

High Performance Computations in NMR

by

Wyndham Bolling Blanton

B.S. Chemistry (Carnegie Mellon University) 1998

B.S. Physics (Carnegie Mellon University) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Chemistry

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alexander Pines, Chair

Professor Jeffrey A. Reimer

Professor Raymond Y. Chiao

David E. Wemmer

Fall 2002

The dissertation of Wyndham Bolling Blanton is approved:

Chair

Date

Date

Date

Date

University of California, Berkeley

Fall 2002

High Performance Computations in NMR

Copyright © 2002

by

Wyndham Bolling Blanton

Abstract

High Performance Computations in NMR

by

Wyndham Bolling Blanton

Doctor of Philosophy in Chemistry

University of California, Berkeley

Professor Alexander Pines, Chair

As an analytic noninvasive technique to study molecules in their natural environment, NMR has little equal. The advancement of the technique is beginning to enter a new phase, where many body dynamics, complex control, and precise measurements of many body spin properties preclude any exact theoretical treatment. Approximation methods and other reductions in the set of parameter spaces are currently used to obtain some form of intuition about a simplified NMR system; however, to exactly profile a real system, numerical simulation is required.

The scope of most NMR simulations is chiefly regulated to small spin systems, where the dynamics are simplified enough to simulate efficiently. The cause is typically based on a poor understanding of how to simulate an NMR situation effectively and efficiently. This seems consistent with the fact that most NMR spectroscopists are not computer scientists as well. The introduction of novel programming paradigms and numerical techniques seems to have eluded the field. A complete simulation environment for NMR is

presented here marrying three fundamental aspects of simulations 1) numerical speed and efficiency, 2) simplicity in implementation, and 3) NMR specific algorithmic developments.

The majority of numerical NMR is reduced to a simple simulation framework. The framework allows for more complex simulations for explorations of both many body spin dynamics and control. A specific large scale simulation is applied to recoupling sequences in solid-state NMR. Using simple permutations on base pulse sequences can result in control enhancements on both the simple system and the many body system beyond a theoretical approach. The sheer number of permutations required to solve the problem would have certainly been impossible without the aid of this framework. This new framework now opens other unexplored possibilities of using simulation as a development tool for the larger problems of many body dynamics and control.

Professor Alexander Pines
Dissertation Committee Chair

To my Grandmother and Grandfather, Lucy and Wyndham Jr.

Contents

List of Figures	v
List of Tables	viii
1 Introduction	1
2 Computer Mechanics	4
2.1 Data Types	7
2.2 The Object	8
2.2.1 Syntax	9
2.3 Expression Templates	13
2.3.1 Motivations	13
2.3.2 Stacks	14
2.3.3 An Array Object and Stacks	15
2.3.4 Expression Template Implementation	19
2.4 Optimizing For Hardware	27
2.4.1 Basic Computer Architecture	30
2.4.2 A Faster Matrix Multiplication	37
3 NMR Forms	42
3.1 Classical Mechanics	42
3.2 Bloch Equation Magnetic Fields	43
3.3 Quantum Mechanics	59
3.3.1 Rotations	60
3.3.2 Rotational Frames	64
3.3.3 The Hamiltonians	67
3.4 NMR Initial Conditions	73
3.4.1 Quantum	73
3.4.2 Classical	74
4 NMR Algorithms	76
4.1 Classical Algorithms	76
4.1.1 Eigenvalue Problem	76
4.1.2 ODE solvers	78
4.2 Quantum Algorithms	82

4.2.1	The Direct Method	82
4.2.2	Periodicity and Propagator Reduction	83
4.2.3	Eigenspace	89
4.2.4	Periodicity and Eigen-Space methods	95
4.2.5	Non-periodic Hamiltonians	100
4.2.6	Powder Average Integration	100
4.3	Conclusions and Comments	103
5	BlochLib	105
5.1	Introduction	105
5.2	The Abstract NMR Simulation	106
5.2.1	Experimental Evolutions (EE)	106
5.2.2	Theoretical Evolutions (TE)	106
5.2.3	Existing NMR Tool Kits	108
5.2.4	Why Create a new Tool Kit?	109
5.3	BlochLib Design	109
5.3.1	Existing Numerical Tool Kits	110
5.3.2	Experimental and Theoretical Evolutions for NMR simulations	111
5.3.3	<i>BlochLib</i> Layout	112
5.3.4	Drawbacks	121
5.4	Various Implementations	123
5.4.1	Solid	124
5.4.2	Classical Program: Magnetic Field Calculators	129
5.4.3	Classical Programs: Bloch Simulations	131
5.5	Conclusions	140
6	Massive Permutations of Rotor Synchronized Pulse Sequences	141
6.1	Introduction	141
6.1.1	Rotor Synchronization	142
6.2	Background Theory	143
6.2.1	Average Hamiltonian	143
6.2.2	Recoupling RSS	145
6.2.3	C7	150
6.2.4	Removable of Higher Order Terms	151
6.3	Permutations	155
6.3.1	The Sub-Units	155
6.3.2	The Measure	156
6.3.3	Algorithmic Flow	158
6.4	Data and Results	161
6.4.1	Sequence Measures	161
6.4.2	Transfer Efficiencies	185
6.5	Conclusions	196
7	Future Expansions	201
7.1	Evolutionary Algorithms (EA)	202
7.2	Neural Networks	209
7.3	Final Remarks	211

Bibliography	213
A Auxillary code	225
A.1 General C++ code and examples	225
A.1.1 C++ Template code used to generate prime number at compilation	225
A.1.2 C++ Template meta-program to unroll a fixed length vector at compilation time	226
A.1.3 C++ code for performing a matrix multiplication with L2 cache blocking and partial loop unrolling.	228
A.1.4 An MPI master/slave implimentation framework	230
A.1.5 C++ class for a 1 hidden layer Fully connected back-propagation Neural Network	232
A.2 NMR algorithms	239
A.2.1 Mathematica Package to generate Wigner Rotation matrices and Spin operators.	239
A.2.2 Rational Reduction C++ Class	244
A.2.3 Optimized static Hamiltonian FID propogation	252
A.2.4 $\gamma - COMPUTE$ C++ Class	253
A.3 BlochLib Configurations and Sources	263
A.3.1 <i>Solid</i> configuration files	263
A.3.2 Magnetic Field Calculator input file	266
A.3.3 Quantum Mechanical Single Pulse Simulations	267
A.3.4 Example Classical Simulation of the Bulk Susceptibility	267
A.3.5 Example Classical Simulation of the Modulated Demagnetizing Field	274

List of Figures

2.1	A two state Turing machine	6
2.2	A simple stack tree	15
2.3	How the compiler unrolls an expression template set of operations.	25
2.4	DAXPY speed tests	26
2.5	A pictorial representation for the matrix–matrix tensor multiplication	28
2.6	Speed in MFLOPS of a matrix–matrix multiplication	29
2.7	A generic computer data path.	30
2.8	Pipe lines and loop unrolling	34
2.9	A 128 bit SIMD registers made of 4–32 bit data values	35
2.10	Cache levels in modern Processors	36
2.11	Speed comparison in MFLOPS of loop unrolling	39
2.12	Speed comparison in MFLOPS of L2 cache blocking and loop unrolling	40
3.1	The magnitude of the dipole field	52
3.2	The magnetization of a sample inside a magneti field.	55
3.3	Magnetization in iso–surfaces versus the applied magnetic field, B_o , the temperature T , and number of moles.	75
4.1	Various propagators needed for an arbitrary rational reduction.	84
4.2	Effectiveness of the rational propagator reduction method.	89
4.3	Diagram of one Hamiltonian period and the propagator labels used for the COMPUTE algorithm	96
4.4	Octants of equal volume of a sphere.	102
5.1	Experimental Evolutions and Theoretical Evolutions	107
5.2	The basic design layout of the <i>BlochLib</i> NMR tool kit.	113
5.3	$C=A*B*adjoint(A)$ speed of <i>BlochLib</i>	115
5.4	<i>Solid</i> vs. <i>Simpson</i>	125
5.5	The design of the EE program <i>Solid</i> derived from the input syntax.	127
5.6	1D static and spinning 2 spin simulation	128
5.7	1D and 2D post- <i>C7</i> simulation	128
5.8	The basic design for the Field Calculator program.	130
5.9	Magnetic field of a D–circle	132
5.10	A rough design for a classical Bloch simulation over various interactions.	133

5.11	Bulk susceptibility HETCOR	135
5.12	Simulation of radiation damping and the modulated local field	136
5.13	Magnetic field of a split solenoid	138
5.14	Magnetic field of a solenoid	139
6.1	A general rotor synchronized pulse sequence a) using pulses and delays, and b) using a quasi continuous RF pulse.	142
6.2	The two RSS classes C (a) and R (b).	147
6.3	Compensated C (a), R (b) and posted C (c), R (d) RSS sequences.	149
6.4	Post- $C7$ transfer efficiencies on a two spin system with $\omega_r = 5kHz$ for various dipolar coupling frequencies	152
6.5	Different base permutations on the post- $C7$ sequence	153
6.6	Spin system SS_1 with 4 total number of $C7$ s applied.	164
6.7	Spin system SS_1 with 8 total number of $C7$ s applied.	165
6.8	Spin system SS_1 with 12 total number of $C7$ s applied.	166
6.9	Spin system SS_1 with 16 total number of $C7$ s applied.	167
6.10	Spin system SS_1 with 20 total number of $C7$ s applied.	168
6.11	Spin system SS_1 with 24 total number of $C7$ s applied.	169
6.12	Spin system SS_1 with 32 total number of $C7$ s applied.	170
6.13	Spin system SS_1 with 40 total number of $C7$ s applied.	171
6.14	Spin system SS_1 with 48 total number of $C7$ s applied.	172
6.15	Spin system SS_2 with 4 total number of $C7$ s applied.	173
6.16	Spin system SS_2 with 8 total number of $C7$ s applied.	174
6.17	Spin system SS_2 with 12 total number of $C7$ s applied.	175
6.18	Spin system SS_2 with 16 total number of $C7$ s applied.	176
6.19	Spin system SS_2 with 24 total number of $C7$ s applied.	177
6.20	Spin system SS_2 with 32 total number of $C7$ s applied.	178
6.21	Spin system SS_3 with 4 total number of $C7$ s applied.	179
6.22	Spin system SS_3 with 8 total number of $C7$ s applied.	180
6.23	Spin system SS_3 with 12 total number of $C7$ s applied.	181
6.24	Spin system SS_3 with 16 total number of $C7$ s applied.	182
6.25	Spin system SS_3 with 24 total number of $C7$ s applied.	183
6.26	Spin system SS_3 with 32 total number of $C7$ s applied.	184
6.27	Pulse sequence, initial density matrices and detection for a transfer efficiency measurement.	187
6.28	Transfer efficiencies for a 4 fold application of the basic $C7$ and the post- $C7$ for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	188
6.29	3D transfer efficiencies plots for a 4,8,12,16 fold application of the post- $C7$ and the best permutation cycles for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	190
6.30	Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- $C7$ and the best permutation cycles for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	191
6.31	3D transfer efficiencies plots for a 4,8,12,16 fold application of the post- $C7$ and the best permutation cycles for the SS_2 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	192

6.32	Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- <i>C7</i> and the best permutation cycles for the SS_2 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	193
6.33	3D transfer efficiencies plots for a 4,8,12,16 fold application of the post- <i>C7</i> and the best permutation cycles for the SS_3 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	194
6.34	Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- <i>C7</i> and the best permutation cycles for the SS_3 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$	195
6.35	Transfer Efficiencies using the post- <i>C7</i> and the best permuted cycles across over different cycles for the SS_1 spin system.	197
6.36	Transfer efficiencies using the post- <i>C7</i> and the best permuted cycles across over different cycles for the SS_2 spin system.	198
6.37	Transfer efficiencies using the post- <i>C7</i> and the best permuted cycles across over different cycles for the SS_3 spin system.	199
7.1	The standard evolutionary strategy methods and controls.	204
7.2	An arbitrary permutation cycle parent genes and resulting child.	205
7.3	Evolution Programming (EP) generation step for an $ES_{(2,1)}$ strategy.	206
7.4	Genetic Algorithm (GA) generation step for an $ES_{(3,2)}$ strategy.	207
7.5	Differential Evolution (DE) generation step for an $ES_{(3,1)}$ strategy.	208
7.6	Basic 1 and 2 layer feed–forward neural networks.	209

List of Tables

2.1	Basic High Level Language Data Types	8
2.2	SIMD registers available of common CPUs	34
3.1	Wigner rank 1 rotation elements, $D_{m,m'}^1$	62
3.2	Reduced Wigner rank 2 rotation elements, $d_{m,m'}^2$	63
3.3	Spherical tensor basis as related to the Cartesian basis for spin i and spin j	67
4.1	Time propagation using individual propagators via the Direct Method	86
4.2	A reduced set of individual propagators for $m = 9$ and $n = 7$	86
4.3	Matrix Multiplication (MM) reduction use rational reduction	88
4.4	For $m = 1$ and $n = 5$ we have this series of propagators necessary to calculate the total evolution	90
5.1	Available Matlab visualization functions in <i>BlochLib</i>	121
5.2	Key examples and implementation programs inside <i>BlochLib</i>	124
6.1	A list of some sub-units for a $C7$ permutation cycle.	156
6.2	Sequence Permutation set for the effective Hamiltonian calculations of the post- $C7$ sequence.	160
6.3	Spin operators and tensors generated to probe the effective Hamiltonians	161
6.4	Spin System parameters for the three sets of permutations. All units are in Hz	161
6.5	Relevant weighting factors for Eq. 6.17	162
6.6	Best $C7$ permutation sequences for each spin system and $C7$ cycle length.	186

Acknowledgments

Ack

None of this thesis would have even existed without the aid of an SUV knocking me off my motor cycle at the beginning of my years in the Pines group. It left my arm in a state of mushy goo for 6 months. With only my left (not my ‘good’ arm) functioning I had to leave the experimental track I had started and venture into the only thing I could do, type. From that point on, the CPU was inevitable. So to this yokel, I give my estranged thanks.

Nowledge

To say that one finished anything here without any help would be a nasty lie. Those many years staring at a computer screen have made me appreciate the comments and discussions from those who do not. Their constant volley of questions and ‘requests’ give me the impetuous to push my own skills higher. To all those Pine Nuts I have run into, I give my thanks.

There is always something new spewing forth from the voice boxes of the pines folk. In particular Jamie Walls and Bob Havlin seem to always have something new to try. In essence the mathematical background was brought to bare by Jamie as Bob enlightened the experimental side of NMR. From many years of discussion with these two, I have learned most everything I claim to know.

From this point I thank Dr. Andreas Trabesinger for calling to my attention the classical/quantum crossover opening up totally new CPU problems and solutions. John Logan and Dr. Dimitris Sakellariou pushed the development of speed. John’s constant testing and back and forth has helped me improve almost every aspect of my coding life.

Ment

Sadly, I was not able to work with many others in the lab, as it seemed my instrument of choice was not a common NMR tool. It has been a privilege to have had the ability to explore the capabilities of the CPU even if it was not on the main research track of the group. For this I thank Alex Pines. Were it not for him, this exploration and assembly would not have been possible. Alex seems to have an uncanny foresight into peoples capabilities and personalities creating an interesting blend of skills, ideas, and brain power that seem to fuel the everyday life in the lab as well as pushing new thoughts to the end. I only hope to leave something behind for this group to take to the next stage.

S

We must not forget those folks that have constantly dealt with the emotional sideshow that is grad school. During my stay here, my family has suffered many losses, yet still has the strength to support my own endeavors; however crazy and obnoxious they made me act towards them. One cannot forget the friends as well; Dr. P, Sir Wright, Prof. Brown and ma'am Shirl have been around for many ages and are always a breath of clean, cool air and patience. Were it not for all friend and family, I certainly would not be at this point



So I thank all y'all.

Chapter 1

Introduction

Before the arrival of the computer, analytic mathematical techniques were the only methods to gain insight into physical systems (aside from experiment of course). This limited the scale of the problems that could be solved. For instance, there are few analytic solutions to Ordinary Differential Equations (ODEs) in comparison to the massive number that can be generated from simple physical systems. Nonlinearities in ODEs are extraordinarily hard to treat analytically. Now, computers and simulations have increased the scale, complexity, and knowledge about many systems from nuclear reactions and global weather patterns to describing bacteria populations and protein folding.

The basic function of numerical simulations is to provide insight into theoretical structures, physical systems, and to aid in experimental design. Its use in science comes from the necessity to extend understanding where analytic techniques fail to produce any insight. Numerical techniques are as much an art form as experimental techniques. There are typically hundreds of ways to tackle numerical problems based on the available computer architecture, algorithms, coding language, and especially development cost. Though many

numerical solutions to problems exist, some execute too slowly, others are too complicated for anybody but the creator to use, and still others are not easily extendable.

The basic scientific simulation begins with a theory. The theory usually produces the equations of motion for the system and the simulation task is to evolve a particular system in time. The theory of Nuclear Magnetic Resonance (NMR) is over 50 years[1, 2, 3, 4] strong. The theory is so well developed that simulations have become the corner stone to which all experimental results are measured[5, 6]. This is the perfect setting for numerical simulations. The equations of motion are well established, approximation methods and other simplification techniques are prevalent, and the techniques for experimental verification are very powerful.

Much of the advancement in NMR today comes from the aid provided by numerical investigations (to list single references would be futile, as virtually all NMR publications include a simulation of some kind). Even though there is this wide spread usage of simulation, there is surprisingly little available to assist in the task. This leaves the majority of the numerical formulation to the scientist, when an appropriate tool kit can simplify the procedure a hundred fold. Numerical tool kits are a collection of numerical routines that make the users life easy (or at least easier).

The two largest and most popular toolkits available today are Matlab¹ and Mathematica². These two packages provide a huge number of tools for development of almost any numerical situation. However, they are both costly, slow, and have no tools for NMR applications. Of course it is possible to use these two to create almost any other tool kit, but then the users will have to get the basic programs. Including other toolkits at this level

¹The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, [Mathworks,http://mathworks.com](http://mathworks.com)

²Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL 61820, [Wolfram, http://wolfram.com](http://wolfram.com)

is next to impossible as is creating parallel or distributed programs.

This thesis attempts to collapse the majority of NMR research into a fast numerical tool kit, but because there are over 50 years of mathematics to include, not everything can be covered in a single thesis. However, the presented tool kit here can easily provide a basis to include the rest. After we describe the tool kit, we will show how much easier it is to create NMR simulations from the tiny to the large, and more importantly, how it can be used to aid the ever toiling researcher to develop more and more interesting techniques.

Six chapters will follow this introduction. The second chapter describes the computational knowledge required to create algorithms and code that achieve both simplicity in usage and, more importantly, speed. The third chapter then goes through the various equations of motion for an NMR system in detail. It is these interactions that we need to calculate efficiently and provide the abstract interface. The fourth chapter describes most all the possible algorithmic techniques used to solve NMR problems. The fifth chapter will demonstrate the basic algorithms, data structures, and design issues and how to contain them all into one tool kit called *BlochLib*. The next chapter includes a demonstration of a class of simulations now possible using the techniques developed in previous chapters. Here I investigate the effect of massive permutations on simple pulse sequences, and finally close with several possible future applications and techniques.

Chapter 2

Computer Mechanics

Contrary to almost every other Pines' Lab thesis, this discussion will begin with the fundamentals of computation, rather than the fundamentals of NMR. This discussion is best begun with the bad definition of a Turing Machine from Merriam-Webster Dictionary.

“A hypothetical computing machine that has an unlimited amount of information storage.”

This basically says that a Turing machine is a computational machine, which does not help us at all. What Turing really said is something like the following^[7]. Imagine a machine that can both read and write along one spot on a one dimensional tape divided into sections (this tape can be of infinite length). This machine can move to any section on the tape. The machine has a finite number of allowed states, and the tape has a finite number of allowed values. The machine can read the current spot on the tape, erase that spot and write a new one. What the machine writes and does afterwards is determined by three factors: the state of the machine, the value on the tape, and a table of instructions. The table of instructions is the more important aspect of the machine. They specify for any given state of the machine and value on the tape, what the machine should write on

the tape and where the machine should move to on the tape. This very general principle defines all computations. There is no distinction made between hardware (a physical device that performs computations) or software (a set of instructions to be run by a computing device). Both can be made to perform the same task, however, hardware is typically much faster when optimally designed than software, but in comparison hardware is *very* hard to make. Software allows the massive generalizations of particular ideas and algorithms, whereas hardware suffers the opposite extreme. Our discussions will be limited to software, only introducing hardware where necessary.

A simple example of a two state Turing machine is shown in Figure 2.1. In this very simple Turing machine example, the machine performs no writing, and the instructions change the state of the machine and move the machine. The lack of an instruction for a possible combination of machine state (B) and tape value (0), causes the machine to stop.

This particular example does not do much of anything except demonstrate the basic principles of a Turing machine. To demonstrate a Turing machine's instruction set for even simple operations (like multiplication or addition) would take a few pages, and is beyond the scope here¹. Once a useful set of instructions is given, we can collapse the instructions into a single reference for another Turing machine to use. A function is now born. To be a bit more concrete, a function is a reference to a set of independent instructions.

Of course, writing complex programs using just a Turing machine instruction set is very hard and tedious. When computers first were born, the Turing machine approach was how computer programming was actually performed. One can easily see that we should be able to represent a function by a simple name (i.e. `multiply`), if we had some translator take

¹A good place to find more Turing machine information, including a Turing machine multiplication instruction set is at this web address <http://www.ams.org/new-in-math/cover/turing.html>.

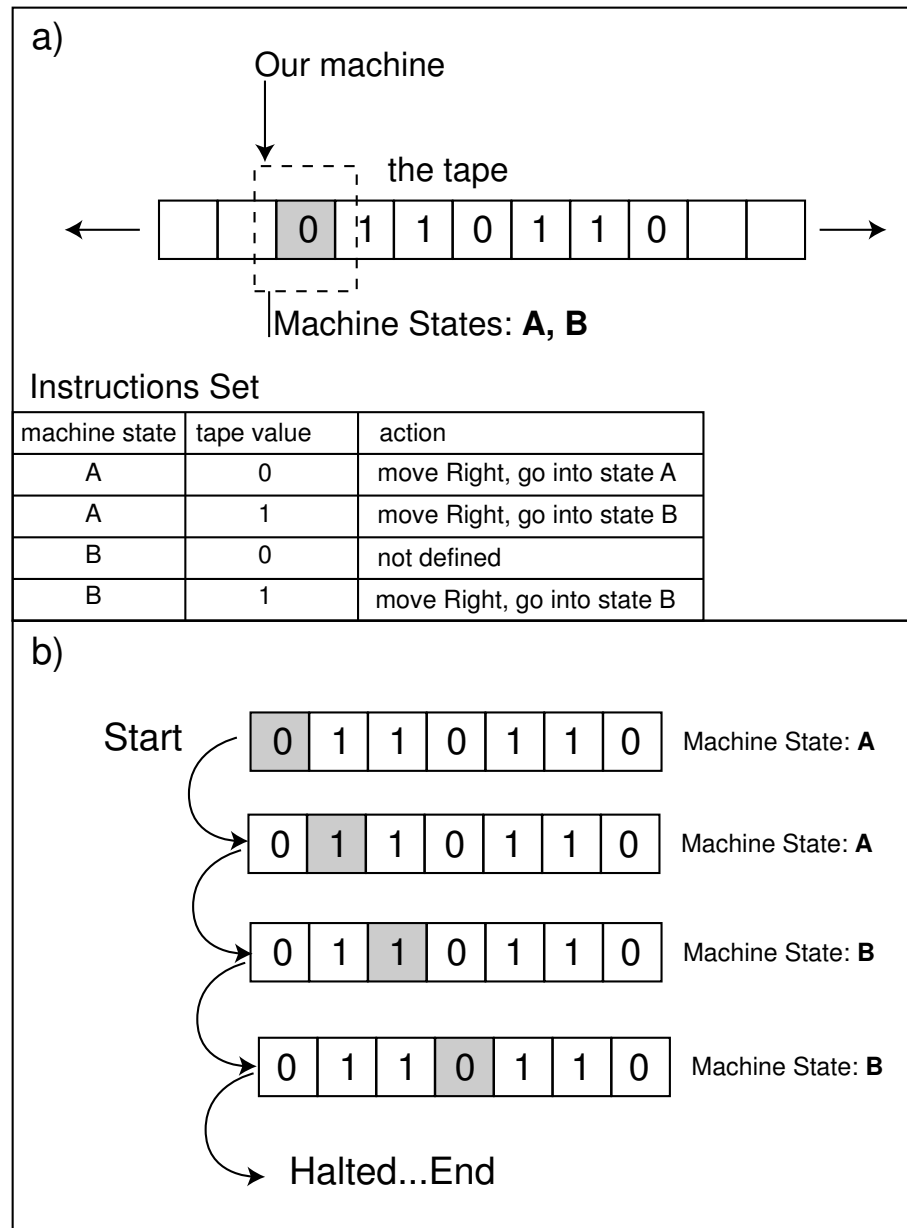


Figure 2.1: A two state Turing machine. The current machines position is represented by the gray box, the tape inputs values can be 0 or 1, and the machine states can be **A** or **B**. The instruction set is designed to stop because one of the four possible combinations of states and inputs is undefined.

our function name and write out the Turing machine equivalent, we could spend much less time and effort to get our computer to calculate something for us. A compiler is such an entity. It uses a known language (at least known to the compiler, and learned by the user), that when the compiler is run, translates the names into working machine instructions. Compilers and their associated languages are called *High Level Languages*, because there is no need for a user to write in the low level machine instruction set.

