Conclusions

Throughout the paper, emphasis on the generic physical simulation design is discussed for the specific case of NMR. The created tool kit, *BlochLib*, adheres to these basic design ideas (OOP, inheritance, and expression-templates). *BlochLib* is designed to be the next generation of simulation tool kits for NMR. It is highly optimized and generalized for almost any NMR simulation situation. It has been shown that utilizing relatively modern numerical techniques and algorithms allows a study of more complicated spin dynamics under various interactions and experimental designs then previous NMR tool kits. The input of complex parameters, coding, and creation of programs should be easy and highly optimized for both the classical and quantum mechanical aspects of NMR. Diffusion and other partial differential equation entities (like fluid flow) are currently being designed for inclusion into the tool kit. Relaxation using normal Louville space operators and Redfield approximations should also be included. The total tool kit and documentation can be found at http://waugh.cchem. berkeley.edu/blochlib/.

Acknowledgments

The author wished to thank Robert Havlin and Jamie Walls for quantum mechanical discussions, Andreas Trabesinger for classical discussions, John Logan for assistance with the quadrupole interactions, Dimitri Sakellariou for field calculation discussions, Josef Granweh for useful comments, and Alex Pines for his support. This work was supported by the Director, Office of Science, Office of Basic Energy Sciences, Materials Science and Engineering Division, US Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] R.R. Ernst, G. Bodenhausen, A. Wokaun, Principles of Nuclear Magnetic Resonance in One and Two Dimensions, Clarendon Press, Oxford, 1989.
[2] J. Walls, W.B. Blanton, R.H. Havlin, A. Pines, Chem. Phys. Lett. 363 (2002) 372–380.
[3] M. Bak, J.T. Rasmussen, N.C. Nielsen, J. Magn. Reson. 147 (2000) 296.
[4] P. Hodgkinson, L. Emsley, Prog. Nucl. Magn. Reson. Spectrosc. 36 (2000) 201.
[5] S. Smith, T. Levante, B. Meier, R. Ernst, J. Magn. Reson. 106a (1994) 75.
[6] Y.Y. Lin, N. Lisitza, S.D. Ahn, W.S. Warren, Science 290 (5489) (2000) 118.
[7] C.A. Meriles, D. Sakellariou, H. Heise, A.J. Moule, A. Pines, Science 293 (2001) 82.
[8] H. Heise, D. Sakellariou, C.A. Meriles, A. Moule, A. Pines, J. Magn. Reson. 156 (2002) 146.
[9] T.M. Brill, S. Ryu, R. Gaylor, J. Jundt, D.D. Griffin, Y.Q. Song, P.N. Sen, M.D. Hurlimann, Science 297 (2002) 369.
[10] R. McDermott, A.H. Trabesinger, M. Muck, E.L. Hahn, A. Pines, J. Clarke, Science 295 (2002) 2247.
[11] J.D. Walls, M. Marjanska, D. Sakellariou, F. Castiglione, A. Pines, Chem. Phys. Lett. 357 (2002) 241.
[12] R.H. Havlin, G. Park, A. Pines, J. Magn. Reson. 157 (2002) 163.
[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, MA, 1995.
[14] B. Stroustrup, The C++ Programming Language, third ed., Addison-Wesley, Boston, MA, 1997.
[15] S. Haney, Comput. Phys. 8 (6) (1994) 690.
[16] T.L. Veldhuizen, M.E. Jernigan, in: Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97), Lecture Notes in Computer Science, Springer, Berlin, 1997.
[17] T. Veldhuizen, C++ Rep. 7 (5) (1995) 26.
[18] T. Veldhuizen, C++ Rep. 7 (4) (1995) 36.
[19] U.W. Eisenecker, K. Czarnecki, Generative Programming—Towards a New Paradigm of Software Engineering, Addison Wesley, Boston, MA, 2001.
[20] C. Pescio, C++ Rep. 9 (7) (1997).
[21] W. Clint, Automatically Tuned Linear Algebra Software (ATLAS), URL http://math-atlas.sourceforge.net.
[22] M. Frigo, S.G. Johnson, Tech. Rep., Massachusetts Institute of Technology, 1997, URL http://fftw.org.
[23] E. Lusk, Tech. Rep., University of Tennessee, 1997, URL http://www.mpi-forum.org.
[24] F. James, Tech. Rep., Computing and Networks Division CERN Geneva, Switzerland, 1998, URL http://wwwinfo.cern.ch/asdoc/minuit/minmain.html.
[25] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C. The Art of Scientific Computing, Cambridge University Press, Cambridge, 1997.
[26] J.R. Cash, A.H. Karp, ACM Transactions on Mathematical Software, vol. 16, 1990.
[27] J. Stoer, R. Bulirsch, Introduction to Numerical Analysis, Springer, Berlin, 1980.
[28] P. Deuflhard, Numer. Math. 41 (1983) 399.
[29] P. Deuflhard, SIAM Rev. 27 (1985) 505.
[30] G. Bader, P. Deuflhard, Numer. Math. 41 (1983) 373.
[31] M. Hohwy, H. Bildse, N.C. Nielsen, J. Magn. Reson. 136 (6) (1999).
[32] M. Hohwy, H.J. Jakobsen, M. Eden, M.H. Levitt, N.C. Nielsen, J. Chem. Phys. 108 (1998) 2686.
[33] M.P. Augustine, K.W. Zilm, J. Magn. Reson. Ser. A 123 (1996) 145.
[34] Mueller K.T., B. Sun, G. Chingas, J. Zwanziger, T. Terao, A. Pines, J. Magn. Reson. 86 (3) (1990) 470.
[35] R.H. Havlin, T. Mazur, W.B. Blanton, A. Pines, 2002, in preparation.
[36] S. Wi, L. Frydman, J. Chem. Phys. 112 (7) (2000) 3248.
[37] G. Deville, M. Bernier, J.M. Delrieux, Phys. Rev. B 19 (11) (1979) 5666.
[38] M.K. Stehling, R. Turner, P. Mansfield, Science 254 (5028) (1991) 43.
[39] In the current version of *BlochLib*, other flow type interactions like diffusion are not treated.
[40] The website http://waugh.cchem.berkeley.edu/solid/ has the input configuration for these simulations and other simulations.