Programming languages can then be created from a set of translation functions. Until the development of programming languages like C++, many of the older languages (Fortran, Algol, Cobal) were only “words” to “machine-code” translators. The next level of language would be the function of functions. These would translate a set of functions into a series of functions then to a machine code level. Such a set of functions and actions are now referred to as a *class* or an *object*, and the languages C++ and Java are such languages. The next level, we may think, would be an object of objects, but this is simple a generality of an object already handled by C++ and Java. For an in depth history of the various languages see Ref. [8]. For a history of C++ look to Ref. [9].

2.1 Data Types

Besides simple functions, high level languages also provide basic data types. A data type is a collection of more basic data types, where the most basic data type for a computer is a binary value (0 or 1), or a bit. Every other data type is some combination and construction of the bit. For instance a byte is simple the next smallest data type consisting of eight bits. Table 2.1 shows the data available to almost all modern high level languages.

Table 2.1: Basic High Level Language Data Types

Name	Composition
bit	None, the basic block
byte	8 bits
character	1 byte
integer	2 to 4 bytes
float	4 bytes
double	8 bytes

The languages also define the basic interactions between the basic data types. For example, most compilers will know how to add an integer and a float. Beyond these basic types, the compiler knows only how to make functions and to manipulate these data types.

In current versions of Fortran, C and most other modern languages, the language also gives one the ability to create their own data types from the basic built in ones. For example we can create a **complex** data type composed of two **floats** or two **doubles**, then we must create the functions that manipulate this data type (i.e. addition, multiplication, etc.).

Suppose we wish to have the ability to mix data types and functions: creation of a data type immediately defines the functions and operations available to it, as well as conversion between different data types. These are what we referred to as *objects* and are the subject of the next section.

2.2 The Object

Scientific computation has seen much of its life stranded in the abyss of Fortran. Although Fortran has come a long way since its creation in the early 1950s, the basic syntax and language is the same. Only the basic data types (plus a few more) shown in Table 2.1 are allowed to be used, and creation of more complex types are not allowed.

The functions and function usage are typically long and hard to read and understand². Its saving grace is that it performs almost ideal machine translation, meaning it is fast (few unnecessary instructions are used during the translation). Given the scientific need for speed in computation, Fortran is still the choice today for many applications. However, this all may change soon due to fairly recent developments in C++ programming paradigms.

2.2.1 Syntax

Before we can go any further, it is necessary to introduce some syntax. Throughout this document, I will try to present actual code for algorithms when possible. As it turns out, much of the algorithmic literature uses “pseudo-code” to define the working procedures for algorithms. Although this usually makes the algorithm easier to understand, it leaves out the details that are crucial upon implementation of an algorithm. The implementation determines the speed of the algorithms execution, and thus its overall usefulness. Where appropriate, both the algorithmic steps and actual code will be presented.

The next several paragraphs will attempt to introduce the syntax of C++ as it will be the implementation language of choice for the remainder of this document. It will be short and the reader is encouraged to look towards an introductory text for more detail (Ref. [10] is a good example of many). Another topic to grasp when using C++ is the idea of inheritance. This is not discussed here, but the reader should look to Ref. [11] as inheritance is an important programming paradigm. It will be assumed that the reader has had some minor experience a very high level language like Matlab.

- The first necessary fact of C++ (and C) is declaration of data types. Code Example

2.1 declares an integer data type, that can be used by the name `myInt` later on.

²Look to the Netlib repository, <http://netlib.org> for many examples of what is claimed here.

Code Example 2.1 Integer declaration

```
int myInt;
```

- The definition of functions requires a return type, a name, and arguments where both the return type and the arguments must be valid data types as shown in Code Example 2.2. In code example 2.3 the `Return_T` is the return data type, `Arg_T1` through `Arg_TN`

Code Example 2.2 Function declarations: general syntax

```
Return_T functionname(Arg_T1 myArg1, ..., Arg_TN myArgN)
```

are the argument data types. For example, in Code Example 2.3 is a function that adds two integers.

Code Example 2.3 Function declarations: specific example

```
int addInt(int a, int b)
{ return a+b; }
```

- Pointers (via the character ‘*’) and references (via the character ‘&’) claim to be what they say: Pointers point to the address (in memory) of the data type, and references are aliases to an address in memory. The difference between them illustrated in the example in Code Example 2.4.
- Creating different data types can be performed using a `class` or `struct`. A complex number data type is shown in Code Example 2.5. The above example shows the syntax for both creation of the a data type and how to access its sub elements.
- Templates allow the programmer to create generic data types. For instance in the `class complex` example in Code Example 2.5, we assigned the two sub elements to a `double`. Suppose we wanted to create one using a `float` or an `int`. We do not

Code Example 2.4 Pointers and References

```
//declare a pointer
int *myPoinerToInt;
//assign it a value
//the '*' now acts to extract the memory
// not the address
*myPoinerToInt=8;

//declare an integer
int myInt=4;

//this will print '4 8'
cout<<myInt<<" "<<*myPoinerToInt<<endl;
//make out pointer above, point to this new integer
// using the reference
myPoinerToInt=&myInt;

//now when we change 'myInt' BOTH objects will change
myInt=10;
//this will print '10 10'
cout<<myInt<<" "<<*myPoinerToInt<<endl;
```

Code Example 2.5 Object declaration Syntax

```
class complex{
public:
//a complex number contains has two real numbers
double real;
double imag;
//The constructor defines how to create a complex number
complex():
real(0), image(0) {}
//The constructor defines how to create a complex number
//with input values
complex(double r, double i):
real(r), image(i) {}
};

//here we use the new data type
complex myCmx(7,4);
//this will print '7+i4'
cout<<myCmx.real<<"i"<<myCmx.imag<<endl;
```

wish to create a new class for each type, instead we can template the class as in Code

Example 2.6. In C++ we can template both classes and the arguments of functions.

Code Example 2.6 Template Objects

```

template<class Type_T>
class complex{
public:
//a complex number contains has two real numbers
    Type_T real;
    Type_T imag;
//The constructor defines how to create a complex number
    complex():
        real(0), image(0) {}
//The constructor defines how to create a complex number
//with input values
    complex(Type_T r, Type_T i):
        real(r), image(i) {}
};

//here we use the new data type
// use a double as the sub element
complex<double> myCmx(7,4);
//this will print ‘‘7+i4’’
cout<<myCmx.real<<"i"<<myCmx.imag<<endl;

// use a int as the sub element
complex<int> myCmxInt(7,4);
//this will print ‘‘7+i4’’
cout<<myCmxInt.real<<"i"<<myCmxInt.imag<<endl;

```

This template procedure allows the creation of a wide range of generic data types and function that operate over a large range of data types without having to code a different function or object for each different combination of data types. In Fortran, one must code a different function for each different data type making the creation of general algorithms tedious[12]. Given M data types, and N functions, using templates can in principle reduce the $O(M \times N)$ number of procedures in a Fortran environment to $O(N + M)$ procedures.

Given those simple syntax rules, we can move forward to explain the object and the power that resides in a templated object.

2.3 Expression Templates

2.3.1 Motivations

Until recently[13], C++ has been avoided for scientific computation because of an issue with speed. We have shown how to create an object, but we can also create specific functions, or operators, that define the mathematics of the object. Let us revisit the `class complex` example and define the addition operator. We also must define the assignment (`'='`) operator before we can define an addition operator as shown in Code Example 2.7. Now we can use our addition operator to add two complex numbers. The addition operator

Code Example 2.7 Defining operators

```
template<class Type_T>
class complex{
    public:
        //define the sub elements
        ....
        //define the assignment operator
        //an INTERNAL CLASS FUNCTION
        complex operator=(complex a){ real=a.real; imag=a.imag;}
};

template<class Type_T>
complex<Type_T> operator+(complex<Type_T> a, complex<Type_T> b)
{
    return complex<Type_T>(a.real+b.real, a.imag+b.imag);
}
```

(and any others we define) can be nested into a long sequence as shown in Code Example 2.9.

Code Example 2.8 simple addition

```
complex<double> A(4,5), B(2,3), C;  
C=A+B;  
//this will print ‘‘6+i8’’  
cout<<C.real<<"i"<<C.imag<<endl;
```

Code Example 2.9 Single operations

```
complex<double> A(4,5), B(2,3), C;  
C=A+B-B+A;  
//this will print ‘‘8+i10’’  
cout<<C.real<<"i"<<C.imag<<endl;
```

2.3.2 Stacks

We should take note as to what the compiler and the computer are doing when it sees an expression like the one in Code Example 2.9. Initially the compiler will attempt to translate our mathematical expression into a *stack*. A *stack* is a list with a last-in-first-out property. The order of the list is determined by the syntax, using standard mathematical rules (e.g. items inside parentheses are treated first, multiplication is performed before addition, etc.). The expression will be parsed from the last element to the first in the sequence, $B+A$, then $B-(\text{result of } (B+A))$, then $A+(\text{result of } (B-(\text{result of } (B+A))))$, finally $C=\text{result of } (A+(\text{result of } (B-(\text{result of } (B+A))))$. Each step represents a stack step, and can be best represented as a stack tree shown in Figure 2.2. After this stack is created, the compiler writes the appropriate instruction set to complete the operation once the program is run. When the program is run, the machine must go to the bottom of the stack and perform each operation as it works its way up the stack tree. Another way to perform the same operations shown in Code Example 2.9 is to follow the exact stack tree, in the code itself as shown in Code Example 2.10.

It is then easy to see that in the process of using the operators we necessitate the

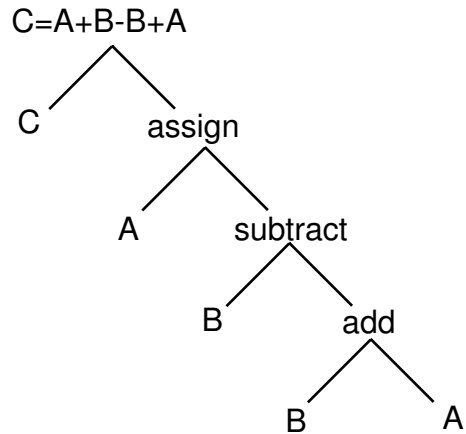


Figure 2.2: A simple stack tree

Code Example 2.10 Code representation of a stack tree

```

complex A(4,5), B(2,3), C;
complex tmp1=B+A;
complex tmp2=B-tmp1;
C=A+tmp2;

```

use of temporary objects. For individual data types (doubles, floats, ints, and our complex example), there is no way around this fact³. But for arrays of values, we can potentially create a much more optimal situation.

2.3.3 An Array Object and Stacks

First we shall define a templated `Vector` class so that we can continue our discussion. The vector class shown in Code Example 2.11-2.12 maintains a list of numbers and defines appropriate operators for addition, multiplication, subtraction, and division of two `Vectors`.

The code examples in Code Example 2.11 also gives the definitions for element

³There is no easy way to see how such a stack tree can be simplified. However, the ever increasing complexity of microchip architectures are actually creating new instruction sets that give the compiler the ability to, for example, add and multiply two numbers under the same instruction as in a PowerPC chip. The complex functions like `sin` and `cos` are now included on the microchips instruction set which then increase the speed of the produced code by reducing the stack tree length.

Code Example 2.11 a simple Template Vector class

```
template<class T>
class Vector{
private:
    T *data_;
    int len_;
public:
    Vector():data_(NULL), len_(0){}
    Vector(int len, T fillval=0)
    {
        data_=new T[len];
        len_=len;
        for(int i=0;i<len;++i)
            { data_[i]=fillval; }
    }
    //this is the 'destructor' or how we free the memory
    // after we are done with the Vector
    ~Vector()
    { if(data_!=NULL) delete [] data_; }
    Vector &operator=(Vector rhs)
    {
        if(data_!=NULL) delete [] data_;
        data_=new T[rhs.size()];
        len_=rhs.size();
        for(int i=0;i<len;++i)
            { data_[i]=rhs(i); }
        return *this;
    }
    T &operator()(int i){ return data_[i]; }
    T &operator[](int i){ return data_[i]; }
    int size(){ return len_; }
};
```

Code Example 2.12 a simple Template Vector operations

```
template<class T>
Vector<T> operator+(Vector<T> a, Vector<T> b)
{
    Vector c(a.size());
    for(int i=0;i<len;++i)
        { c[i]=a[i]+b[i]; }
}

template<class T>
Vector<T> operator-(Vector<T> a, Vector<T> b)
{
    Vector c(a.size());
    for(int i=0;i<len;++i)
        { c[i]=a[i]-b[i]; }
}

template<class T>
Vector<T> operator/(Vector<T> a, Vector<T> b)
{
    Vector c(a.size());
    for(int i=0;i<len;++i)
        { c[i]=a[i]/b[i]; }
}

template<class T>
Vector<T> operator*(Vector<T> a, Vector<T> b)
{
    Vector c(a.size());
    for(int i=0;i<len;++i)
        { c[i]=a[i]*b[i]; }
}
```

access (the `operator()(int)` and `operator[](int)`) as well as a way to determine how long the `Vector` is (the `int size()` function). The destruction (the `~Vector()`) function is also important as it frees the memory used by the vector. Also note that in the examples there are no error checking on the sizes of the vectors when we perform an operation. Such checks are easy to implement, but add clutter to the code, so they will be left out here.

A simple expression using our new object is shown in Code Example 2.13. Using

Code Example 2.13 a simple vector expression

```
Vector<double> a(5,7), b(5,8), c(5,9), d(5,3);
d=c+b+b-a;
```

our stack representation, we can also write the example in Code Example 2.13 as the stack produced code as shown in Code Example 2.14. In the example in Code Example 2.14 we

Code Example 2.14 a simple vector expression as it would be represented on the stack.

```
Vector<double> a(5,7), b(5,8), c(5,9), d(5,3);
Vector<double> t1(5), t2(5), t3(5);
int i=0;
for(i=0;i<d.size();++i)
{ t1[i]=b[i]-a[i]; }
for(i=0;i<d.size();++i)
{ t2[i]=b[i]+t1[i]; }
for(i=0;i<d.size();++i)
{ t3[i]=c[i]+t2[i]; }
for(i=0;i<d.size();++i)
{ d[i]=t3[i]; }
```

could have both saved the temporary vectors (`t1`, `t2`, and `t3`), as well as the final assignment loop. In general, however, this optimization is not possible for the compiler to see, and this example is an accurate representation of the expression `d=c+b+b-a`. An experienced programmer could easily reduce everything to a single loop requiring no temporary vectors as shown in Code Example 2.15. This case is at least a factor of 3 faster than the previous

Code Example 2.15 a simple vector expression in an optimal form.

```
Vector<double> a(5,7), b(5,8), c(5,9), d(5,3);  
for(int i=0;i<d.size();++i)  
{ d[i]=c[i]+b[i]+b[i]-a[i]; }
```

case in Code Example 2.14 (it is a even faster the three because we did not have to create the temporaries). It is for this reason that C++ has been avoided for scientific or other numerically intensive computations. One may as well write a single function that performs the specific optimal operations of vectors (or any other array type). In fact the Netlib⁴ is full of such specific functions.

2.3.4 Expression Template Implementation

A few years ago Todd Veldhuizen developed a technique that uses templates to trick the compiler into creating the optimized case shown in Code Example 2.15 from a simple expression like the one shown in Code Example 2.13[14]. This technique is called *expression templates*. Because the technique is a template technique, it is applicable to many data types without much alteration.

This trickery with templates began with Erwin Unruh when he made the compiler itself calculate prime numbers[15]. He could do this because for templated objects to be compiled into machine code, they must be *expressed*, or they must have a real data type replace the template argument (as in our examples of using the `Vector` class with the `double` replacing the `class T` argument). The code that generated the prime numbers can be found in Appendix A. In fact Erwin showed that the compiler itself could be used as Turing machine (albeit a very slow one).

Now we can describe the technique in painful detail. It uses fact that any template

⁴See <http://netlib.org>

augment must be expressed before it can be used. To allow a bit of ease in the discussion we will assume that only one data type, the `double`, is inside the array object⁵.

We will restrict ourselves to the `Vector`, as most other data types are simply extensions to a vector type. Second, in our discussions, we will restrict the code to the addition operation, as other operations are easily implemented in exactly the same way. A better definition of what we wish to accomplish is given below.

Given an arbitrary right-hand-side (rhs) of a given expression, a single element on the left-hand-side (lhs) should be able to be assignable by only one index reference on the rhs.

This statement simply means that the entire rhs should be collapsible into one loop. But the key is in the realization that we require the index for both the lhs and the rhs. The beginning is already given, namely the `operator()(int i)` function shown in Code Example 2.11. The remaining task is to figure out how to take an arbitrary rhs and make it indexable by this operator.

We can analyze the inside of the operators in Code Example 2.12. Notice that they are binary operations using a single index, meaning they require two elements to perform correctly (the `a[i]` and `b[i]` with the index `i`). A new object can be created that performs the binary operation of the two values `a[i]` and `b[i]` as shown in Code Example 2.16. The addition operation has been effectively reduced to a class, which means the operation can be templated into another class. The reason why the `apply` function is `static`⁶ will be come apparent in the Code Example 2.17. The class, `VecBinOp`, in Code Example 2.16 does not give us the single index desired. The class shown in Code Example 2.17 does. `VecBinOp` stands for a Vector-Vector binary operation. Note that the object is

⁵We can perform more generic procedures if we use the `typedef`. A `typedef` is essentially a short cut to naming data types. For instance if we had a data type that was templated like `Vector<Vector<double> >` we could create a short hand name to stand for that object like `typedef Vector<Vector<double> > VMat;`

⁶A `static` function or variable is one that never changes from any declaration of the object.

Code Example 2.16 A Binary operator addition class

```
class ApAdd {
public:
    ApAdd() { }
    static double apply(double a, double b)
    { return a + b; }
};
```

created by creating pointers to the input vectors *not* copying the vectors. This object takes three template arguments, the two vector types and the operation class. One may wonder

Code Example 2.17 A Binary operator class

```
template<class V1, class V2, class Op>
class VecBinOp{
private:
    V1 *vec1;
    V2 *vec2;
public:
    VecBinOp(V1 &a, V2 &b):
        vec1(&a), vec2(&b){}
    ~VecBinOp(){ vec1=NULL; vec2=NULL; }
    //requires 'Op::apply' to be static
    // to be used in this way
    double operator()(int i)
    { return Op::apply(vec1(i), vec2(i)); }
};
```

why we templated the two vector class `V1` and `V2` as we know we are dealing with only `Vector<double>` objects, the reasons for this will be clear below. Our object creates the desired single index operator; however, we are far from finished. We could use the `VecBinOp` alone, to create our new addition operator as shown in Code Example 2.18. This addition operator did nothing more than make that code more complex, and actually slowed down the addition operation because of the creation of the new `VecBinOp` object, and it does not allow us to nest multiple operations (e.g. `d=a+b+c`) with any improvement. But we are a step closer to realizing our goal and we wish to nest the template arguments and

Code Example 2.18 A bad expression addition operator

```
template<class V1, class V2>
Vector &operator+(V1 &a, V2 &b)
{
    Vector<double> out(a.size());
    VecBinOp<V1, V2, ApAdd> addObj(a,b,ApAdd());
    for(int i=0;i<a.size();++i)
    { out(i)=addObj(i); }
    return out;
}
```

not the operations themselves. In order to nest the template operations, we need to create another object that can maintain the binary operation in name only (e.g. `VecBinOp<V1, V2, ApAdd>`), then use this name to pass to the next operation. Such an object is shown in Code Example 2.19. This new object gives use the ability to pass an arbitrary expression

Code Example 2.19 A simple Vector Expression Object

```
template<class TheExpr>
class VecExpr{
private:
    TheExpr *expr;
public:
    VecExpr(TheExpr &a):
        expr(&a){}

    double operator()(int i)
    { return expr(i); }
};
```

around as an object, but not evaluating the expression. The expression is only evaluated when the `operator()(int)` is called. Thus we can delay the evaluation until have an assignment. This object can then be passed back to the `VecBinOp` object as a template argument (the reason why we left the ‘`Vector`’ template input for `VecBinOp` as a template argument and not directly assigned it to the `Vector`). Now we can rewrite out addition operator to simply pass back the `VecExpr` object as shown in Code Example 2.20. Now the

Code Example 2.20 A good expression addition operator

```
template<class Expr_T1, Expr_T2>
VecExpr< VecBinOp<Expr_T1,Expr_T2, ApAdd> >
  operator+(Expr_T1 &a, Expr_T2 &b)
{
  return VecExpr<
    VecBinOp<Expr_T1,Expr_T2, ApAdd>
    >(a,b, ApAdd());
}
```

addition operation does not evaluate any arguments, it simply passes a staging expression that we will need to find another means to evaluate. This new addition operator can be used for any combination of `Vector` or `VecExpr` objects. It can also be used for *any* object as well, but it will more than likely give you *many* errors because of conflicts of data types. For instance there is not `operator()(int)` defined for a simple double number, thus the compiler will give you an error. The best method around this problem is to create a quadruple of operators using the more specific objects as shown in Code Example 2.21. Here, we *partially express* the templates to show that they are only for `Vector`'s and `VecExpr`'s. Now we have any rhs that will be condensed into a single expression. The final step is the evaluation/assignment. Since all the operators return a `VecExpr` object, we simply need to define an assignment operator (`operator=(VecExpr)`). Assignments can only be written internal to the class, so inside of our `Vector` class in Code Example 2.11 we must define this operator as shown in Code Example 2.22. Besides the good practice checking the vector sizes and generalization to types other than doubles, this completes the entire expression template arithmetic for adding a series of vectors. It is easy to extend this same procedure for the other operators (`-`, `/`, `*`) and unary types (`cos`, `sin`, `log`, `exp`, etc.) where we would create a `VecUniOp` object. Now that we have a working expression template structure, we can now show in Figure 2.3 what the compiler actually performs upon compilation of an

Code Example 2.21 A quadruple of addition operators to avoid compiler conflicts.

```
//Vector+Vector
template<class Expr_T2>
VecExpr< VecBinOp<Vector<double>,Vector<double>, ApAdd> >
  operator+(Vector<double> &a, Vector<double> &b)
{
  return VecExpr<
    VecBinOp<Vector<double>,Vector<double>, ApAdd>
    >(a,b, ApAdd());
}

//Vector+VecExpr
template<class Expr_T2>
VecExpr< VecBinOp<Vector<double>,VecExpr<Expr_T2>, ApAdd> >
  operator+(Vector<double> &a, VecExpr<Expr_T2> &b)
{
  return VecExpr<
    VecBinOp<Vector<double>,VecExpr<Expr_T2>, ApAdd>
    >(a,b, ApAdd());
}

//VecExpr+Vector
template<class Expr_T1>
VecExpr< VecBinOp<VecExpr<Expr_T1>,Vector<double>, ApAdd> >
  operator+(VecExpr<Expr_T1> &a, Vector<double> &b)
{
  return VecExpr<
    VecBinOp<VecExpr<Expr_T1>, Vector<double>,ApAdd>
    >(a,b, ApAdd());
}

//VecExpr+VecExpr
template<class Expr_T1, class Expr_T2>
VecExpr< VecBinOp<VecExpr<Expr_T1>,VecExpr<Expr_T2>, ApAdd> >
  operator+(VecExpr<Expr_T1> &a, VecExpr<Expr_T2> &b)
{
  return VecExpr<
    VecBinOp<VecExpr<Expr_T1>,VecExpr<Expr_T2>,ApAdd>
    >(a,b, ApAdd());
}
```

Code Example 2.22 An internal `VecExpr` to `Vector` assignment operator

```
template<class Expr_T>
Vector &operator=(VecExpr< Expr_T > &rhs)
{
    for(int i=0;i<size();++i)
    { this->operator(i)=rhs(i); }
    return *this;
}
```

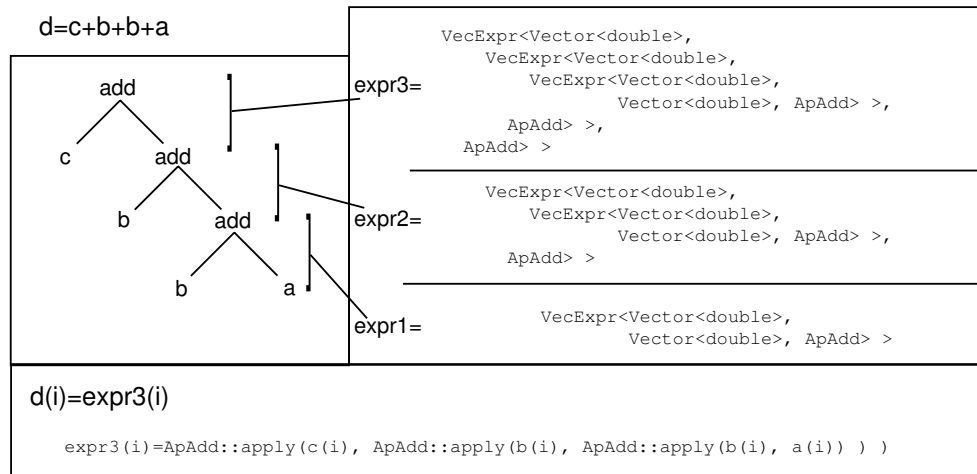


Figure 2.3: How the compiler unrolls an expression template set of operations.

expression such as `d=c+b+b+a`. This technique is not limited to vectors but also matrices and any other indexable data type; all one has to do is change the `operator()` to the size and the index type desired.

To show the actual benefit of using the expression templates, Figure 2.4 shows a benchmark for performing a *DAXPY* (Double precision A times X Plus Y) for a variety of languages and programming techniques. You can see from the figure that the results are comparable to a highly optimized Fortran version. The degree of matching depends greatly on the compiler and the platform. The data in the figure is using `gcc-3.2.1` under the Cygwin environment, under Linux (Red Hat 7.3) the results match even better. From the figure it is apparent that if the size of the vector is known and fixed before the code

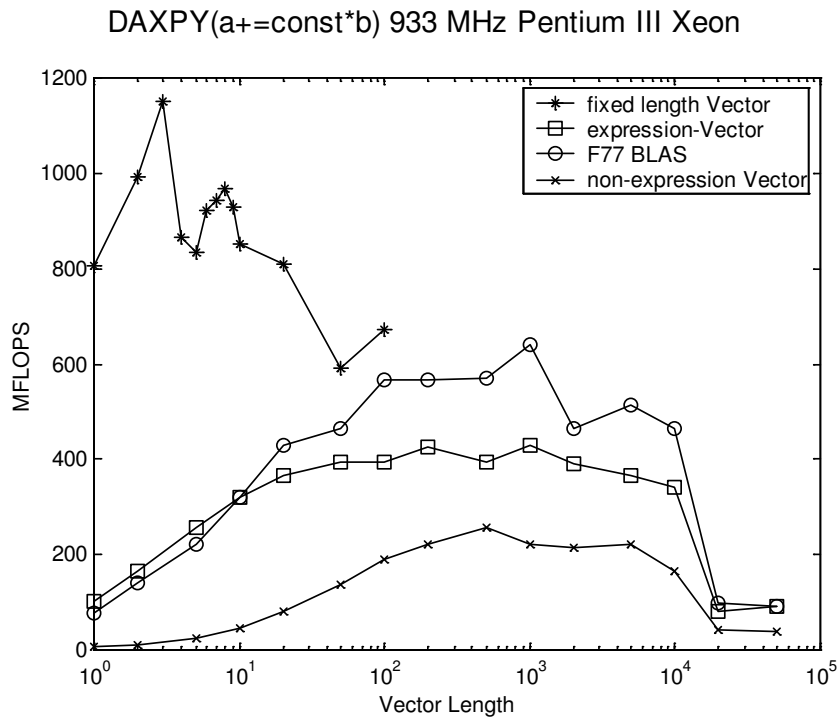


Figure 2.4: Double precision A times X Plus Y , *DAXPY* benchmarks in Millions of Floating Point operations per Second (MFLOPS) for a fixed length expression template vector (*), the basic expression template vector (the box), the optimized Fortran 77 routine (o) and the normal non-expression template vector (x). All code was compiled under the Cygwin environment using gcc-3.2.1.

is compiled, then we can perform even further optimizations using the template structures. This technique is called *meta-programming*[16, 17, 18, 19] and exploits the compilers ability to be a Turing machine as in the example in Appendix A.1.1. An example meta-program for unrolling fixed length vectors is shown in Appendix A.1.2. More about template based programming can be found in Ref. [20].

There are, however, situations where this simple expression unrolling does not improve the speed. Such operations typically require the use of a *workspace*; they require the use of temporary data structures. This type of optimization is the topic of the next section.

2.4 Optimizing For Hardware

Expression templates provide a nice technique for reducing complex expressions into a single expression allowing simlare speed of a hand produced reduction, but still maintain the powerful ease and readability of the produced code.

Consider the matrix multiplication⁷. Figure 2.5 depicts a representation of a matrix multiplication. To compute each element in the resulting matrix, an entire row of the first matrix and an entire column of the second matrix is needed. We can implement a simple matrix multiplication via the Code Example 2.23. Assume that we have defined a `matrix<T>` class already, so we can perform some speed tests using our simple algorithm. We will stick to square matrices (the most common case, and basically the only case in NMR) for our speed test. The results on a 933 MHz Pentium III using gcc-3.2.1 is shown in Figure 2.6. A Basic matrix multiplication takes N^3 operations where the matrix is of

⁷A tensor multiplication, not the element-by-element multiplication. The element-by-element case is handled well by the expression templates.

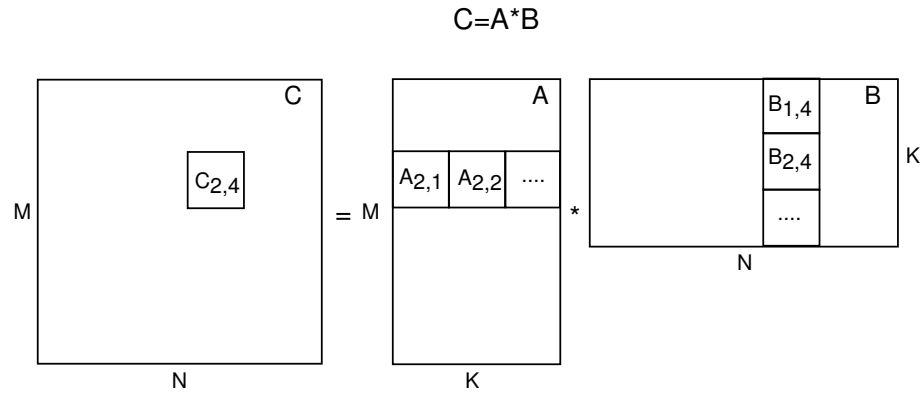


Figure 2.5: A pictorial representation for the matrix–matrix tensor multiplication, $C=A*B$.

The sub box indicates the required elements from each matrix to compute one element in the resulting matrix C.

Code Example 2.23 Simple tensor matrix multiplication

```

template<class T>
matrix<T> operator*(matrix<T> &a, matrix<T> b)
{
  matrix<T> c(a.rows(), b.cols());
  int i,j,k;
  for(i=0;i<a.rows();++i){
    for(j=0;j<b.cols();++j){
      c(i,j)=a(i,0) * b(0,j);
      for(k=1; k<b.cols();++k){
        c(i,j)+=a(i,k) * b(k,j);
      }
    }
  }
  return c;
}

```

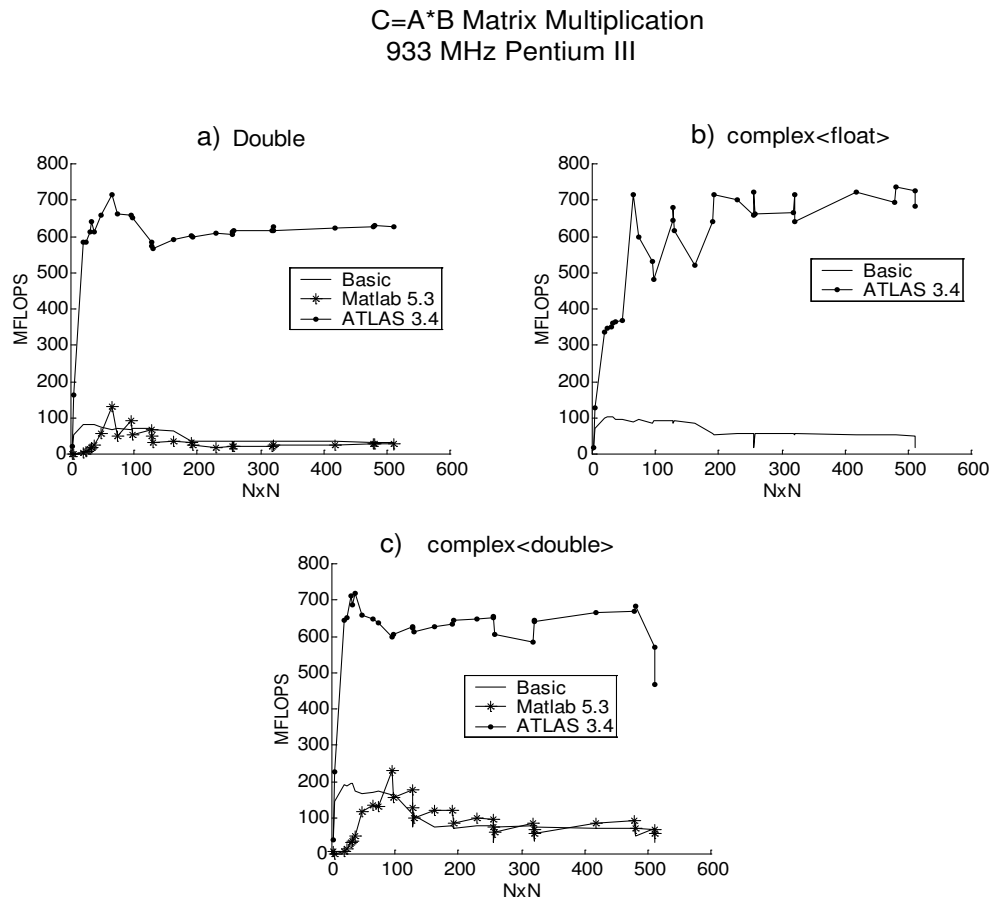


Figure 2.6: Speed in MFLOPS of a double(a), complex <float> (b) and complex < double > (c) matrix-matrix multiplication ($C=A*B$).

this size $N \times N$. A complex matrix multiplication is actually 4 separate non-complex multiplications ($C_r = (A_r * B_r)$, $C_{r+} = (A_i * B_i)$, $C_i = (A_r * B_i)$, $C_{i+} = (A_i * B_r)$). Also shown in Figure 2.6 is the matrix multiplication from another library called ATLAS[21] and the algorithm inside Matlab version 5.3. The ATLAS library is enormously faster and approaches the theoretical maximum for the 933 MHz processor of 933 MFLOPS. Matlab does not have a float as a precision value, so those speed tests are not performed. In all cases the ATLAS algorithm performs an order of magnitude better. How does ATLAS actually perform the multiplication this much faster?

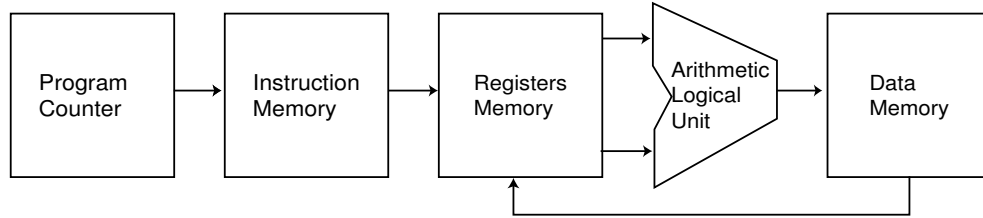


Figure 2.7: A generic computer data path.

The answer is buried deep in the computer architecture. So before we can continue with the explanation, we must first describe a generic computer. The discussion in the following sections are not thorough by any means, they simply are designed to show how one can manipulate programs to use the full potential of specific computer architectures. A good place to learn more nasty details is from Ref. [22].

2.4.1 Basic Computer Architecture

The Data Path

To most programmers, the computer architecture is a secondary concern with algorithms and designs taking precedent. However, Figure 2.6 demonstrates clearly that for even simple algorithms, ignoring the architecture can reduce overall performance by orders of magnitude. For numerically intense programs, this can be the difference in waiting days as apposed to weeks for simulations to finish. To get the most optimum performance from a computer architecture, we must know how the computer functions on a relatively basic level. Figure 2.7 shows a simple generic layout of a Central Processing Unit's (CPU) data path. The data path is the flow of a single instruction, where an instruction tells the computer what to do with selected data stored in memory (things like add, multiply, save, load, etc.). The data path shown in Figure 2.7 is based on the figures and discussion in Ref. [22].

Each element in the data path shown in Figure 2.7 can be implemented in a

variety of different ways giving rise to the production of many different brands (Intel, RISC, PowerPC, etc.). The data path for each of the various CPU's can be described in much the same way based on the simple fact that both data and instructions can be represented as numbers.

- **Program Counter**—This element controls which instruction should be executed and takes care of jumps (function calls) or branches (things like if/else statements).
- **Instruction Memory**—This element holds the number representations of the various instructions the program wishes to perform. The Program Counter then gives the correct address inside the Instruction memory of the instruction to execute.
- **Register Memory**—This element holds 'immediate' data. The immediate data is the data closest to the Arithmetic Logical Unit (ALU) and is the only data that can have any operation performed on it. Thus if a data element is stored in the Data Memory, it must be placed into the Register Memory before an operation on it can occur.
- **Arithmetic Logical Unit (ALU)**—This element is the basic number cruncher of the CPU. It typically takes in two data elements and performs a bit wise operation on them (like add or multiply).
- **Data Memory**—The main data memory of a computer. This can be the RAM (Random Access Memory), a Hard disk, a network connection, etc.

Given a specific architecture, each of the elements in the data path above and the instruction set are fixed entities. A programmer cannot divide two numbers any faster than the data path allows. The most important element of control for the programmer is in what order specific instructions are given.

Programmer Control

There are a number of enhancements to the basic data path described above. In almost every modern processor today there are numerous other hardware additions.

- **pipelines**—This enhancement allows the next instruction to be executed before the previous one has finished. For instance while one instruction is in the ALU, another can be accessing the Register Memory.
- **caches**—The closest memory to the ALU is the fastest memory, caches provide various levels inside the Data Memory that are closer to the ALU, the fastest being closer to the ALU, the slowest farthest away.
- **Single Instruction Multiple Data (SIMD)**—This is called more generically vector processing where more than two data elements can be operated on in one ALU operation. Thus we can add 4 floating point number to 4 another in a single instruction rather than the usual method of 4 instructions for each addition of two floats.

The above list is only partial, but they are the three major features available to a programmer to enhance the speed of a calculation.

Pipelining is easily described in the context of loop unrolling. Many of you may have noticed that in certain codes that there is typically a 4-fold unrolling of `for/do/while` loops (see Code Example 2.24). This 4-fold unrolling may look simply like more typing and added confusion about the algorithm, but this is in fact taking advantage of pipe lining on the processor. In the not unrolled case, the `for` condition (`i<16`) must be evaluated each time before continuing, which is an action that is hard to pipeline because of the dependence on a condition. For the 4-fold unrolled case, not only can each of the four

Code Example 2.24 A simply loop unrolling to a 4 fold pipe line.

```
//length 16 vectors
Vector<double> A(16), B(16), C(16);
//a standard for loop
for(int i=0;i<16;++i){
    C[i]=A[i]+B[i];
}

//a 'loop-unrolled' loop
for(int i=0;i<16;i+=4){
    int i2=i+1,i3=i+2,i4=i+3;
    C[i]=A[i]+B[i];
    C[i2]=A[i2]+B[i2];
    C[i3]=A[i3]+B[i3];
    C[i4]=A[i4]+B[i4];
}
```

operations be pipelined, but the condition testing is reduced by a factor of four. Figure 2.8 shows a pictorial representation of the data path as the loop shown in Code Example 2.24 is run. Some compilers (namely the GNU compiler) perform this sort of loop unrolling automatically when called with optimizations, so writing the fully unrolled loop of the type shown here are becoming a thing of the past. However, if there are more complex data types in the loop or even other branch conditions, the harder it becomes for the compiler to unroll them effectively, so having a good picture of pipelining is still necessary to achieve optimal throughput.

SIMD optimizations are highly system specific and until recently were only available in super computers like Cray system machines. In recent years, consumer CPUs now have these instructions. These instructions act on vectors worth of data at a time, rather than just two elements at a time. They require both special data types and special CPU instructions. Figure 2.9 shows pictorially how a 128 bit SIMD register can be thought of as 4, 32 bit data values. Table 2.2 lists a few of the basic CPUs and there available SIMD

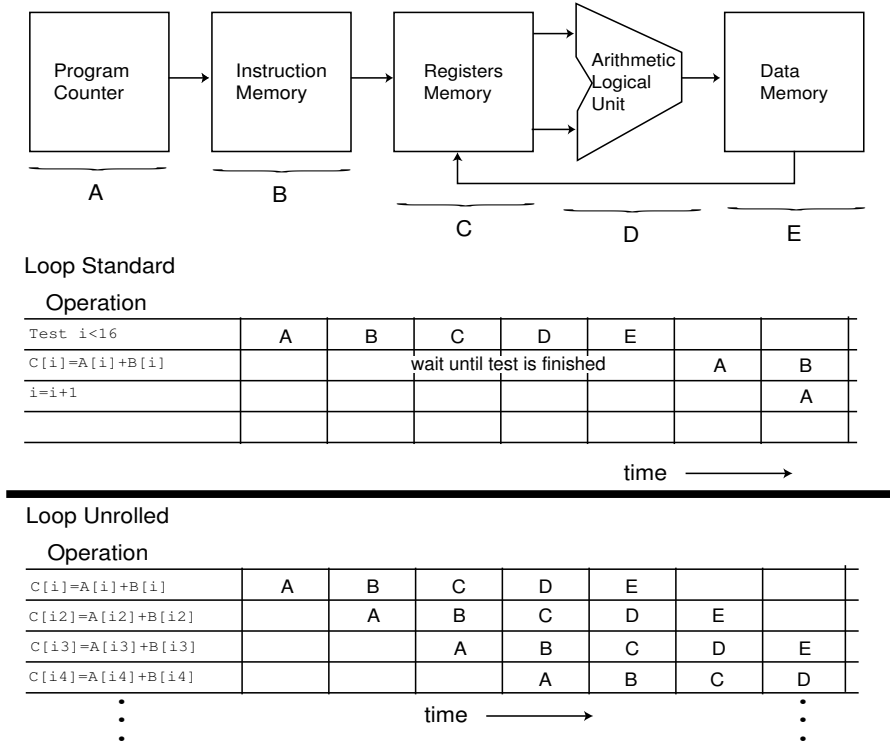


Figure 2.8: Pipe lines and loop unrolling

data types.

Programming using the SIMD types is almost never portable to other CPUs. It may be up to the compiler to attempt to use the SIMD where it can, but currently most compilers are not able to optimize for these registers. As a result programming using SIMD

Table 2.2: SIMD registers available of common CPUs

Architecture	SIMD size	number of common data types
Intel Pentium II MMX	64 bit	4 ints (only int)
Intel Pentium III SSE1	64 bit	4 ints, 2 floats
Intel Pentium IV SSE2	128 bit	8 ints, 4 floats
AMD K5 3Dnow!	64 bit	4 ints, 2 floats
AMD K6 3Dnow2!	128 bit	8 ints, 4 floats
Motorola G4	128 bit	8 ints, 4 floats
Cray J90	64 bit	4 ints, 2 floats
Fujitsu VPP300	2048 bit	128 ints, 64 floats

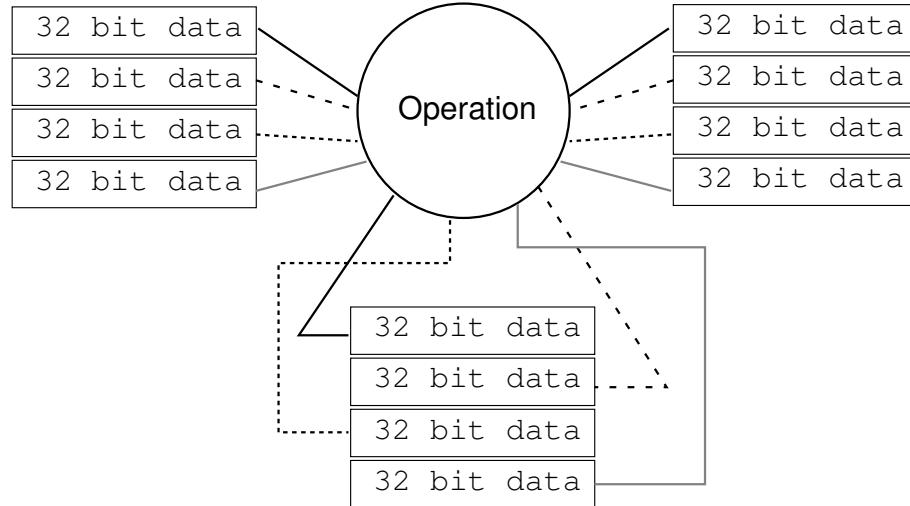


Figure 2.9: A 128 bit SIMD registers made of 4-32 bit data values

tends to be limited to a specific CPU and up to the programmer.

The final optimizing technique involves caching. Caching turns out to be one of the more important aspect in optimizing for modern CPUs. The reason for this is based on the ever growing speed difference between memory access and CPU clock speeds. For instance a 2GHz Pentium IV processor can only access the main data memory (RAM) at rate less then 400 MHz, meaning that while the CPU waits for the data element to arrive from memory, over 5 CPU cycles were wasted doing no work. In actuality the number is much higher because the data element must be found in RAM then sent back.

For large continuous data structure like vectors or matrices, if each element took multiple cycles simply to retrieve and save, calculations would be exceedingly inefficient. Caches, however, provide a method to increase performance using the spatial and temporal locality of a program. This simply means that data just accessed will probably be accessed again soon, and more then likely, the data next to the one just accessed will also be accessed soon. Thus caches tend to load blocks of memory at a time with the hope that the data elements within the block will also be used. Figure 2.10 shows the various levels of caching

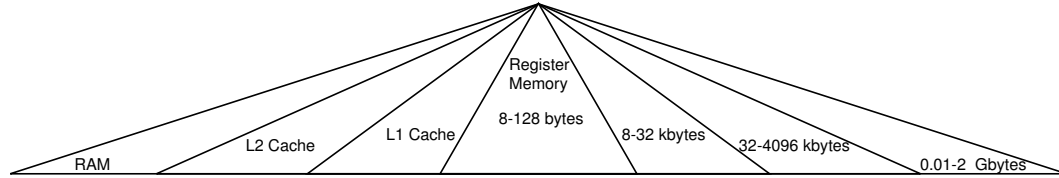


Figure 2.10: Cache levels in modern Processors

available to most computers today. Level 1 (L1) cache is the smallest ranging in size from 8 kb-64 kb but is the fastest with access times very close to the internal CPU Register Memory. Level 2 (L2) caches range in size from 32 kb - 4 Mb and is much slower than the L1 cache with access time about a factor of 2-5 more than the L1 cache. Some computers provide Level 3 cache, but these are few. The next level is the actual RAM with the slowest access times but is the largest.

To make software as fast as possible, careful management of the caches must be maintained. If a data element is not in the cache then we call this a *miss*, if it is in the cache we call this a *hit*. Our desire is to minimize misses. A miss can cost different amounts depending on which cache level misses. If the data is not in the L1 cache, the L2 cache is checked, then the RAM. Because the L2 cache is much larger we can place much more data (i.e. the entire vector or matrix of interest) here initially, then place smaller data chunks inside the L1 cache as needed. The key is to do optimal replacements of the block inside the L1 cache. Simply meaning that when we fill the L1 cache, we only want to operate on those elements, then place the entire block back to the next level and retrieve a new block. This avoids many as many misses as possible.

2.4.2 A Faster Matrix Multiplication

We can now develop a method to improve the matrix multiply. We will do this in a sequential manner. The first step is to look at the loops in Code Example 2.23. Here we can simply rearrange the loop such that the most accessed element $c(i, j)$ is in the innermost loop as in Code Example 2.25. Here the indexes i, j, k have been flipped. The

Code Example 2.25 Simple tensor matrix multiplication with loop indexes rearranged.

```
template<class T>
matrix<T> operator*(matrix<T> &a, matrix<T> b)
{
    matrix<T> c(a.rows(), b.cols());
    c=0; //fill with zeros
    int i,j,k;
    for(k=0;k<b.cols();++k){
        for(j=0;j<b.cols();++j){
            for(i=0; i<a.rows();++i){
                c(i,j)+=a(i,k) * b(k,j);
            }
        }
    }
    return c;
}
```

GNU compiler will rearrange the loops automatically as shown, so we cannot show the improvement in MFLOPS for this particular optimization.

The loop unrolling technique discussed above is also performed by the GNU compiler and even better than by hand as it will unroll the higher level loops also. Here we demonstrate its effect for completeness sake. In Code Examples 2.26 we find a partially unrolled loop. I found that using five fold unrolling was a bit better than the four fold unrolling on the 933 MHz Pentium III. The comparison with the Code Example 2.25 is shown in Figure 2.11.

The next level of optimization would be to make sure the L2 cache is completely

Code Example 2.26 Partial loop unrolling for the matrix multiply.

```

matrix<T> mulmatLoopUnroll(matrix<T> &a, matrix<T> &b)
{
    int i,j,k, leftover;
    matrix<T> c(a.rows(), b.cols(), 0);
    static int Unrolls=5;
    //figure out how many do not fit in the Pipeline unrolling
    leftover=c.cols() % (Unrolls);
    for(k=0;k<c.rows();++k){
        for(j=0;j<c.cols();++j){
            i=0;
            //do the elements that do not fit
            //in the unrolling
            for(;i<leftover;++i)
                { c(i,j)+=a(i,k) * b(k,j); }

            //do the rest
            for(;i<c.cols();i+=Unrolls){
                //avoid calculating the indexes twice
                int i1=i+1, i2=i+2, i3=i+3, i4=i+4;
                //avoid reading the b(k,j) more then once
                T tmpBkj=b(k,j);
                //read the a(i,k)'s first into the registers
                T tmpAij=a(i,k);
                T tmpAi1j=a(i1,k);
                T tmpAi2j=a(i2,k);
                T tmpAi3j=a(i3,k);
                T tmpAi4j=a(i4,k);

                c(i,j)+=tmpAij * tmpBkj;
                c(i1,j)+=tmpAi1j * tmpBkj;
                c(i2,j)+=tmpAi2j * tmpBkj;
                c(i3,j)+=tmpAi3j * tmpBkj;
                c(i4,j)+=tmpAi4j * tmpBkj;
            }
        }
    }
    return c;
}

```

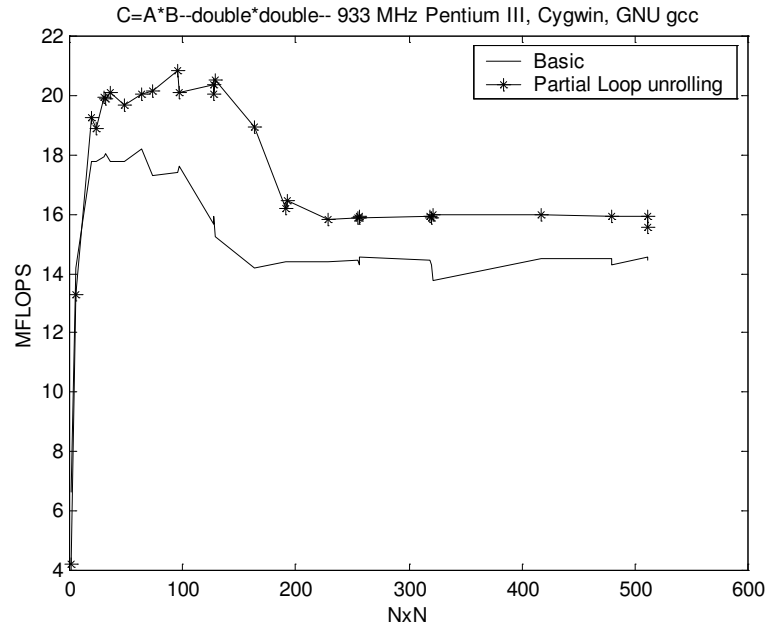


Figure 2.11: MFLOPS of a matrix multiplication: comparison of Code Example 2.25 (solid line) and Code Example 2.26 (*).

full. For large matrices we would have to divide the matrix into sub matrices that fit into the L2 cache. For a L2 cache of 1 Mb, we can fit approximately 125000 doubles. Of course we have 3 matrices to consider so that would drop us to ~ 42000 doubles per matrix. This assumes that we would have the total L2 cache, but we will need some space for the indexes and other functional elements as well as operating system elements and other programs running as well as the required instruction set. We will halve this number to 20000 doubles as the *L2 block* size. The largest square matrix that will fit into a 20000 data chunk is $\sim 140 \times 140$. If one looks back to Figure 2.6a and Figure 2.11 you can see the unoptimized multiply has a performance drop when the matrix size is over 160. This is a result of the L2 cache being unoptimally used on the 1Mb L2 cache of the Pentium III.

In order to realize our blocking technique, we must *copy* sub-matrices of the larger matrix into smaller matrices that fit into the L2 cache. Each sub-matrix is then added to

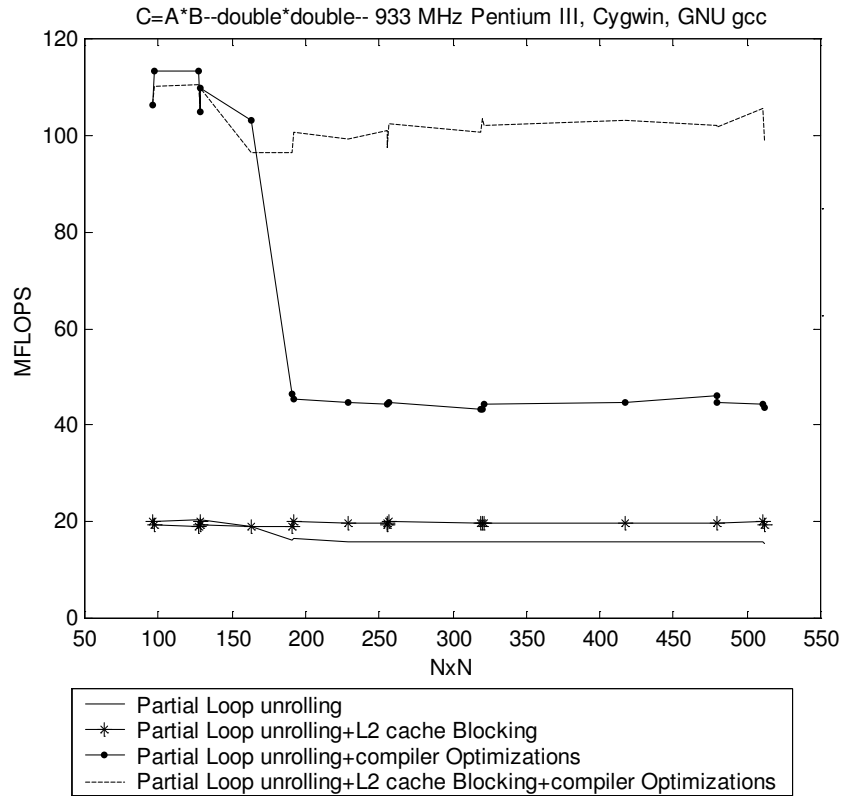


Figure 2.12: MFLOPS of a matrix multiplication: comparison of simple loop unrolling Code Example 2.26 (solid line) and the L2 cache blocking with unrolling (see Appendix A.1.3). Compiler optimizations increase the total MFLOPS (line with dots) and the benefit of L2 blocking (dashed line).

the output matrix. This is same as performing a normal matrix multiply as in Figure 2.5 if we consider each box a sub matrix rather than one element.

The code for performing the L2 cache blocking is shown in Appendix A.1.3 because it is a bit long. Figure 2.12 shows the improvement over not blocking the L2 cache. In general the L2 cache is still relatively slow when compared to the L1 cache and thus the speed enhancement is small (a few MFLOPs here). We can turn on the compiler optimizations to see a much more dramatic effect of L2 blocking and this is also shown in Figure 2.12.

The next level of optimization is L1 cache blocking. This turns out to be a very

hard problem to optimize for many reasons. Each hardware has a different memory structure for the L1 cache that needs to be matched exactly to improve performance. Because the sub matrices need to be much smaller to fit into the L1 cache size, there is a large penalty for copying the sub matrix if the copying is not optimized. Being in the L1 cache does not guarantee the highest performance, because the register memory must then be used effectively. Finally we would need to program in assembly to use the SIMD extensions. All of these factors can then have varying degrees of pipe line optimizations. Finding an optimal solution would take a single person much tinkering with all of them, or a computer search algorithm to find the best one. ATLAS[21] performs this search effectively and this is why its performance in Figure 2.6 is much higher than anything we have shown here.

Conclusions

In this chapter I have given you the tools and methods for constructing highly optimal computer algorithms. There are two essential levels. The first, expression templates, involves an abstract software interface to reduce the number of operations on a stack while still maintaining a high-level of simplicity for the user. The second, hardware optimizations, is an art unto itself and should be used where we cannot rely on the compiler to generate the optimized code. Because each hardware is different, optimizing for one architecture will most definitely not work on another. For this reason, there is not generic abstraction like the expression template technique. However, both code abstraction and hardware optimization are necessary for fully optimal solutions.

Chapter 3

NMR Forms

Before we can lay down the foundation of a complete NMR tool kit, we need to know what sort of mathematics we are dealing with. What kind of interactions must we take into account to achieve a real physical model? All of NMR, like most physical systems, can be reduced to two extremes, the Quantum and the Classical. Both are treated in fundamentally different ways both mathematically and numerically. Since classical mechanics is usually a bit more intuitive, we shall start there.

3.1 Classical Mechanics

The basic tenant of the classical description of NMR is a magnetic dipole interacting with some external magnetic field. As we will show later there are many ‘external magnetic fields’. The basic interaction is easily described by a first order differential equation

$$\frac{d\mathbf{M}}{dt} = -\gamma\mathbf{M} \times \mathbf{B}. \quad (3.1)$$

Most of the world calls this the Bloch Equation [23]. Here \mathbf{M} is the magnetic

moment (sometimes called μ) and is a 3 dimensional vector (a coordinate). Various orthogonal representations can be given to the three components, here, we will stick to simple Cartesian

$$\mathbf{M} = (M_x, M_y, M_z). \quad (3.2)$$

\mathbf{M} is usually considered a bulk property: the entire macroscopic sample of spins add together to produce \mathbf{M} . An individual spin's magnetic moment will be called μ , and a normalized bulk magnetic moment will be called little m , \mathbf{m} . \mathbf{B} is also a 3-vector and represents the external magnetic field. γ is a nuclear spin's gyromagnetic ratio which converts the 'Gauss' or 'Tesla' of $\mathbf{M} \times \mathbf{B}$ to a frequency. The gyromagnetic ratio is spin specific. The cross product is representative of a torque which the magnetic moment feels from the external field.

3.2 Bloch Equation Magnetic Fields

We are interested in any and all magnetic fields of the most general form $\mathbf{B}(\mathbf{r}, \mathbf{t})$, a magnetic field as a function of position and time.

Offsets

The first sets of fields of interest are those that we can apply to the system using electro-magnets, superconducting magnets, or simple coils. In most NMR circumstances we apply a very large static magnetic field along the z -axis (actually we define our z -axis about this large field). Typically superconducting fields are of the order of Telsa or higher, and our Bloch equation is

$$\frac{d\mathbf{M}}{dt} = -\gamma\mathbf{M} \times \mathbf{B}_z. \quad (3.3)$$

This simple equation gives us a three equations of motion

$$\frac{dM_x}{dt} = -\gamma B_z M_y \quad (3.4)$$

$$\frac{dM_y}{dt} = \gamma B_z M_x \quad (3.5)$$

$$\frac{dM_z}{dt} = 0 \quad (3.6)$$

If we specify an initial condition with $\mathbf{M}(\mathbf{0}) = (M_x^o, M_y^o, M_z^o)$, we get the analytic solution of

$$M_x(t) = M_x^o \cos(\gamma B_z t) - M_y^o \sin(\gamma B_z t) \quad (3.7)$$

$$M_y(t) = M_y^o \cos(\gamma B_z t) + M_x^o \sin(\gamma B_z t) \quad (3.8)$$

$$M_z(t) = M_z^o \quad (3.9)$$

The magnetization therefore spins around the z -axis. The γB_z term is very large. For a $B_z = 1$ Telsa we get a oscillation frequency of 42.58 MHz for a proton. To solve such a fast solution is akin to suicide when a typical NMR experiments can last on order of seconds. A solver would be required to evaluate millions of functions, making it very inefficient. Given that the field is static, we can go into the rotating frame of the field. That is, spin ourselves at the in the opposite direction, but at the same rate as the magnetization is spinning. The physics should not change in this new frame but we need to add this new term to the equations of motion

$$\frac{d\mathbf{M}}{dt} = \mathbf{M} \times \boldsymbol{\Omega}_r + \left[\frac{d\mathbf{M}}{dt} \right]_r. \quad (3.10)$$

Where Ω_r is the rotational frequency of the rotating frame and $\left[\frac{d\mathbf{M}}{dt} \right]_r$ is the term for how \mathbf{M} appears in the rotating frame. We wish to satisfy the condition,

$$\left[\frac{d\mathbf{M}}{dt} \right]_r = 0. \quad (3.11)$$

In other words, we want everything to be still in the rotating frame. Comparing Eq. 3.11 and Eq. 3.10 we see we need to rotate counter to the B_z field, which give us

$$\begin{aligned}\Omega_r &= -\gamma B_z \\ \left[\frac{d\mathbf{M}}{dt}\right]_r &= -\gamma\mathbf{M} \times \mathbf{B}_z - \gamma\mathbf{M} \times (-\mathbf{B}_z). \\ &= 0\end{aligned}\tag{3.12}$$

As you can see we have gone into a frame where nothing evolves, a rather boring frame. So suppose that our magnetization feels a slightly different field. Call the difference $\Delta\mathbf{B}$ then our applied field, we then must add a term that describes this new offset

$$\frac{d\mathbf{M}}{dt} = -\gamma\mathbf{M} \times (\mathbf{B}_z + \Delta\mathbf{B})\tag{3.13}$$

Looking at the 3-vector of equations of motion,

$$\begin{aligned}\frac{dM_x}{dt} &= -\gamma((B_z + \Delta B_z)M_y - \Delta B_y M_z) \\ \frac{dM_y}{dt} &= \gamma((B_z + \Delta B_z)M_x - \Delta B_x M_z) \\ \frac{dM_z}{dt} &= -\gamma(\Delta B_y M_x - \Delta B_x M_y)\end{aligned}\tag{3.14}$$

If we assume that $B_z \gg \Delta B_i$ by orders of magnitude, then the only terms that contribute to the observable evolution will be terms with B_z , thus eliminating any solo ΔB_i terms in our equations of motion. This approximation is sometimes called first order perturbation theory, where the only terms that remain in a small perturbation are the ones parallel to the axis of the main interaction. We will also call this *truncation* of an interaction because we essentially drop some terms from the interaction. If we then apply to the rotating frame to this reduced form we get

$$\begin{aligned}\left[\frac{dM_x}{dt}\right]_r &= -\gamma\Delta B_z M_y \\ \left[\frac{dM_y}{dt}\right]_r &= \gamma\Delta B_z M_x \\ \left[\frac{dM_z}{dt}\right]_r &= 0.\end{aligned}\tag{3.15}$$

As you can see we are back to the form of the equation in Eq. 3.4 except now, the ΔB_z terms are much smaller: anywhere from Hz to kHz. In the rotating frame a small offset from the main magnetic field will appear to oscillate about the main magnetic field axis.

If we *cannot* assume that $B_z \gg \Delta B_i$, then we must use the full form of the equations of motion in Eq. 3.14. This does have an analytic solution but it is a bit messy to write here. Furthermore if the main static applied field is not along the z-axis then we still can reduce the equations of motion to the form shown in Eq. 3.3 and we can use the technique in section 4.1.1 to solve the problem.

Magnetic Pulses

Magnetic pulses are how one can manipulate the magnetization. I describe them here as magnetic pulses, instead of the usual ‘Radio Frequency Pulses’ because the ‘Radio Frequency’ applies only when there is a large Telsa external field already applied on the system. In general a magnetic pulse is similar to an offset with the exception that it is applied along an arbitrary direction and for some length of time. Offsets are usually independent of time.

If the sample is not under the influence of a large external field, then any directed DC (Direct Current) pulse will behave as the external field and the spins will evolve under that field according to the same equations as section 3.2. If we are under the influence of a large external field, a relatively weak DC pulse (all one can muster experimentally) will do nothing unless applied along the same axis as the main field, as we showed in section 3.2. All we would observe then is a larger offset. So we need some other way to use an applied field to give us some control.

First lets assume that we can make our applied field time dependant. We will call

this field \mathbf{B}_1 by convention. Then our equation of motion in the non-rotating frame become

$$\frac{d\mathbf{M}}{dt} = -\gamma\mathbf{M} \times (\mathbf{B}_z + \mathbf{B}_1(\mathbf{t})). \quad (3.16)$$

We wish to go into the rotating frame to make things numerically simple, but now we have an added complication. We wish that the rotated $\mathbf{B}_1(\mathbf{t})$ (calling it $\mathbf{B}_1^r(\mathbf{t})$) appear in the rotating frame and not become truncated. We have already made the assumption that our B_z was much larger than anything else, so our rotating frame will still be $-\gamma B_z$ giving our rotating frame equation of motion as

$$\left[\frac{d\mathbf{M}}{dt} \right]_r = -\gamma\mathbf{M} \times (\mathbf{B}_1^r(\mathbf{t})). \quad (3.17)$$

Now we can express the non-rotating frame B_1 field as the rotating field multiplied by the reverse time dependant rotation around the z-axis.

$$\mathbf{B}_1(\mathbf{t}) = \mathbf{R}\mathbf{B}_1^r(\mathbf{t}) \quad (3.18)$$

where \mathbf{R} is a rotation matrix is given by the solution to $\frac{d\hat{r}}{dt} = \hat{r} \times \Omega_z$, the solution given in Eq. 3.7 with $M \rightarrow r$ and $B_z \rightarrow \Omega_z$.

$$R = \begin{pmatrix} \cos(\Omega_z t) & \sin(\Omega_z t) & 0 \\ -\sin(\Omega_z t) & \cos(\Omega_z t) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (3.19)$$

thus

$$\mathbf{B}_1(\mathbf{t}) = \begin{pmatrix} B_{1,x}^r(t) \cos(\Omega_z t) + B_{1,y}^r(t) \sin(\Omega_z t) \\ B_{1,y}^r(t) \cos(\Omega_z t) - B_{1,x}^r(t) \sin(\Omega_z t) \\ B_{1,z}^r(t) \end{pmatrix} \quad (3.20)$$

So we see that in order for the external applied pulse to remain in the rotating frame, it must be rotating at the same frequency as the rotating frame (i.e. a resonance condition).

We could have intuitively guessed this result, but the result for off axis rotating frames and other time dependant interactions, which give much more complicated expressions, can all be derived the same way as this example.

This most typical magnetic pulse NMR uses is one that is constant in the rotating frame, thus $\mathbf{B}_1^r(\mathbf{t}) = \mathbf{B}_1^r$. The non-rotating frame from this point on will be called the *lab frame*. In the lab frame, we still have a time dependence of $\Omega_z t$. In the rotating frame a perfectly resonant pulse is typically applied perpendicular to the z -axis, and can reside anywhere in the xy -plane. We represent this in the rotating frame as

$$\mathbf{B}_1 = (\mathbf{B}_{1x} \cos(\phi), \mathbf{B}_{1y} \sin(\phi), \mathbf{0}) \quad (3.21)$$

where ϕ is a phase factor. If we move off resonance, the \mathbf{B}_1 vector points out of the xy -plane thus introducing an extra z term

$$\mathbf{B}_1 = (\mathbf{B}_{1x} \cos(\phi), \mathbf{B}_{1y} \sin(\phi), \mathbf{B}_{1z}). \quad (3.22)$$

Shaped pulses introduce time dependent amplitudes ($B_{1i} = B_{1i}(t)$) or phase factors ($\phi = \phi(t)$) or both.

Gradients

Gradients can be thought of as a combination of magnetic pulses with a spatial dependence. A gradient magnetic field will be called B_g .

$$\mathbf{B}_g = \mathbf{B}_g(\mathbf{r}, \mathbf{t}) \quad (3.23)$$

Because they are spatially varying, the quantity of interest is its derivative with respect to the spatial coordinates which have 9 components. The result is the gradient

tensor

$$\mathbf{G} = \begin{pmatrix} \frac{\delta B_{gx}}{\delta x} & \frac{\delta B_{gy}}{\delta x} & \frac{\delta B_{gz}}{\delta x} \\ \frac{\delta B_{gx}}{\delta y} & \frac{\delta B_{gy}}{\delta y} & \frac{\delta B_{gz}}{\delta y} \\ \frac{\delta B_{gx}}{\delta z} & \frac{\delta B_{gy}}{\delta z} & \frac{\delta B_{gz}}{\delta z} \end{pmatrix} = \begin{pmatrix} G_{xx} & G_{xy} & G_{xz} \\ G_{yx} & G_{yy} & G_{yz} \\ G_{zx} & G_{zy} & G_{zz} \end{pmatrix}. \quad (3.24)$$

In a high magnetic field along the z -axis, the only components of the gradient tensor that contribute to any observed effect are the terms along the z direction

$$B_g^{hf} = (0, 0, B_{gz}). \quad (3.25)$$

If we apply a linear gradient (the most common NMR situation) then

$$\frac{\delta B_{gi}}{\delta j} = \text{const} = G_{ij} \quad (3.26)$$

and we can get the total field along the z -axis via a sum of all the z derivatives times its position

$$B_{gz}^{hf} = \mathbf{G}_{iz} \bullet \mathbf{r} = \mathbf{G}_{xz} * \mathbf{x} + \mathbf{G}_{yz} * \mathbf{y} + \mathbf{G}_{zz} * \mathbf{z} \quad (3.27)$$

As you can see this simply acts like a spatial dependent offset. If we are not in a high field the entire tensor must be used; furthermore, the simple formula in Eq. 3.27 is not valid for non-linear gradients.

Relaxation

Relaxation itself is not a ‘magnetic field’ but more a heuristic addition to the equations of motion. There are two fundamentally different forms of relaxation. The first occurs from energy transfer from the system we are interested in to a system outside of our control (usually called the *lattice*). This form is usually called longitudinal relaxation or T_1 relaxation. This phenomenon occurs in almost every physical system where we must separate control to the system of interest from the outside world. This basic relaxation is

the driving force that drives the dynamics of the system back to its equilibrium condition at some rate, $1/T_1$. If our equilibrium condition is $\mathbf{M}^o = (\mathbf{M}_x^o, \mathbf{M}_y^o, \mathbf{M}_z^o)$, then at any given time, relaxation will move the magnetization back towards this vector. Thus we have a new term in the equation of motion defined as

$$\frac{d\mathbf{M}}{dt} = \frac{1}{T_1}(\mathbf{M}^o - \mathbf{M}(\mathbf{t})). \quad (3.28)$$

In NMR there are many ways to calculate T_1 based on the system of study, the reader is referred to Ref. [24] for more information and the various equations. For many computational studies, all we need to know is this value. The most common case in NMR is the high field case, where $\mathbf{M}^o = (\mathbf{0}, \mathbf{0}, \mathbf{M}_z^o)$, so T_1 relaxation is only applicable to the z part of our equations.

The second form of relaxation is usually an *internal* phase disturbance relaxation. In a many body system there are slight differences in the local environments of each individual spin cause slightly different evolutions between them. In a bulk sample (where we have an Avogadro's number of spins) this effect manifests itself as a dephasing of any previously inphase magnetization. This type of relaxation is typically called transverse or T_2 relaxation. Unlike T_1 relaxation, this interaction can be reversed because it is still within our system (under our control). The reversibility or irreversibility of the relaxation mechanism is what defines T_2 from T_1 . It is called transverse relaxation because it acts in a plane perpendicular to the equilibrium condition. Because it acts on the plane perpendicular to the equilibrium condition, we have to rotate from this axis to the perpendicular plane. If we remain in our Cartesian basis, then we can get two angles from the z-axis to unit

equilibrium vector ($\hat{M}^o = \mathbf{M}^o / \|\mathbf{M}^o\|$), θ , and the rotation about the xy-plane, ϕ as

$$\begin{aligned}\theta &= \arccos\left(\hat{M}_z^o\right) \\ \phi &= \arctan\left(\frac{M_x^o}{M_y^o}\right)\end{aligned}\tag{3.29}$$

T_2 does not ‘return’ magnetization to equilibrium, it simply removes magnetization, so our equation of motion should look something like

$$\frac{d\mathbf{M}}{dt} = \frac{-1}{T_2} \mathbf{C} \cdot \mathbf{M}(t)\tag{3.30}$$

where \mathbf{C} is the rotation matrix to take us into the plane perpendicular to the equilibrium axis.

$$\mathbf{C} = \begin{pmatrix} \cos(\phi) & \sin(\phi)\cos(\theta) & 0 \\ \cos(\phi)\cos(\theta) & \sin(\phi) & 0 \\ 0 & \sin(\theta) & 0 \end{pmatrix}.\tag{3.31}$$

To get \mathbf{C} we can first assume that our plane of interest is the xy-plane, then rotate the x and y axis to the \hat{M}^o axis. For those of you that are paying attention, to accurately describe a three dimensional rotation, we need three angles, not just two. However, the third angle here would describe the relative rotation of a *vector* in that plane perpendicular to \mathbf{M}^o . Luckily for us, this third angle is irrelevant as it would place the T_2 relaxation along a specific axis, where it actually is directionally independent in that plane. I say ‘luckily’, because there is no way for us to get this third angle.

The standard high field NMR case of $\mathbf{M}^o = (\mathbf{0}, \mathbf{0}, M_z^o)$, leads us to the normal form of the T_2 relaxation equations

$$\frac{d\mathbf{M}}{dt} = \frac{-1}{T_2} (M_x(t), M_y(t), 0).\tag{3.32}$$

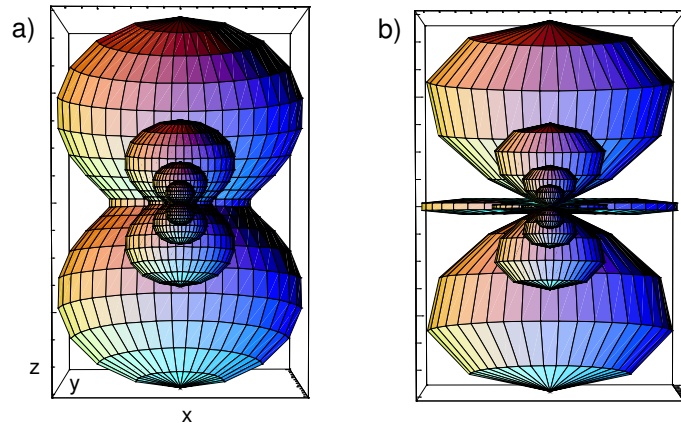


Figure 3.1: The magnitude of the dipole field in no static external field (a), and in a high magnetic field along \hat{z} (b). Each shell represents $B^D \bullet B^D$ at $r = 0.7..1$ in 0.1 steps and where $\mu = (\mathbf{0}, \mathbf{0}, \mathbf{1}/\mu_o)$.

Dipole Interaction

The dipole interactions is one of the most important to the field of NMR, for in this one interaction we have a method for determination of distances between atoms, for simplifying (and complicating) spectra, and basically adding a second dimension on to our normal 1-D offset interaction. It is also one of the chief mechanisms for T_2 type relaxation. The interaction is a spin-spin interaction: the dipolar field on one spin is felt as a magnetic field by its neighbor. In its most general form the dipolar field, B^D , at position \mathbf{r} from a single spin is given by

$$\mathbf{B}^D(\mathbf{r}) = \frac{\mu_o}{4\pi} \frac{\mathbf{3}(\mu \bullet \hat{\mathbf{r}}) \hat{\mathbf{r}} - \mu}{(\mathbf{r} \bullet \mathbf{r})^{3/2}} \quad (3.33)$$

where μ_o is the permeability of free space ($12.566370614 * 10^{-7} T^2 m^3 / J$). The dipolar field from a single magnetic moment μ at \mathbf{r} is proportional to the cube of the distance away from the spin. If we are not in a high field, this give us the ‘dumbbell’ picture of the magnetic field as shown in Figure 3.1a. To get the images in Figure 3.1, we have to first transform

the spherical basis, and choose a direction for μ . Here we choose $\mu = (\mathbf{0}, \mathbf{0}, \mu_z)$, to get the following equations

$$\begin{aligned} B_x^D(r) &= \frac{\mu_o \mu_z}{4\pi} \left[\frac{3\hat{x}\hat{z}}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_z}{4\pi} \left[\frac{3 \cos \phi \sin \theta \cos \theta}{|r-r_i|^3} \right] \\ B_y^D(r) &= \frac{\mu_o \mu_z}{4\pi} \left[\frac{3\hat{y}\hat{z}}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_z}{4\pi} \left[\frac{3 \sin \phi \sin \theta \cos \theta}{|r-r_i|^3} \right] \\ B_z^D(r) &= \frac{\mu_o \mu_z}{4\pi} \left[\frac{3\hat{z}^2-1}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_z}{4\pi} \left[\frac{3 \cos^2 \theta - 1}{2|r-r_i|^3} \right]. \end{aligned} \quad (3.34)$$

In the high field case, we must remember that the only terms that survive from the above equations are those that either contribute to the z-axis or those that are invariant to any rotation (terms like $\mu \bullet \mu$ and $\mu \bullet \hat{r}$). Then we get the following sets of equations shown in Eq. 3.35 which has a field shown in Figure 3.1b

$$\begin{aligned} B_x^D(r) &= \frac{\mu_o \mu_x}{4\pi} \left[\frac{-1}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_x}{4\pi} \left[\frac{-1}{|r-r_i|^3} \right] \\ B_y^D(r) &= \frac{\mu_o \mu_y}{4\pi} \left[\frac{-1}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_y}{4\pi} \left[\frac{-1}{|r-r_i|^3} \right] \\ B_z^D(r) &= \frac{\mu_o \mu_z}{4\pi} \left[\frac{3\hat{z}^2-1}{2|r-r_i|^3} \right] = \frac{\mu_o \mu_z}{4\pi} \left[\frac{3 \cos^2 \theta - 1}{2|r-r_i|^3} \right]. \end{aligned} \quad (3.35)$$

In more interesting simulation we are interested in many spins, and every spin has a magnetic dipole moment, thus every spin, spin j , sees a field generated by all of its neighbors, B_i^D , as

$$\mathbf{B}_j^D = \sum_{i \neq j}^N \mathbf{B}_i^D(\mathbf{r}_i - \mathbf{r}_j) \quad (3.36)$$

where N is the total number of spins and $\mathbf{r}_i - \mathbf{r}_j$ is the vector separating the two spins. This sum is one of the computationally limiting steps as it requires the sum to be calculated for every spin j at every integration step. All the previous interaction have been had N fold scaling, where this one has N^2 scaling. There are other complications as well. In a bulk sample, we are not concerned with a single spin magnetic moment, but a small volume of spins, which has total magnetization M . Thus we need to calculate the dipole fields due to the small volume. If M is not chosen properly the value of B^D will grow out of

control. These considerations dealing with what M really is will be treated at the end of this chapter. Also in macroscopic samples, the sum in Eq. 3.36 becomes an integral, this integral is the topic of another effect, the local field and will be discussed below.

Bulk Susceptibility

The next three and final fields I will discuss are all high field, bulk effects simply meaning they are inherently due to the high magnetic field and the fact that we are dealing with a macroscopic magnetized sample. The first and easiest to understand is the bulk susceptibility. All matter when exposed to a magnetic or electric field ‘reacts’ to the field’s presence by either opposing the field or aligning with the field. A sample of nuclear spins is a slightly polarized by an applied field so it creates a magnetic moment in the sample. Inside the sample the total field, \mathbf{B} , is simply a sum of the two fields

$$\mathbf{B} = \mu_0(\mathbf{H} + D\mathbf{M}) \quad (3.37)$$

where $\mathbf{H} = \frac{\mathbf{B}_{\text{applied}}}{\mu_0}$ is the applied field intensity, \mathbf{M} is the sample magnetization. The constant in front of \mathbf{M} , D depends on the sample shape. For a perfect sphere this constant is 0, for a flat disk it is 1/3, and for a long cylinder (i.e. a liquid NMR sample tube) it is 1 (its maximum value). A pictorial representation of this effect is shown in Figure 3.2. We can further simplify this equation by realizing that the applied field \mathbf{H} is responsible for the magnetization \mathbf{M}

$$\mathbf{M} = \mu_0\chi\mathbf{H} \quad (3.38)$$

where χ , called the magnetic susceptibility, is related to the sample. We have made the assumption that the material is isotropic and linear in \mathbf{H} . For paramagnetic material this constant is large, for diamagnetic materials (like water, and most NMR samples), χ is quite

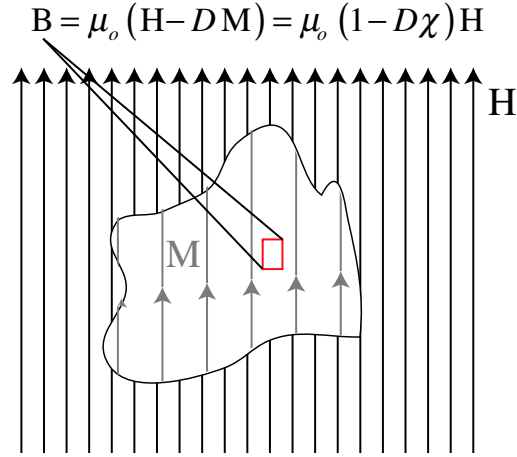


Figure 3.2: The magnetization of a sample inside a magnetic field.

small (of order 10^{-6}) and negative. We will discuss this constant more later. Thus our field equation becomes, for diamagnetic samples,

$$\mathbf{B}^{\text{bs}} = \mu_0(1 - D\chi)\mathbf{H}. \quad (3.39)$$

In a high \mathbf{H} intensity (by high we mean $|H| \gg |M|$), our rotating frame transformation indicates that only the components of $\chi\mathbf{H}$ that are parallel with \mathbf{H} will contribute to any observed effect. Again, this effect behaves simply like an offset. However, we can control \mathbf{M} using the magnetic pulses, as a consequence, in a high field along \hat{z} we can turn on and off this effect (i.e. turn off and on the offset). If you place our magnetization on the xy -plane, then there is no contributing effect, on the other hand, if we align our magnetization along the field, we would see a slight offset. The equations of motion are only affected by the magnetic field along z and we get these terms in our equations of motion:

$$\begin{aligned} \frac{dM_x}{dt} &= \chi\gamma M_z^T(t)M_y \\ \frac{dM_y}{dt} &= -\chi\gamma M_z^T(t)M_x \\ \frac{dM_z}{dt} &= 0 \end{aligned} \quad (3.40)$$

$M_z^T(t)$ is the *TOTAL* magnetization along the z axis at any given time.

Radiation Damping

Another high field effect comes from the actual hardware used in an NMR experiment. Typically, a solenoid is placed around the sample that acts both as the ‘magnetic pulse’ and the detection apparatus. It detects signal using Faraday’s Law, which states any moving magnetic field creates a reaction electric current inside a conductor. The changing current/voltage (or electro-motive-force (*emf*)) is related to the changing magnetic flux, Φ as

$$\frac{d\Phi}{dt} = emf \quad (3.41)$$

In a simple solenoid Φ is simply the magnetic field B times a constant area, thus we can relate this *emf* to our Bloch equations

$$\frac{dM}{dt} * \text{Area} = emf. \quad (3.42)$$

This new *emf*, which is time dependant, then creates another magnetic field (Lenz’s Law), which apposes our magnetization. This is in essence applying another magnetic pulse as its time dependence is the same as the magnetization’s time dependence (i.e. still on resonance).

$$\frac{d(emf)}{dt} = -\frac{d(B^{rd})}{dt} = -\alpha \frac{dM}{dt}. \quad (3.43)$$

The constant, α , simply represents the strength of the back reaction field. We now have a non-linear equation as this effect depends directly on how much magnetization is present (but in the opposite rotating sense). The amount of back reaction depends on a number of physical parameters which can be reduced to two constants called the Quality factor or Q and the sample filling factor, η . Q is dependant on the coil size, inductance,

and a slew of other details. η is simply how much of the sample fills the space in the coil. Obviously the effect is also driven by how much total magnetization exists in the sample. This magnetization is only appreciable enough to create any effect in high magnetic fields, so our rotating frame approximation still holds. Another key bit of information is that the coils are aligned perpendicular to the main field, so the reaction field is then only applied in the xy -plane. We can finally write down the radiation damping reaction field as

$$\begin{aligned} B_x^{rd}(t) &= -1/\tau_r M_y^T(t)/\gamma \\ B_y^{rd}(t) &= 1/\tau_r M_x^T(t)/\gamma \end{aligned} \tag{3.44}$$

where the $1/\tau_r$ is related to the coils parameters ($1/\tau_r = Q\eta|M^o|\gamma$), and $M_i^T(t)$, is the *total* magnetization at a given time t . We have made the assumption that the back reaction field is the same at any point in the sample. If this is not the case (which it is not in a real experiment due to coil edge effects) then Eq. 3.44 becomes an integral where $B_i^{rd}(t) \rightarrow B_i^{rd}(\mathbf{r}, t)$, $\tau_r \rightarrow \tau_r(\mathbf{r})$, and $M_i^T(t) \rightarrow M_i^T(\mathbf{r}, t)$. Assuming a uniform interaction, the new terms in our equations of motion are

$$\begin{aligned} \frac{dM_x}{dt} &= -\frac{1}{\tau_r} M_x^T(t) M_z \\ \frac{dM_y}{dt} &= -\frac{1}{\tau_r} M_y^T(t) M_z \\ \frac{dM_z}{dt} &= -\frac{1}{\tau_r} (M_x^T(t) M_x + M_y^T(t) M_y). \end{aligned} \tag{3.45}$$

Local Field

The final field I will discuss involves extending the bulk susceptibility and the dipole fields to a more closed form. In macroscopic samples, the dipole field at a position \mathbf{r} would be the sum over all the various dipoles in the sample. The trouble with performing this sum is that there is an Avogadros number of them, making this sum impossible to

perform, but with such a large number, we can reduce the sum to an integral

$$\mathbf{B}^{LF}(\mathbf{r}) = \frac{\mu_o}{4\pi} \int \frac{1 - 3 \cos(\theta_{r-r'})}{2 |\mathbf{r} - \mathbf{r}'|^3} [3M_z(r') \hat{z} - \mathbf{M}(\mathbf{r}')] dr'^3. \quad (3.46)$$

where $\theta_{r-r'}$ is the angle between the $r - r'$ vector and the magnetic field. Eq. 3.46 assumes that we are in a high field. \mathbf{B}^{LF} still exists in low field situations, however, the bulk property of the magnetization is so small, that it all but eliminates this effect. This integral can only be integrated analytically for a few special cases. If we assume uniform magnetization ($M(r) = M$), then the integral reduces to

$$\mathbf{B}^{LF}(\mathbf{r}) = \frac{\mu_o}{4\pi} [3M_z \hat{z} - \mathbf{M}] \int \frac{1 - 3 \cos(\theta_{r-r'})}{2 |\mathbf{r} - \mathbf{r}'|^3} dr'^3. \quad (3.47)$$

The remaining term in the integral will give a simple constant which is dependant on the shape of the sample, and we are left with something that looks very similar to the bulk susceptibility. If the sample shape is an ellipsoid (a sphere, disk, cylinder) then the integral in Eq. 3.47 is soluble[25, 26]. The dipolar local field looks like

$$\mathbf{B}^{LF}(\mathbf{r}) = \frac{\mu_o}{6} (3n_z - 1) [3M_z \hat{z} - \mathbf{M}] \quad (3.48)$$

where n_z is called the demagnetizing factor ($n_z = 0$ for a long rod, $n_z = 1/3$ for a sphere, $n_z = 1$ for a thin disk).

If the magnetization is not uniform, then in general we would have to evaluate this integral. In this case the best we can do numerically is break the integral into a sum over little Δr^3 volume elements. The problem with this technique is that it require many volume cells for the integral to converge properly, which can result in very long function evaluations. There do exist techniques using Fourier transforms to simply this integral[27, 28]. Those techniques however are for the most general case and the algorithmic complexity becomes daunting. There is another special case that is of greater interest

because it allow manipulation of this field[29, 27]. Upon an applied external gradient which completely ‘crushes’ (the magnetization in the total volume sums to 0) the magnetization along a single direction, we can write the field as

$$\mathbf{B}^{MLF}(\mathbf{r}) = \frac{3(\hat{s} \cdot \hat{z})^2 - 1}{2\tau_D} \left\{ [M_z(\hat{s}) - \langle M_z \rangle] - \frac{1}{3} [\mathbf{M}(\hat{s}) - \langle \mathbf{M} \rangle] \right\}. \quad (3.49)$$

We will call this the *Modulated* local field. In Eq. 3.49, \hat{s} is the direction of the modulation, and $\mathbf{M}(\hat{s})$ is the magnetization along the direction of the modulation. The $\langle \dots \rangle$ indicate the mean of the magnetization. Finally the time constant τ_d is $1/(\mu_o \gamma M^o)$ where M^o is the total equilibrium magnetization. This form of the \mathbf{B}^{MLF} does not require any explicit sums, so this interaction scales a N as well.

3.3 Quantum Mechanics

The fundamental fields discussed in section 3.1 also form the basis of the quantum mechanical description, with two fundamental differences: instead of fields, we are interested in the *Hamiltonian*, \mathbf{H} , of the system which evolve a *density operator*, ρ , rather than a magnetization.

$$\frac{d\rho}{dt} = -i\hbar[\mathbf{H}, \rho] \quad (3.50)$$

where [...] is a commutator operator ($[A, B] = AB - BA$). This equation is called the Liouville-Von Neumann equation. NMR specifically treats atomic spin states. There are a variety of properties associated with the spin states based on the spins quantum number I . A spin of quantum number I has $2I - 1$ possible states. The most common NMR spin of $I = 1/2$ has two states. NMR is a measure of bulk phenomena so simple states are not an accurate description. Like the classical case we must pick a basis in which to describe our

spin(s), here we choose the spin operator basis in a Cartesian frame

$$\begin{aligned}
 I_x &= \frac{1}{2} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\
 I_y &= \frac{1}{2} \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix} \\
 I_z &= \frac{1}{2} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \\
 I_e &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.
 \end{aligned} \tag{3.51}$$

The density operator has four possible states or linear combinations of any of the above 4 operators.

$$\rho = aI_e + bI_y + cI_x + dI_z + \text{higher order terms} \tag{3.52}$$

Usually, we work in a reduced basis, where we factor out an I_e term which describe a non-polarized, and non-manipulable set of states (the identity operator, I_e never effects any evolution nor is affected by any interactions). From this point on we will mention ρ as the reduced density operator. The higher order terms (terms proportional to I_i to some power $n > 1$) are typically ignored in NMR and will be discussed further in Section 3.4.

3.3.1 Rotations

Almost all of the mathematics of NMR can be reduced to rotations on quantum operators. There are two equally important views in treating quantum mechanical rotations: Cartesian based rotations and spherical tensor rotations. Computationally, Cartesian rotations should be a slight bit faster in execution in the general case, as rotation matrices

are all 3x3. Spherical tensor rotations are typically used for theoretical/symmetry based rotational considerations because of their nice symmetry properties. Both, however, may be used to treat NMR computationally or theoretically.

Cartesian Based Rotations

All 3 dimensional rotations can be reduced to three angles $\Omega = (\phi, \theta, \gamma)$. The angle ϕ rotates the xy-plane around the z-axis into new axes x' and y' , then θ around the old y -axis to rotate z-axis creating three new rotated basis x'' , y'' , and z' . Finally, γ rotates x'' and y'' about the z' -axis into the final new rotated state (x''' , y''' , and z'). Mathematically, this can be represented by a 3×3 matrix

$$\mathbf{R}(\Omega) = \begin{pmatrix} \cos \gamma \cos \phi - \cos \theta \sin \gamma \sin \phi & \cos \gamma \sin \phi - \cos \theta \sin \gamma \cos \phi & \sin \gamma \sin \theta \\ -\sin \gamma \cos \phi - \cos \theta \cos \gamma \sin \phi & -\sin \gamma \sin \phi - \cos \theta \cos \gamma \cos \phi & \cos \gamma \sin \theta \\ \sin \theta \sin \phi & -\cos \phi \sin \theta & \cos \theta \end{pmatrix}. \quad (3.53)$$

This can easily be generated by taking into account the separate three rotations

$$\mathbf{R}(\Omega) = R_z(\gamma)R_y(\theta)R_z(\phi). \quad (3.54)$$

Each R_i has the form shown in Eq. 3.7.

Spherical Tensor Rotations

The spherical tensor rotation representation actually comes about by treating symmetry and invariants of angular momentum in quantum mechanics. The most common form for rotation of angular momentum are the Wigner matrix elements [30]. Given the total angular momentum L , there are $(2L - 1)^2$ elements to rotate each of the $2L - 1$ eigenvectors

Table 3.1: Wigner rank 1 rotation elements, $D_{m,m'}^1$.

$\begin{matrix} \cdot\cdot\cdot & m' \\ m & \cdot\cdot\cdot \end{matrix}$	-1	0	1
-1	$e^{-i(\gamma+\phi)} \cos\left(\frac{\theta}{2}\right)^2$	$-e^{-i\phi} \frac{\sin(\theta)}{\sqrt{2}}$	$e^{-i(\phi-\gamma)} \sin\left(\frac{\theta}{2}\right)^2$
0	$e^{-i\gamma} \frac{\sin(\theta)}{\sqrt{2}}$	$\cos(\theta)$	$-e^{-i\gamma} \frac{\sin(\theta)}{\sqrt{2}}$
1	$e^{-i(\gamma-\phi)} \sin\left(\frac{\theta}{2}\right)^2$	$e^{i\phi} \frac{\sin(\theta)}{\sqrt{2}}$	$e^{i(\gamma+\phi)} \cos\left(\frac{\theta}{2}\right)^2$

(m). For $L = 1$, we have 9 matrix elements as shown in Table 3.1 using the same three angles as in the Cartesian case.

These matrix elements are usually called $D_{m,m'}^l$ where l is the rank of the matrix, and the m, m' correspond to a particular matrix element. There is a reduced notation given as

$$e^{i(m\gamma+m'\phi)} d_{m,m'}^l(\theta) \quad (3.55)$$

where $d_{m,m'}^l(\theta)$ is called the ‘reduce Wigner element’ because the two z-rotation angles ϕ and γ are easily factored out of the total matrix element. The reduced Wigner elements for $l = 2$ are shown in Table 3.2. Almost all NMR interactions are some form of the rank 0, 1, and 2 matrix elements. Like the Cartesian these rotations matrices can be generated from the individual rotations

$$\mathbf{R}(\Omega) = R_z(\gamma)R_y(\theta)R_z(\phi) = e^{iI_z\gamma}e^{iI_y\theta}e^{iI_z\phi}. \quad (3.56)$$

We can decompose our Hamiltonian into a spherical tensor basis. Hamiltonians are scalar/energy operators and invariant under a total system rotation, we end up with a

Table 3.2: Reduced Wigner rank 2 rotation elements, $d_{m,m'}^2$.

$\begin{matrix} \ddots & m' \\ & \ddots \\ m & \ddots \end{matrix}$	-2	-1	0
-2	$\cos(\frac{\theta}{2})^4$	$2 \cos(\frac{\theta}{2})^3 \sin(\frac{\theta}{2})$	$2\sqrt{\frac{3}{2}} \sin(\theta)^2$
-1	$2 \cos(\frac{\theta}{2})^3 \sin(\frac{\theta}{2})$	$\cos(\frac{\theta}{2})^2 (-1 + 2 \cos(\theta))$	$-\sqrt{\frac{3}{2}} \cos(\theta) \sin(\theta)$
0	$\sqrt{\frac{3}{2}} \sin(\theta)^2$	$\sqrt{\frac{3}{2}} \cos(\theta) \sin(\theta)$	$\frac{1+3 \cos(2\theta)}{4}$
1	$2 \cos(\frac{\theta}{2}) \sin(\frac{\theta}{2})^3$	$(1 + 2 \cos(\theta)) \sin(\frac{\theta}{2})^2$	$\sqrt{\frac{3}{2}} \cos(\theta) \sin(\theta)$
2	$\sin(\frac{\theta}{2})^4$	$2 \cos(\frac{\theta}{2}) \sin(\frac{\theta}{2})^3$	$\frac{\sqrt{\frac{3}{2}} \sin(\theta)^2}{2}$
$\begin{matrix} \ddots & m' \\ & \ddots \\ m & \ddots \end{matrix}$	1	2	
-2	$-2 \cos(\frac{\theta}{2}) \sin(\frac{\theta}{2})^3$	$\sin(\frac{\theta}{2})^4$	
-1	$(1 + 2 \cos(\theta)) \sin(\frac{\theta}{2})^2$	$-2 \cos(\frac{\theta}{2}) \sin(\frac{\theta}{2})^3$	
0	$-\left(\sqrt{\frac{3}{2}} \cos(\theta) \sin(\theta)\right)$	$\frac{\sqrt{\frac{3}{2}} \sin(\theta)^2}{2}$	
1	$\cos(\frac{\theta}{2})^2 (-1 + 2 \cos(\theta))$	$-2 \cos(\frac{\theta}{2})^3 \sin(\frac{\theta}{2})$	
2	$2 \cos(\frac{\theta}{2})^3 \sin(\frac{\theta}{2})$	$\cos(\frac{\theta}{2})^4$	

Hamiltonian of the form

$$\mathbf{H} = \sum_l \alpha_l \mathfrak{S}_l. \quad (3.57)$$

where each \mathfrak{S}_l is a spherical tensor basis element and each α_l is a complex constant. It is implied that \mathfrak{S}_l contains all the m subcomponents. An important aspect in NMR is that the Hamiltonians can be separated into a spatial tensor component, A_l and a spin tensor component T_l . So we can rewrite \mathfrak{S} as a tensor product of the two

$$\mathfrak{S}_l = A_l \cdot T_l. \quad (3.58)$$

Using the explicit form of the product we get

$$\mathfrak{S}_l = \sum_{m=-l}^l (-1)^m A_{l,m} T_{l,-m} = \sum_{m=-l}^l (-1)^m A_{l,-m} T_{l,m} \quad (3.59)$$

Each tensor component can be rotated by using our Wigner rotation matrix elements

$$A_{l,m'} = \sum_{m=-l}^l D_{m',m}^l(\Omega) A_{l,m} \quad (3.60)$$

3.3.2 Rotational Frames

PAS

The Hamiltonians are typically created with an initial reference frame centered on the atom. We can think of the atomic frame as being the diagonal representation of the interaction. As soon as we move from this frame via some rotation then elements become mixed combinations of the atomic frame. This atomic frame is given the name Principle Axis System (**PAS**). In the **PAS** frame the arbitrary interaction in NMR can be reduced to 3 components. In the Cartesian frame these are typically given the labels δ_x, δ_y , and δ_z in the spherical frame they are given the labels δ_{iso} (isotropic), δ_{ani} (anisotropic) and η (asymmetry), and are related via

$$\begin{aligned} \delta_{iso} &= 1/3(\delta_x + \delta_y + \delta_z) \\ \delta_{ani} &= \delta_z \\ \eta &= \frac{\delta_x + \delta_y}{\delta_z} \end{aligned} \quad (3.61)$$

The Cartesian interaction frame is a 3×3 matrix, and in the **PAS** it is given as

$$A_{cart}^{PAS} = \begin{pmatrix} \delta_x & 0 & 0 \\ 0 & \delta_y & 0 \\ 0 & 0 & \delta_z \end{pmatrix}. \quad (3.62)$$

The spherical basis reduced to a sum over the various rank l components as

$$\begin{aligned}
 A_{sph}^{PAS} &= A_0 + A_1 + A_2 \\
 A_{0,0} &= -\sqrt{3}\delta_{iso} \\
 A_{1,\pm 1} &= A_{1,0} = 0 \\
 A_{2,\pm 1} &= 0, \quad A_{2,\pm 2} = \frac{1}{2}\delta_{ani}\eta, \quad A_{2,0} = \sqrt{\frac{3}{2}}\delta_{ani}.
 \end{aligned}
 \tag{3.63}$$

Molecule Frame

The next frame is the molecular frame, where we have gone past the atomic frame and now look at the various atomic frames relationship to each other on a molecule where we assume the atoms are fixed in space. To create this transformation, one needs to define another axis system in the molecular frame, then rotate each of the atomic interactions to this new frame, either by a Cartesian Euler rotation (Eq. 3.53) or a spherical Wigner rotations (Eq. 3.60). The Euler angles used to perform this rotation will be called Ω_{mol} .

$$\begin{aligned}
 A_{cart}^{mol} &= R(\Omega_{mol}) \cdot A_{cart}^{PAS} \cdot R(\Omega_{mol})^{-1} \\
 A_{sph}^{mol} &= A_0^{PAS} + \sum_{m'=-2}^2 D_{m,m'}^2(\Omega_{mol}) A_{2,m'}^{PAS}
 \end{aligned}
 \tag{3.64}$$

Rotor Frame

This particular rotation takes the molecule frame into the frame of the physical sample. Again we need to pick a reference axis by which all molecules are to be rotated. If the sample is a liquid, then this particular rotation would be time dependant as all the molecules are rotating in various ways in time inside the liquid. In a solid powder sample, then there are many different orientations relative to the chosen reference axis and they are ‘fixed’ in time. In a liquid this rotation is unnecessary as usually the time dependence of this rotation (on the order of micro seconds) is much faster then the observable on the NMR

times scale (on the order of seconds/millisconds). So the effect of this rotation in a liquid averages away. This assumption is not true for large molecules like proteins or bicelles that have a very slow rotational rate, then the rotational average is only partial and must be included to achieve a proper model.

In solids, however, NMR experiments are performed in a ‘rotor’ (the sample holder) which is aligned in some arbitrary direction. So we call the Euler angles to rotate into this frame Ω_{rot} and this rotation is given by

$$\begin{aligned} A_{cart}^{rot} &= R(\Omega_{rot}) \cdot R(\Omega_{mol}) \cdot A_{cart}^{PAS} \cdot R(\Omega_{mol})^{-1} \cdot R(\Omega_{rot})^{-1} \\ A_{sph}^{rot} &= A_0^{PAS} + \sum_{m'=-2}^2 D_{m,m'}^2(\Omega_{rot}) \sum_{m''=-2}^2 D_{m',m''}^2(\Omega_{mol}) A_{2,m''}^{PAS} \end{aligned} \quad (3.65)$$

Lab Frame

The final rotational frame relates the rotor frame back a chosen lab frame. The lab frame is the final resting point for all interactions and is static (like the superconducting magnet is static). This frame needs to be included only when the rotor frame moves, otherwise, we could simply choose that static frame as the rotor frame and there is then no need to perform this rotation. However, many solid-state techniques use the fact that a rotating rotor provides another method of control over the interactions. We will call the Euler angles that rotate the rotor into the lab frame Ω_{lab} . The final set of rotations is then given by

$$\begin{aligned} A_{cart}^{lab} &= R(\Omega_{lab}) \cdot R(\Omega_{rot}) \cdot R(\Omega_{mol}) \cdot A_{cart}^{PAS} \cdot R(\Omega_{mol})^{-1} \cdot R(\Omega_{rot})^{-1} \cdot R(\Omega_{lab})^{-1} \\ A_{sph}^{lab} &= A_0^{PAS} + \sum_{m'=-2}^2 D_{m,m'}^2(\Omega_{lab}) \sum_{m''=-2}^2 D_{m',m''}^2(\Omega_{rot}) \sum_{m'''=-2}^2 D_{m'',m'''}^2(\Omega_{mol}) A_{2,m'''}^{PAS} \end{aligned} \quad (3.66)$$

Table 3.3: Spherical tensor basis as related to the Cartesian basis for spin i and spin j

Spherical Tensor $T_{l,m}^{spin}$	Cartesian Representation
$T_{0,0}^i$	$\mathbf{I} \cdot \mathbf{I}$
$T_{1,0}^i$	I_z^i
$T_{1,\pm 1}^i$	$\frac{1}{\sqrt{2}} I_{\pm}^i = \frac{1}{2\sqrt{2}} (I_x^i \pm i I_y^i)$
$T_{2,0}^{(i,j)}$	$\frac{1}{\sqrt{6}} [3I_z^i I_z^j - \mathbf{I}^i \cdot \mathbf{I}^j]$
$T_{2,\pm 1}^{(i,j)}$	$\frac{\pm 1}{2} [I_{\pm}^i I_z^j + I_z^i I_{\pm}^j]$
$T_{2,\pm 2}^{(i,j)}$	$\frac{1}{2} [I_{\pm}^i I_{\pm}^j]$

3.3.3 The Hamiltonians

Now that we know how to move our interaction into any frame we desire, we can describe the system Hamiltonians in the **PAS** frame. Before we will discuss the specific interactions, we again must address the rotating frame/truncation in the new basis. In the last section, there was no mention of any spin system or an NMR system (except of some small enlightenments). The above discussion is general for any Hamiltonian, so in the absence of any ‘rotating frame’ transformation, the final Hamiltonian will be of the form of Eq. 3.57. If the spatial components can be separated from the other components, then the rotation discussion and Eq. 3.59 holds. However, the rotations *DO NOT* effect the final energy spectrum of the Hamiltonian. Applying a large magnetic field removes the spherical symmetry of the Hamiltonian in Eq. 3.57, so that now the spectrum of the Hamiltonian has a directional dependence. To show this we need to look at the spherical spin tensors basis (T_l) shown in Table 3.3 and how they relate to the Cartesian basis in Eq. 3.51.

Zeeman

The Zeeman interaction is the one responsible for the symmetry breaking. Like Eq. 3.3 except that we desire an energy term, not a torque, the Zeeman Hamiltonian is

$$\mathbf{H}_{zee} = \gamma \mathbf{I} \cdot \mathbf{B} \quad (3.67)$$

Again, if \mathbf{B} is large and static, then all the remain interactions will be truncated with respect to this axis. For simplicity $\mathbf{B} = \mathbf{B}_z$, thus the main Hamiltonian has a term proportional to I_z . Using the fact that the first order perturbation theory only keeps those terms that commute with the main Hamiltonian, in this case I_z . Looking at Table 3.3 only $T_{0,0}$ and $T_{2,0}$ survive this truncation.

Chemical Shift Anisotropy

The Chemical Shift Anisotropy (CSA) Hamiltonian is caused by the electronic shielding around the nucleus. The electron cloud slightly deforms in a field causing shifts in the offset in 3 directions in the PAS. In Cartesian space we still perform the standard $\mathbf{I} \cdot \mathbf{B}$, but with respect to the PAS system.

$$\mathbf{H}_{CSA} = \mathbf{I}^i \cdot \mathbf{C}^i \cdot \mathbf{B} \quad (3.68)$$

where \mathbf{C}^i is the chemical shielding tensor on spin i

$$\mathbf{C}_{cart}^{i,PAS} = \begin{pmatrix} \delta_x & 0 & 0 \\ 0 & \delta_y & 0 \\ 0 & 0 & \delta_z \end{pmatrix} \quad (3.69)$$

In the spherical basis this interaction reduces to

$$\mathbf{H}_{CSA} = \delta_{iso}(\mathbf{I}^i \cdot \mathbf{B}) + A_{2,0}^{CSA,i} T_{2,0}^i \quad (3.70)$$

Even though the η term does not explicitly appear in the original Hamiltonian, upon a rotation (where all m components become mixed), it will. Excluding the molecular rotation, and the lab frame rotations, we can get a rotor frame angular dependence of the frequencies (rad/sec) to be

$$\omega^{csa} = 2\pi\delta_{iso} + \pi\delta_{ani} [3\cos^2\theta - 1 + \eta\sin^2\theta\cos(2\phi)]. \quad (3.71)$$

Scalar Coupling

Scalar coupling (or J) comes as a 2 atom, through bond interaction. There is no equivalent of this interaction in the classical sense because it is a result of the anti-symmetry of the electron (a purely quantum mechanical effect). The atoms must be inequivalent for one to observe this effect, so atoms with the same chemical shift do not have a J. Furthermore, if the two atoms have huge chemical shift differences when compared to the J coupling, then the J-coupling is truncated again with respect to the isotropic part of the chemical shifts. This is called ‘weak’ coupling, the other case being ‘strong’. For most NMR the J-coupling is considered solely isotropic, but there can easily be an electron cloud distortion like the CSA, so there is an anisotropic component as well. Below in Eq. 3.72 is the weak coupling limit where we have assumed the high magnetic field (and thus the Chemical shifts) are along the z-axis. Eq. 3.73 shows the strong case.

$$\mathbf{H}_{weak}^J = \delta_{iso}^{i,j} I_z^i I_z^j + A_{2,0}^J [2I_z^i I_z^j] \quad (3.72)$$

$$\mathbf{H}_{strong}^J = \delta_{iso}^{i,j} \mathbf{I}_i \cdot \mathbf{I}_j + A_{2,0}^J [3I_z^i I_z^j - \mathbf{I}_i \cdot \mathbf{I}_j] \quad (3.73)$$

Dipole Coupling

The dipole interaction in a high field looks much like Eq. 3.33, except that we are interested in the relative orientation of the two nuclei spin degree of freedom ($\mathbf{I}_i, \mathbf{I}_j$). So the $\mu \cdot \hat{r}$ terms switch to $\mathbf{I}_i \cdot \mathbf{I}_j$ terms. Much like J couplings, there are two extremes. For two homonuclear spins the scale of the dipolar interaction is usually the same as the chemical shift, so no chemical shift truncation will occur (Eq. 3.74). For hetero-nuclear dipole systems, the chemical shift difference is in the MHz, where as dipole-dipoles are on the order kHz. The hetero-nuclear coupling is truncated with respect to the chemical shift difference on the two hetero nuclear spins (Eq. 3.75). The dipolar coupling is symmetric about the z-axis, therefore there will be no η terms. Also, there is no part of the total dipolar Hamiltonian invariant under rotations, therefore there is no isotropic component.

$$\mathbf{H}_{hom}^D = A_{2,0}^D T_{2,0}^{i,j} = \frac{\omega_D^{i,j} (1 - 3 \cos^2 \theta)}{2} [3I_z^i I_z^j - \mathbf{I}^i \cdot \mathbf{I}^j] \quad (3.74)$$

$$\mathbf{H}_{het}^D = A_{2,0}^D T_{2,0}^{i,j} = \omega_D^{i,j} (1 - 3 \cos^2 \theta) [I_z^i I_z^j] \quad (3.75)$$

where $\omega_D^{i,j}$ is

$$\omega_D^{i,j} = \frac{\gamma_i \gamma_j \mu_o \hbar}{4\pi |r_i - r_j|^3} \quad (3.76)$$

Quadrupole

The quadrupolar coupling is due to electric field gradients around a single nucleus, and it only effect nuclei with spin $> 1/2$. If the gradient is 0 (spherical) then the interaction is also 0, therefore there is no isotropic component, but there can be an asymmetry (η) to the gradient. The anisotropic component (the gradient along the z-axis) is

$$\delta_z^Q = e^2 q Q = \frac{2}{3} 2I(2I - 1) \omega_Q \quad (3.77)$$

where e is the charge of an electron, qQ is the actual gradient value, and I is the spin of the nucleus, and ω^Q is the coupling constant. Simulations need only ω^Q which can be expressed as

$$\omega_Q = \frac{3\delta_z^Q}{2I(2I-1)}. \quad (3.78)$$

The first order truncated Hamiltonian is then

$$\mathbf{H}_1^Q = A_{2,0}^Q T_{2,0} = \omega_Q (1 - 3\cos^2\theta) [3I_z^2 - I_e(I(I-1))]. \quad (3.79)$$

The quadrupole interaction tends to be very large, on the order of MHz, the same order as our magnetic field. Our truncation approximation breaks down, so the second order quadrupole is needed for an accurate description. The second order effect is proportional to $\frac{\omega_Q^2}{\gamma B_z}$ and includes contributions of all the commutators of the basic spin tensors that commute with the Zeeman interaction (terms proportional to $T_{2,1}T_{2,-1}$ and $T_{2,2}T_{2,-2}$). The functional form of this interaction can be broken down into a total rank 2 component and a total rank 4 component. Note that the rank 4 component is obtained from the rank 2 components via rules for tensor multiplication [30]. The second rank quadrupolar Hamiltonian is

$$\begin{aligned} \mathbf{H}_2^Q &= \frac{\omega_Q^2}{\gamma B_z} \left\{ [A_{2,-1}^Q A_{2,1}^Q] [T_{2,-1} T_{2,1}] + [A_{2,-2}^Q A_{2,2}^Q] [T_{2,-2} T_{2,2}] \right\} \\ &= \frac{\omega_Q^2 I_z}{\gamma B_z} \left[A_{2,-1}^Q A_{2,1}^Q (4I_e I(I+1) - 8I_z^2 - I_e) + A_{2,-2}^Q A_{2,2}^Q (2I_e I(I+1) - 2I_z^2 - I_e) \right] \end{aligned} \quad (3.80)$$

Computationally, the second order quadrupole under the many rotations is best done in the Cartesian basis where we only need to multiply 3×3 matrices rather than the many 5×5 rotations. Let \mathbf{Q} be the quadrupolar Cartesian tensor in the **PAS** frame as

$$\mathbf{Q} = \begin{pmatrix} \frac{\eta-1}{2} & & \\ & -\frac{\eta+1}{2} & \\ & & 1 \end{pmatrix}. \quad (3.81)$$

After our series of rotations all the elements will be mixed and non-zero, the first and second order quadrupole can be reduced to

$$\begin{aligned} \mathbf{H}_1^Q &= \omega_Q \mathbf{Q}(2, 2) T_{2,0} \\ \mathbf{H}_2^Q &= \frac{\omega_Q^2}{\gamma B_z} c_4 \left[\frac{(\mathbf{Q}(0,0) - \mathbf{Q}(1,1))^2}{4} + \mathbf{Q}(0,1)^2 \right] \\ &\quad - \frac{\omega_Q^2}{\gamma B_z} c_2 [(\mathbf{Q}(0,2)^2 - \mathbf{Q}(1,2)^2)] \end{aligned} \quad (3.82)$$

where $\mathbf{Q}(\mathbf{i}, \mathbf{j})$ indicate the matrix elements within \mathbf{Q} and c_4 and c_2 are

$$\begin{aligned} c_2 &= 2I_z (4I(I+1) - 8I_z^2 - I_e) \\ c_4 &= 2I_z (2I(I+1) - 2I_z^2 - I_e) \end{aligned} \quad (3.83)$$

Pulses

Much like the classical case in the rotating frame, our magnetic pulse can be described simply by giving each direction (\hat{i}) the corresponding spin tensor. The pulse Hamiltonian on spin j is then given as

$$\mathbf{H}_{\mathbf{RF}}^j = \omega_1 (\cos \phi I_x^j + \sin \phi I_y^j) + \Delta\omega_1 I_z^j \quad (3.84)$$

where $\omega_1 = \gamma_j B_1$, ϕ a the phase factor, and $\Delta\omega_1$ is the offset.

There are typically two extremes of pulses when we treat the problem computationally. The first are called ‘hard’ pulses where the pulse is very strong (much larger than any other Hamiltonian) and of short duration such that the effective Hamiltonian for this small time is only the pulse. The second case, a called ‘soft’ pulse, is the opposite extreme where the pulse is either applied for long times and/or is relatively weak. In this case, the total Hamiltonian is the system Hamiltonian plus the pulse.

Other Second Order Effects

If the main field is weak enough (or the interactions strong enough), then all the interactions will have second order effects much like the quadrupole. Not only will they need to be treated individually to second order, but the *total* Hamiltonian will have to be treated to second order. This results in very messy expressions for the Hamiltonians that require much care to evaluate. A good reference how to treat the second order components is by Sungsool and Frydman[31].

3.4 NMR Initial Conditions

3.4.1 Quantum

NMR measures bulk magnetic properties of nuclei, not a single nucleus. Much of our quantum discussion above appeared as if we were treating one or two nuclei, when in fact the Hamiltonians apply to the bulk sample of identical particles. The density matrix ρ is the quantum mechanical way to treat many quantum states in terms of populations of states rather than explicit eigenstates. The density matrix at equilibrium is given by a simple Boltzmann distribution

$$\rho_o = \frac{\exp\left[\frac{-\mathbf{H}}{k_B T}\right]}{\text{Tr}\left[\frac{-\mathbf{H}}{k_B T}\right]} = \frac{\exp\left[\frac{-h\gamma B_z}{k_B T} \sum_i I_z^i\right]}{\text{Tr}\left[\frac{-h\mathbf{H}}{k_B T}\right]} \quad (3.85)$$

$$\rho_o \approx \frac{I e^{-\left[\frac{h\gamma B_z}{k_B T} \sum_i I_z^i\right]} + \frac{1}{2} \left[\left(\frac{h\gamma B_z}{k_B T}\right)^2 \sum_{j,i} I_z^j I_z^i\right] - \frac{1}{6} \left[\left(\frac{h\gamma B_z}{k_B T}\right)^3 \sum_{j,i,k} I_z^k I_z^j I_z^i\right] + \dots}{\text{Tr}\left[\frac{-h\mathbf{H}}{k_B T}\right]}$$

where k_B is Boltzmann constant and T is the temperature. We have also assumed the Hamiltonian is simply the Zeeman Hamiltonian as it is the largest of all the other interactions. The indexes i, j, k sum over the entire number of spins. At this point most of NMR makes a fundamental approximation, the ‘high temperature limit’ where $k_B T \gg \gamma B_z$.

Thus the only term that contributes any component to the density matrix is the first term, which for a B_z of 7 Telsa, $T = 300K$, and one mole of nuclei is about $4.3 * 10^{-5}$. The deviation from an equal distribution is only $4.3 * 10^{-5}$, so ignoring any spins that we cannot manipulate or measure (the Identity (I_e) term), we only have about 1 in every 10^5 spins that we can control. The reduced density matrix is then simply

$$\rho_o \approx \frac{-\hbar\gamma B_z}{k_B T} \sum_i I_z^i / 2 \left(\frac{\hbar\gamma B_z}{k_B T} \right) \approx -\frac{1}{2} \sum_i I_z^i \quad (3.86)$$

Since we are dealing with identical particles, the sum over I_z is easily reduced to a single I_z matrix with the implication that when we effect this term we effect every spin. So the Hamiltonians discussed above are valid to this reduced spin matrix as well as the individual nuclei.

Eq. 3.86 is usually the initial condition in most NMR situations. However, certain experimental observations lead Warren[32] to include higher order terms of the density matrix. This results in an explanation for certain cross peaks in 2D NMR[33, 34, 35, 36] and new imaging techniques[37, 38, 39]

3.4.2 Classical

In the classical case we are concerned with the total magnetization of a volume, or a single spin. The magnitude of a single spin's magnetic moment is simply a nuclear Bohr magneton. The quantum density matrix picture describes a polarization difference. This polarization difference manifests itself as a bulk magnetization of the form[3, 40]

$$M_o = \frac{N\gamma^2 \hbar^2 I(I+1)B_o}{3\mu_o k_B T} = \chi \frac{B_o}{\mu_o} = \chi H \quad (3.87)$$

where N is the number of spins and I is the nuclear spin. This value, like the polarization is very small. Figure 3.3 shows this magnetization as a function of temperature, concentration,

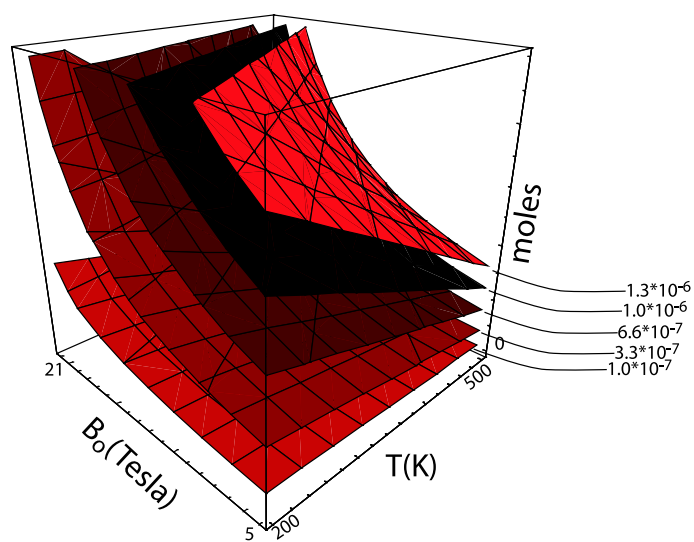


Figure 3.3: Magnetization in iso-surfaces versus the applied magnetic field, B_0 , the temperature T , and number of moles.

and the applied magnetic field.

Chapter 4

NMR Algorithms

4.1 Classical Algorithms

4.1.1 Eigenvalue Problem

The most general form for Eq. 3.1 can be given as a matrix equation

$$\frac{d\mathbf{M}}{dt} = -\gamma \begin{pmatrix} B_{xx} & B_{xy} & B_{xz} \\ B_{yx} & B_{yy} & B_{yz} \\ B_{zx} & B_{zy} & B_{zz} \end{pmatrix} \cdot \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix}. \quad (4.1)$$

This is a standard tensor equation

$$\frac{d\mathbf{M}}{dt} = \mathbf{B} \cdot \mathbf{M} \quad (4.2)$$

and due to the properties of the cross product we know that \mathbf{B} is an antisymmetric matrix (i.e. $B_{ij} = -B_{ji}$). We also know that \mathbf{B} is always real as it represents a real physical quantity. We can easily solve this equation using standard the eigensystem, where we know that there is some transformation matrix Λ such that

$$\mathbf{B} = \Lambda \cdot \Omega \cdot \Lambda^{-1} \quad (4.3)$$

where Ω is a diagonal matrix. Ω are the eigen-frequencies and the matrix Λ are the eigen-vectors of \mathbf{B} . Because of the anti-symmetry of \mathbf{B} , Λ forms a set of orthonormal basis states equivalent to our Cartesian set. If we transform our Cartesian set into this eigen-basis via

$$\tilde{\mathbf{M}} = \Lambda \cdot \mathbf{M}, \quad (4.4)$$

the equations of motion become

$$\begin{pmatrix} \dot{\tilde{M}}_1 \\ \dot{\tilde{M}}_2 \\ \dot{\tilde{M}}_3 \end{pmatrix} = \begin{pmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{pmatrix} \cdot \begin{pmatrix} \tilde{M}_1 \\ \tilde{M}_2 \\ \tilde{M}_3 \end{pmatrix}. \quad (4.5)$$

This has the trivial solution given at $t = 0$, $\tilde{\mathbf{M}} = (\tilde{M}_1^o, \tilde{M}_2^o, \tilde{M}_3^o)$ of

$$\begin{pmatrix} \tilde{M}_1(t) \\ \tilde{M}_2(t) \\ \tilde{M}_3(t) \end{pmatrix} = \begin{pmatrix} \tilde{M}_1^o e^{\omega_1 t} \\ \tilde{M}_2^o e^{\omega_2 t} \\ \tilde{M}_3^o e^{\omega_3 t} \end{pmatrix}. \quad (4.6)$$

The final step is then to transform back to our Cartesian basis to get the solutions in a space we can visualize

$$\mathbf{M}(t) = \Lambda^{-1} \cdot \tilde{\mathbf{M}}(t), \quad (4.7)$$

Another equally valid, and algorithmically simple, is the solution directly in the Cartesian basis of

$$\mathbf{M}(t) = e^{\mathbf{B}t} \mathbf{M}^o \quad (4.8)$$

This is a general solution to the Bloch equations. It requires, numerically, to perform a matrix diagonalization (or matrix exponentiation) which for one spin is very simple, however for many spins, the matrix becomes huge, rendering this method unusable especially if \mathbf{B} is a function of time, as we then must perform a numerical matrix integration of the form

$$\mathbf{M}(t) = e^{\int_0^t \mathbf{B}(t') dt'} \mathbf{M}^o \quad (4.9)$$

which requires *many* matrix diagonalizations and is prohibitively numerically expensive for large (i.e. hundred to thousands) of spins.

Looking at the form of these equations, you may think that this very large matrix is simply many 3×3 sub matrices along the diagonal. This is true so long as there are no spin-spin interactions, and spin-spin interactions are what make NMR interesting in the first place. So it seems we need another method to solve our systems of equations.

4.1.2 ODE solvers

Because the classical time evolution we wish to probe is a first order ordinary differential equation (ODE), to evolve the system we simply need a differential equation solver. The basic property of an ODE solver is that it marches through time using the approximation that step size is small enough such that the integrating function is constant.

ODE solvers come in many varieties, here we are concerned only with the ‘Initial Value Problem’ where we have an initial condition and that is all we know at the beginning. Other algorithms treat more than one point of knowledge (a n-point boundary value problem). There are an abundance of such solvers all of them with certain accuracy, efficiency and usefulness.

For our initial value problem there are a few subclasses of solvers

- **Implicit**–Requires a guess of the k point in order to evaluate a correct point at k , where k is some step in the integration (for us k is always t , time). These are sometimes called predictor–corrector methods because they must ‘guess’ the k value at least initially (although the guess can be quite educated), then correct the ‘guess.’
- **Explicit**–An explicit need only the previous, $k - 1$, point(s) to calculate the next one,

k .

- **Semi-Implicit**—Uses a tad of both integration techniques.

The simplest Initial Value Problem ODE solver is the Euler solver. It is essentially worthless in real life applications because it is much too inefficient, but is the basic model for all others that follow it. Given an ODE like

$$\frac{dy}{dt} = f(y, t) \quad (4.10)$$

and an initial value

$$y(t_o) = x \quad (4.11)$$

then we can approximate the next point, Δt , from t_o in an explicit fashion as

$$y(t_o + \Delta t) = y(t_o) + \Delta t * f(y(t_o), t_o) \quad (4.12)$$

or in an implicit fashion

$$y(t_o + \Delta t) = y(t_o) + \Delta t * f(y(t_o + \Delta t), t_o + \Delta t). \quad (4.13)$$

The Euler solver is a linear solver. You can see the implicit formula Eq. 4.13 requires a guess at the starting value for $y(t_o + \Delta t)$ to be useful. Both forms demonstrate how most ODE solvers function. The typical difference from one ODE solver to the next is how many other $f(y_n, t_n)$ in between t_o and $t_o + \Delta t$ are sampled. Each sampled point would have a series of coefficients associated with it. To extend the Euler solver, we could simply split Δt in half, and use two function evaluations, thus the new coefficients would be $\frac{1}{2}$ rather than 1. The number of sampled points determines the order of the algorithm. Taking 4 function evaluations between t_o and $t_o + \Delta t$ is a fourth order solver.

The three classes of solvers have their benefits and hardships. Implicit methods can usually handle stiff ODEs. A stiff ODE is one that has two or more solutions that differ tremendously in their rates of evolution (one is very slow, the other very fast). Explicit methods cannot treat these sorts of equations efficiently, because the time step must be very small. The small time step is required for explicit solvers because one must be certain not to ‘jump’ over the fast solution. This problem is reminiscent of the Nyquist Frequency problem encountered in experimental spectra in NMR where if the time step is too large, higher frequencies appear as much lower frequencies in the resulting spectrum. Implicit methods perform better because we initially must guess the solution, then correct it and continue this prediction–correction scheme until the solution stabilizes.

Implicit methods, however, tend to be very hard to start accurately (because they need an initial guess and there is no previous points to get a educated guess from). This results a large algorithmic complexity that can slow down the solvers and other accuracy difficulties. Explicit methods can operate very efficiently and without the starting problems so long as the system is not stiff.

It turns out that the NMR classical fields do not lead to stiff equations, so we can freely use any explicit method we desire. The two basic solvers are the Runge-Kutta and Richardson Extrapolation. Before we describe these two solver, lets review a few algorithmic points/sections that one should be aware of when treating ODE solvers.

- **Errors**–To know our accuracy in the results, we must have some way to measure errors. This is typically done by using the information of the next order. For example if we had a 4th order algorithm, then we could monitor the error by the difference between the 4th and 5th order results.

- **Time step adjustment**—In order for an ODE algorithm to be efficient, it should be able to take large time steps when the solution is slowly evolving and shrink the time step when the solution is evolving faster. We can simply adjust the step size based on the ratio of a desired accuracy and our error monitor[41].
- **Kernel**—The kernel should operate independently of any errors or time step adjusters. It should simply produce the next point.

The 5th order embedded Runge–Kutta[42] algorithm uses a total of 6 function evaluations and has the error estimate ‘embedded’ into the formula. By embedded we mean it uses all 6 function evaluations to produce a 5th order result and a 6th order error estimate. This kernel is a standard work horse for most any problem. It is very robust, but can be slow as it requires 6 function evaluations. A much better kernel is the Bulirsh–Stoer–Richardson–extrapolation method[43, 44]. This kernel evaluates n points between t and $t + \Delta t$ (usually equally spaced), storing the final value, $y(t + \Delta t)$. It then performs $n + 2$ points, stores another point at $y(t + \Delta t)$, and so on up to $n + m$ where m can be arbitrary (i.e. $m = 2, 4, 6, 8, \dots$, for hard problems m should be 12, for easier problems m can be 6). Given these m point at $y(t + \Delta t)$ we can fit an m –order polynomial and use our fitted function to extrapolate to the case where $m \rightarrow \infty$ (or as $\Delta t_m \rightarrow 0$). The benefit here over the Runge–Kutta kernel is Δt can be quite large due to the extrapolation, thus we can minimize many function calls. The only problem with this kernel is that it is not nearly as robust as the Runge–Kutta, and even slightly stiff problems cause this method to fail.

That is all the algorithmic complexity we really need to solve the classical form of Bloch equation, for a review of even more ODE solvers one should look to these two references [45] and [46].

4.2 Quantum Algorithms

Unlike the classical case where everything can be placed in to one ODE solver and the trajectories of N spins can be easily calculated, the Quantum Mechanical nature prohibits the calculation of arbitrary N . For 10 spin 1/2 nuclei the matrix sizes are 1024×1024 equivalent to about 35000 classical spins. As shown in chapter 2, the matrix multiplication can be quite time consuming. Unless other techniques are used to address the problem, even a 10 spin system may prove prohibitively long. In this section, we do not wish to use any theoretical approximations to simplify the problem because we are more interested in the exact solution numerically, not the approximation.

4.2.1 The Direct Method

The solution to our Eq. 3.50 is given by

$$\rho(t) = U\rho_oU^{-1}. \quad (4.14)$$

where U is called the propagator and is defined as

$$U(t_o, t_o + \Delta t) = T \exp \left[-i \int_{t_o}^{t_o + \Delta t} H(t') dt' \right]. \quad (4.15)$$

Here, T is the Dyson Time Ordering operator, and maintains that a propagator is multiplied in correct time order. By time order it is easiest to look at the approximation to the integral solution.

$$U(t_o, t_o + \Delta t) = \prod_{k=1}^{k\delta t = \Delta t} \exp \left[-i\delta t H \left(\frac{t_o + k\delta t}{2} \right) \right] \quad (4.16)$$

where δt is much smaller than any time dependence in H . The product cannot be performed in any other order than in the series given. This product requires two time consuming operations to calculate. The first is the matrix exponential which takes about N^3 operations

to complete, the second is the matrix product, another N^3 operations. Numerically we can solve any NMR problem in this fashion, calling it the ‘Direct Method.’

For many cases of NMR simulation, this method is not as bad as it sounds. If the Hamiltonian is not time dependant, then the integral vanishes only to produce a constant, Δt , multiplication factor. This reduces the problem to a single matrix exponentiation. Most liquid state or static solid state NMR simulations can be calculated very quickly using the direct method. We run into computational trouble when time dependence is introduced into the system.

4.2.2 Periodicity and Propagator Reduction

Periodic Hamiltonians appear over and over in NMR. In this section we will go over the few algorithmic tricks we can play with periodic Hamiltonians and their propagators. By periodic we mean that

$$H(t) = H(t + \tau_p) \tag{4.17}$$

where τ_p is the period. There are three cases where using periodicity reduces the total number of calculated propagators necessary to have a complete description of the dynamics.

The typical NMR experiment requires observation of the system at specific intervals of time, Δt . Given that every observation is the same distance in time away from the last observation and that the time dependence of the Hamiltonian is periodic, we can have 3 possible situations. Figure 4.1 shows the three possible situations given those conditions if we wish to observe the Hamiltonian at some rational fraction of the this periodicity time ($\frac{m}{n}\tau_p$). In general the m factor, called the periodicity factor, represents the number of periods of length τ_p that must occur before an observation is synchronous with τ_p . The n factor,

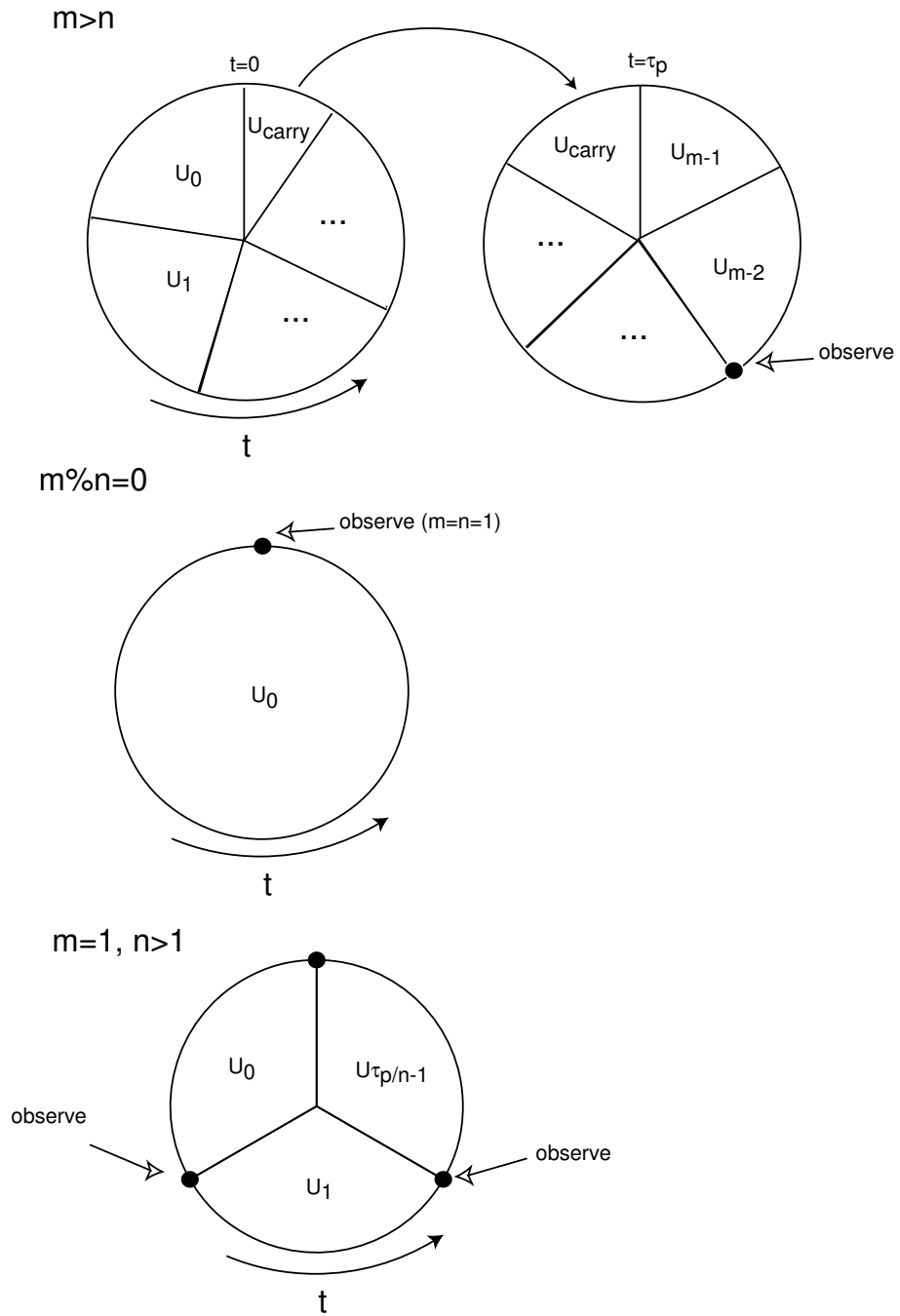


Figure 4.1: Various propagators needed for an arbitrary rational reduction.

called the observation factor, represents the number of the m sub-propagators necessary to advance one observation time step. Each sub propagator (U_i in Figure 4.1) is assumed to be calculated via the direct method.

Point-to-Point, $\text{mod}(m, n) = 0$

The simplest way to use the periodicity is to realize that at each interval n of τ_p (for a total time of $n\tau_p$) the propagator equation reduces to

$$U(n\tau_p) = (U(\tau_p))^n. \quad (4.18)$$

This means we only have to evaluate the computationally expensive integral in Eq. 4.16 n times, and we can take much larger steps in multiples of τ_p . This method I will call ‘Point-to-Point’ (PtoP) as we can only get the dynamics at the specific point of $t = n\tau_p$, not arbitrary points. The method is well suited for any rotor synchronized pulse sequences where the pulses we apply and the rotor spinning frequency are synchronized. Here we only have to store and calculate n propagators.

Rational Reduction, $m > n$ (also $n > m$)

Many times one wants to get dynamics in-between the period and we may be reduced back to using the ‘Direct Method’ for such tasks. Consider still that our Hamiltonian is periodic with a cycle of τ_p . The condition is where $n > m$ is also the case where $n < m$, all we have to do here is switch the indices and perform an extra observation step at the beginning of the sequence. In this case there will be a propagator (U_{carry} in Figure 4.1) that will span over a τ_p . To see how the reduction works first consider a normal propagation series in time shown in Table 4.1.

Table 4.1: Time propagation using individual propagators via the Direct Method

time step	Propagator series
1	U_0
2	U_1U_0
3	$U_2U_1U_0$
4	$U_3U_2U_1U_0$
5	$U_4U_3U_2U_1U_0$
6	$U_5U_4U_3U_2U_1U_0$
7	$U_6U_5U_4U_3U_2U_1U_0$
8	$U_7U_6U_5U_4U_3U_2U_1U_0$
9	$U_8U_7U_6U_5U_4U_3U_2U_1U_0$
10	$U_9U_8U_7U_6U_5U_4U_3U_2U_1U_0$
11	$U_{10}U_9U_8U_7U_6U_5U_4U_3U_2U_1U_0$

Table 4.2: A reduced set of individual propagators for $m = 9$ and $n = 7$

observation time step	Propagator series
1	$U_6U_5U_4U_3U_2U_1U_0=U_0^T$
2	$U_4U_3U_2U_1U_0U_8U_7 * U_0^T=U_1^T$
3	$U_2U_1U_0U_8U_7U_6U_5 * U_1^T=U_2^T$
4	$U_0U_8U_7U_6U_5U_4U_3 * U_2^T=U_3^T$
5	$U_7U_6U_5U_4U_3U_2U_1 * U_3^T=U_4^T$
6	$U_5U_4U_3U_2U_1U_0U_8 * U_4^T=U_5^T$
7	$U_3U_2U_1U_0U_8U_7U_6 * U_5^T=U_6^T$
8	$U_1U_0U_8U_7U_6U_5U_4 * U_6^T=U_7^T$
9	$U_8U_7U_6U_5U_4U_3U_2 * U_7^T=U_8^T$
10	$U_6U_5U_4U_3U_2U_1U_0 * U_8^T=U_0^T * U_8^T$

Consider now the case where $m = 9$, and $n = 7$. Here we require 7 time steps for each observation, and we know that every 9 observations the propagator sequence repeats.

Table 4.2 shows a clearer picture of this situation.

From this table, we can see that based on the repetition number m we only need to calculate m sub-propagators, U_i , each with a relatively small Δt of τ_p/m using the direct method. Then calculate 9 more larger time step propagators (that span a Δt of $n/m\tau_p$) of propagators, U_i^T using simple matrix multiplication. The hardest calculation is the using

the direct method to calculate the initial 9 propagators, and this step cannot be avoided. However the remaining 9 larger propagators require 63 extra matrix multiplications to calculate. If the matrices are large, this can be an expensive operation. To further reduce the number of matrix multiplications, we can use that fact that one typically calculates the U_i in sequence (i.e. $i = 0$, then $i = 1, \dots, i = m - 1$) and that no matter what other tricks we play, we know that we will have to calculate $U_6 U_5 U_4 U_3 U_2 U_1 U_0 = U_0^T$. Along the way to calculating U_0^T we can easily store each sub sequence (U_0 , $U_1 U_0 = U_{1,0}$, $U_2 * U_{1,0} = U_{2,1,0}$, ...). Call these sequences the ‘forward’ sequences. Calculating these results in at least m matrix multiplications. Calculating the ‘backwards’ sequences (U_8 , $U_8 U_7 = U_{8,7}$, $U_{8,7} * U_{6,*} = U_{8,7,6}$, ...) is also relatively simple, because we have stored all the U_i ’s and we can use the same procedure as in calculating the ‘forward’ sequences resulting in another m multiplications.

Calculating the forward propagators seems to at least make intuitive sense, so why did we calculate the seemingly unnecessary backwards propagators? The next step is to realize that inside each U_i^T are sequences of U_i that repeat many times. For instance the sub sequence $U_8 U_7$ (a backwards propagator) appears 6 times in Table 4.2, so we could save at least 5 matrix multiplications by simply storing the $U_8 U_7$ result the first time we calculate it. One can look over this entire set of propagators U_i^T looking at the ones we have already saved from the above operations to see that there will be some optimal set of backwards and forward propagators that reduce the total number of matrix multiplications needed.

Of course to figure out this rational reduction of propagators, no propagators are actually necessary, simply the indexes i , the ‘forward’ labels, $(i, j, k \dots)$ and the ‘backward’ labels (k, j, i, \dots) . Given m and n these can all be automatically generated as simple integers,

Table 4.3: Matrix Multiplication (MM) reduction use rational reduction

m	n	Original MMs	Reduced MMs	% reduction
4	3	12	8	33%
6	5	30	14	53%
11	5	55	39	29%
21	5	105	89	15%
9	7	54	21	61%
11	7	77	41	47%
13	7	91	55	40%
104	7	728	692	5%
104	103	10712	308	97%

as can each U_i^T sequence. The problem is then reduced to finding each forward and backward set of indexes inside each U^T set of indices. Comparing the number of multiplications required by each set and pick the minimum. After the minimum is found, it is up to the user to provide the set of U_i , $U_{i,j,k\dots}$ and $U_{k,j,i,\dots}$, and the algorithm can then generate the basic m propagators of observation. A C++ class is given in Appendix A.2.2 that performs this task.

Table 4.3 shows the effective reduction in the matrix multiplications using this technique for a range of m and n . From the Table it is easy to see that if the periodicity factor m is much different then the observation factor n then the rational reduction does not produce much improvement because we still need to calculate the m sub-propagators. Figure 4.2 shows better picture as to the effectiveness of the rational reduction method. In the Figure, the spikes are where m becomes a multiple of n and we get a PtoP method.

Sub-Point-to-Point, $m = 1, n > 1$,

This case is another special case closely related to the PtoP method because it essentially means that n is a factor of τ_p , so each n sub-propagators calculated leads us

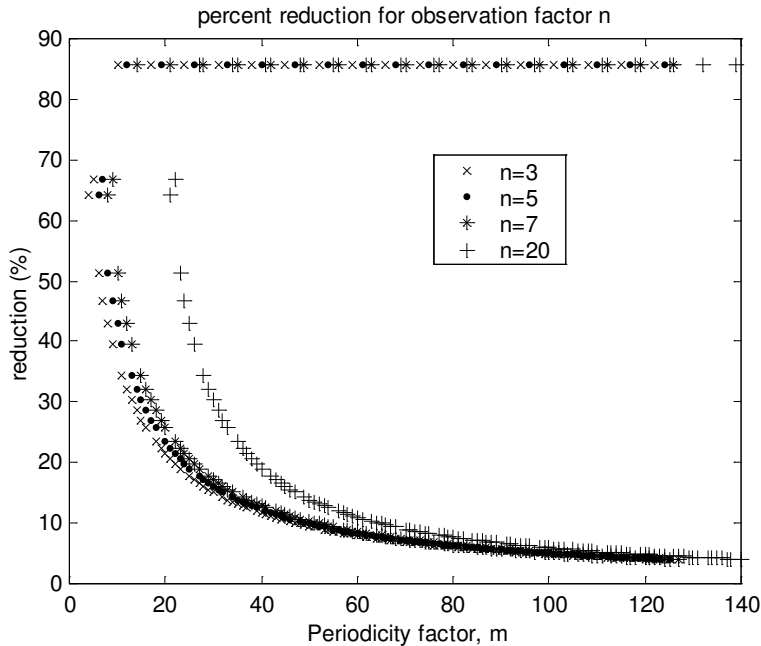


Figure 4.2: Effectiveness of the rational propagator reduction method.

back to the τ_p . This special conditions leaves us only to calculate n sub-propagators and the various combinations for an optimal minimization of the matrix multiplications. For instance if $n = 5$ then we need to calculate the n sub-propagators ($U_0 \dots U_4$) each spanning a time of $\Delta t = \tau_p/n = \tau_p/5$. While we complete one τ_p cycle, we collect the sub multiplications shown in Table 4.4 using them in later observation points. This particular case is a typical NMR situation, and will be used later.

4.2.3 Eigenspace

The past methods have only been treating time dependence explicitly. We can easily do a transformation to treat the frequency domain. This is a natural transformation because our Hamiltonians all have the units of Hz and NMR typically collects data in equal spaced time steps. Because Hamiltonians are Hermetian, we know that all the eigenvalues

Table 4.4: For $m = 1$ and $n = 5$ we have this series of propagators necessary to calculate the total evolution

observe Point	Propogators
1	$U_0 = U_0^T$
2	$U_1 U_0 = U_1^T$
3	$U_2 U_1 U_0 = U_2^T$
4	$U_3 U_2 U_1 U_0 = U_3^T$
5	$U_4 U_3 U_2 U_1 U_0 = U_4^T$
6	$U_0^T U_4^T$
7	$U_1^T U_4^T$
8	$U_2^T U_4^T$
...	...

are real and the eigenvectors form an orthonormal basis. The next few methods use the eigenp-basis to potentially remove the explicit time dependence to avoid performing many matrix multiplications, when only a matrix digitalization is necessary.

Eigenvalue propagation

One special case involves non time dependant Hamiltonians, or inhomogeneous Hamiltonians. In this case the eigenvectors to not change in time, and we can easily write our Hamiltonian in terms of the eigenvectors and eigenvalues.

$$\mathbf{H} = \mathbf{\Delta} \cdot \mathbf{\Omega} \cdot \mathbf{\Delta}^\dagger. \quad (4.19)$$

where $\mathbf{\Delta}$ is a matrix of the eigenvectors, and $\mathbf{\Omega}$ is a diagonal matrix of the eigenvalues and the \dagger is the adjoint (complex transpose) of a matrix, which for unitary matrices is also the inverse. To propagate forward in time, we can use this property of unitary matrices and matrix exponentials

$$\exp[H] = \mathbf{\Delta} \exp[\mathbf{\Omega}] \mathbf{\Delta}^{-1}. \quad (4.20)$$

Placing this solution back into Eq. 4.15, we get

$$\rho(t) = \mathbf{\Delta} \exp[-it\mathbf{\Omega}] \mathbf{\Delta}^\dagger \rho \mathbf{\Delta} \exp[it\mathbf{\Omega}] \mathbf{\Delta}^\dagger \quad (4.21)$$

In NMR we can only detect certain components of the density matrix at a given time, those two components are the two axis where our coils sit, \hat{x} and \hat{y} . So we can only detect I_x and I_y . We then need to project out the component of our detection operator (which from now on will be called I_{det}) in the density matrix via the trace operator Tr .

$$I_{det}(t) = Tr[\rho(t)I_{det}^\dagger]I_{det} \quad (4.22)$$

Using Eq. 4.21, Eq. 4.22, and the cyclic permutation properties of the trace we find that

$$I_{det}(t) = Tr[\rho(\tilde{t})\tilde{I}_{det}^\dagger]I_{det}. \quad (4.23)$$

where the $\tilde{A} = \Delta A \Delta^\dagger$, a similarity transform. Our recorded signal, S is then just the Tr constant

$$S(t) = Tr[\exp[-it\Omega]\tilde{\rho}_o \exp[it\Omega]\tilde{I}_{det}^\dagger]. \quad (4.24)$$

This results in a sum over all over the diagonal of the multiplication inside the result. The multiplied result contains all the differences in frequencies in Ω with coefficients given by the $\rho(t)$ and I_{det} . Thus when we calculate this signal, we only need to be concerned with the multiplied quantities along the diagonal, turns a previously N^3 operation into an approximately N^2 operation.

$$Tr[AB] = \sum_i^N \sum_j^N A_{ij}B_{ji} \quad (4.25)$$

If we assume that we are sampling n steps the evolution in equal times of Δt , our signal in terms of explicit elements is

$$S(n\Delta t) = \sum_k^N \sum_j^N I_{kj}^{det} \rho_{jk}^o (\Phi_{kj})^n \quad (4.26)$$

where Φ_{kj} is the transition frequency matrix, $\Phi_{kj} = \exp(-i(\omega_k - \omega_j)\Delta t)$. An algorithm for this static case is shown in Appendix A.2.3.

Effective Hamiltonians

This method is an extension to the PtoP method. In the PtoP method, we have only one propagator necessary to calculate our evolution. To move in to observation point n of τ_p it is then necessary to multiply the propagator n times, resulting in $n * N^3$ operations. We can in principle invert the propagator to find the effective Hamiltonian, \mathbf{H}_{eff} for this one period using the matrix logarithm.

$$\mathbf{H}_{\text{eff}} = \frac{-\mathbf{i}}{\tau_p} \log(\mathbf{U}) \quad (4.27)$$

This effective form is now time independent on the period. Once we have the effective Hamiltonian, we can easily use this as the Hamiltonian in the static eigenvalue propagation discussed in section 4.2.3. Thus we avoid performing many matrix multiplications. There is a problem using this technique in general in that it has a very narrow frequency bandwidth. The largest frequency it can accurately obtain from the propagator is one that falls within the $\pm 1/\tau_p/2$ range. If the real Hamiltonian has frequencies outside this range (which it usually does) then they will be folded into the resulting spectrum and will result in a cluttering of spectral features. This problem leads to amplitude discrepancies in any frequencies of integer multiples of the period, as higher order multiples that are not in the range will be folded on top of ones that are in range.

This technique works very well when τ_p is very short when compared to the real Hamiltonian time dependence because our spectral window is quite large. If one can accurately claim this condition, no other technique can match this one for its speed in calculating the evolution.

Fourier components and the Floquet approximation

The Floquet method corrects the folding and amplitude problems of the effective Hamiltonian by working totally in frequency space. But before we can venture further into using frequency domain methods, we need to decompose our Hamiltonian into its Fourier components. Mind you, this is not necessarily where the Hamiltonian is diagonal, but where the Hamiltonian can be broken into a form

$$\mathbf{H} = \sum_m H_m e^{-im\phi}. \quad (4.28)$$

where ϕ is some phase factor, and H_m is a Fourier component of \mathbf{H} . Given our entire sequence of possible rotations in Eq. 3.66 we have these Fourier components already calculated as well as the phase factor. So the Fourier components of the Hamiltonian are nothing more the rotated l, m components.

Assuming that the time dependence is in the ϕ phase factor, and can be factored ($\phi = \phi(t) = \omega t$), we can write a Floquet Hamiltonian H_F [47, 48, 49, 50, 51] of the form

$$\langle pn | H_F | qm \rangle = n\omega \delta_{nm} \delta_{pq} + \langle p | H_{n-m} | q \rangle \quad (4.29)$$

where p, q are spin states and n, m and Fourier indices. Both n and m have the range $(-\infty, \infty)$ in integer multiples. Thus the Floquet Hamiltonian is an infinity sized matrix

which looks like

$$H_F = \begin{pmatrix} \ddots & \ddots & \ddots & & & & & & & & H_n \\ \ddots & H_0 + 2\omega & H_1 & H_2 & & & \ddots & & & & \\ \ddots & H_{-1} & H_0 + \omega & H_1 & H_2 & & & & & & \\ & H_{-2} & H_{-1} & H_0 & H_1 & H_2 & & & & & \\ & & & H_{-2} & H_{-1} & H_0 - \omega & H_1 & \ddots & & & \\ & & \ddots & & H_{-2} & H_{-1} & H_0 - 2\omega & \ddots & & & \\ H_{-n} & & & & & & \ddots & \ddots & \ddots & & \end{pmatrix} \quad (4.30)$$

To evolve this matrix we must diagonalize it, but this time we get the raw frequencies and amplitudes that are time *independent*, so we only have to diagonalize it once. However the matrix is infinitely sized, so computationally we must set an arbitrary size of $N \times N$. We then get N frequencies and amplitudes. If N contains all the necessary frequencies to describe our system Hamiltonian then this matrix truncation is valid, if not, then we must go to higher and higher N . Eventually we will hit a computational limit as diagonalization become prohibitively hard. We do have another simplification for this matrix, in that the only H_{n-m} terms that appear in NMR are usually $n - m = \pm 2$ for most interactions and $n - m = \pm 4$ for second order interactions (the second order quadrupole for instance). So the Floquet matrix is a banded matrix. It turns out that the size of H_F necessary to handle most normal NMR spin systems is much too large to be used efficiently. So this technique is not used much in computational NMR. It is still a valuable tool for theoretical studies [52, 53, 54] and simulation of small (1-2 spins) systems.

It can be used rather powerfully when there are only a few frequencies that describe the Hamiltonian. Cases like rotational resonance [53, 55, 56, 57] and multi-photon effects [3, 58].

4.2.4 Periodicity and Eigen–Space methods

This section will cover the blending of both aspects of periodicity in the Hamiltonians and the fact that they are easily decomposed Fourier components. In essence we wish to combine the aspects of both the propagator reductions discussed in section 4.2.2 and the Fourier methods discussed in 4.2.3.

COMPUTE

The COMPUTE algorithm was first proposed by Eden *et. al* in 1996[59]. First, let’s assume that we are observing in the regime in our periodic picture where $m = 1$ and $n > 1$. Eden shows we can use the sub–propagators along with the total period propagator to remove the frequencies wrapping and amplitude problems of the effective Hamiltonian method, as well as observe our system at times in-between periods.

To describe the algorithm in more detail, I will use the notation as in Table 4.4, and elucidated a bit more in Figure 4.3. First we note that we can separate the total period propagator U_{n-1}^T can be factored into its diagonal form via

$$U_{n-1}^T = \Gamma e^{-i\tau_p \Omega} \Gamma^\dagger \quad (4.31)$$

Our signal at each $k\Delta t$, where $\Delta t = \tau_p/n$, is then following the same discussion as in section

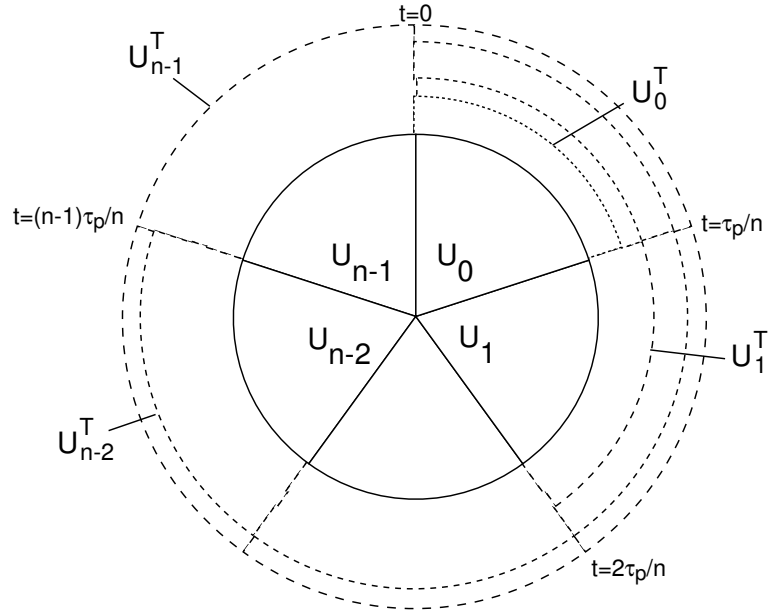


Figure 4.3: Diagram of one Hamiltonian period and the propagator labels used for the COMPUTE algorithm

4.2.3, and using the fact that $\Gamma\Gamma^\dagger = I_e$

$$\begin{aligned}
 S(k\Delta t) &= \text{Tr} [\rho^T(k\Delta t) I_{\text{det}}] \\
 &= \text{Tr} [U_k^T \rho_o (U_k^T)^\dagger I_{\text{det}}] \\
 &= \text{Tr} [I_e \rho_o (U_k^T)^\dagger I_e I_{\text{det}} U_k^T] \\
 &= \text{Tr} [\Gamma\Gamma^\dagger \rho_o (U_k^T)^\dagger \Gamma\Gamma^\dagger I_{\text{det}} U_k^T] \\
 &= \text{Tr} [\Gamma^\dagger \rho_o \Gamma\Gamma^\dagger (U_k^T)^\dagger I_{\text{det}} U_k^T \Gamma] \\
 &= \text{Tr} [\rho_o^T I_{\text{det}}^T]
 \end{aligned} \tag{4.32}$$

where we have defined these two important matrices

$$\begin{aligned}
 \rho_o^T &= \Gamma^\dagger \rho_o \Gamma \\
 I_{\text{det},k}^T &= \Gamma^\dagger (U_k^T)^\dagger I_{\text{det}} U_k^T \Gamma.
 \end{aligned} \tag{4.33}$$

The next part comes from the realization that the NMR signal is simply of sum of frequencies and amplitudes. There are no explicit frequencies in the form of the signal in Eq. 4.32,

but we are always free to multiply the signal by $1 = \exp[-i(\omega_{rs})t] \exp[i(\omega_{rs})t]$, where $\omega_{rs} = (\omega_r - \omega_s)$. When we do this we get

$$S(k\Delta t) = \sum_{r,s=1}^N f_{rs}^k \exp[i\omega_{rs}k\Delta t] \quad (4.34)$$

where

$$f_{rs}^k = (I_{\text{det},k}^T)_{rs} (\rho_o^T)_{sr} \exp[-i\omega_{rs}k\Delta t]. \quad (4.35)$$

Eden showed in Ref. [59] that this function is periodic with $k \rightarrow k + n$, and thus can be expanded as a discrete Fourier series as

$$f_{rs}^k = \sum_{j=-n/2+1}^{n/2} a_{rs}^j \exp[i2\pi kj/n] \quad (4.36)$$

This form can be easily inverted to give us the complex amplitudes, a_{rs}^j

$$a_{rs}^j = 1/n \sum_{k=-n/2+1}^{n/2} f_{rs}^k \exp[-i2\pi kj/n]. \quad (4.37)$$

We now have the the complex amplitudes and the system frequencies exactly for this specific Δt . In essence what we have done is 1) use the Effective Hamiltonian method to get the frequencies of within the period and 2) used the sub-propagators to correct these frequencies and amplitudes due to the wrapping problems of the effective approach. We did all this only calculating n propagators. We still must calculated the list of transformed detection matrices $I_{\text{det},k}^T$ and amplitude functions f_{rs}^k resulting in a few more multiplications.

The algorithm proceeds as follows

- Choose an n such that $n\Delta t = \tau_p$.
- Calculate all the U_k^T from the system Hamiltonian.
- Invert U_{n-1}^T using the matrix logarithm.

- Calculate frequency differences, ω_{rs} from the eigenvalues of the effective Hamiltonian and use eigenvectors to calculate ρ_o^T .
- Using the U_k^T propagators calculate and store all the $I_{\text{det},k}^T$ matrices.
- Using Eq. 4.35 calculate and store all the f_{rs}^k .
- To generate the observed spectrum simply apply Eq. 4.34 using the f_{rs}^k and ω_{rs} realizing that at $k > n$ that $f_{rs}^k = f_{rs}^{k+n}$.

A code example of this is not given, because of a more recent extension to this algorithm which we will discuss in the next section.

γ -COMPUTE

Up until now, we have made minor assumptions about the Hamiltonians we are dealing with: 1) they are periodic, and 2) the time dependence can usually be easily factored out of the rest of the Hamiltonian. In order to describe the γ -COMPUTE algorithm, I will need to expand the Hamiltonian more then I have thus far. In the section 4.2.6 I describe methods of integrating over space using various powder averages. The powder average is necessary to properly simulate all the various orientations of a single crystal in a real powder solid sample. This corresponds the molecule-to-rotor rotation described in section 3.3.2. Below we write out explicitly all the rotational sums in the spherical tensor basis if we only rotate the spatial degree of freedom

$$\mathbf{H}_m = \sum_l \left\{ \begin{array}{c} \left[e^{-im\gamma_{lab}} d_{m,m'}^l(\theta_{lab}) e^{-im'\phi_{lab}} \right] \\ \sum_{m'=-l}^l \sum_{m''=-l}^l \sum_{m'''=-l}^l \left[e^{-im'\gamma_{rot}} d_{m',m''}^l(\theta_{rot}) e^{-im''\phi_{rot}} \right] \\ \left[e^{-im''\gamma_{mol}} d_{m'',m'''}^l(\theta_{mol}) e^{-im'''\phi_{mol}} \right] \\ A_{m'''}^l \end{array} \right\} T_m^l \quad (4.38)$$

The typical ϕ_{lab} is the rotor spinning rate $\omega_r t$. Because the high magnetic field is assumed cylindrically symmetric, the γ_{lab} is arbitrary and constant, so we can easily choose 0. Condensing the molecule rotation into new \hat{A}_m terms, we get a more compact form of the full Hamiltonian.

$$\mathbf{H}_m = \sum_l \left\{ \sum_{m'=-l}^l \sum_{m''=-l}^l \left[d_{m,m'}^l(\theta_{lab}) e^{-im'\omega_r t} \right] \left[e^{-im'\gamma_{rot}} d_{m',m''}^l(\theta_{rot}) e^{-im''\phi_{rot}} \right] \hat{A}_{m''}^l \right\} T_m^l \quad (4.39)$$

Collecting terms we get

$$\mathbf{H}_m = \sum_l \left\{ \sum_{m'=-l}^l \sum_{m''=-l}^l \left[d_{m,m'}^l(\theta_{lab}) e^{-im'(\omega_r t + \gamma_{rot})} \right] \left[d_{m',m''}^l(\theta_{rot}) e^{-im''\phi_{rot}} \right] \hat{A}_{m''}^l \right\} T_m^l. \quad (4.40)$$

Now we notice that the rotor spinning rate and the γ_{rot} powder angle are in the same exponent of the sum, in essence the γ_{rot} powder angles acts like a shift in in time. γ_{rot} is typically considered constant through the evolution. We can factor out an ω_r from the expression $\omega_r + \gamma_{rot}$ to get $\omega_r(t + \gamma_{rot}/\omega_r)$. In most circumstances $\omega_r = 2\pi/\tau_p$. We can pick γ_{rot} to be some multiple of our periodicity, say $\gamma_{rot} = c2\pi\tau_p/n$, where c is some integer index, and n is exactly the same n as discussed in the COMPUTE algorithm. We see the effect of performing a γ_{rot} powder average in the COMPUTE framework is simply reordering the sub-propagators U_k . Rather than recalculating the U_k for each different γ_{rot} angle, we simply reuse the ones we have previously calculated saving us from have to perform a direct method integration step for these angles.

To be a bit more explicit, we can write the propagator at k^{th} or $t = k/\omega_r/n$ division relating to the previous $k - 1$ propagator as

$$U(t_1, t_2, \gamma) = U\left(t_1 + \frac{\gamma}{\omega_r}, t_2 + \frac{\gamma}{\omega_r}, 0\right) = U\left(\frac{k-1}{n\omega_r}, \frac{k}{n\omega_r}, \frac{2\pi c}{n}\right) = U_{k,c} \quad (4.41)$$

Because of the relation of the time shift and the γ angle to previously calculated propagators,

we can remove the c dependence so that

$$U_{k,c} = U_{(c+k-1 \bmod n),(0)}. \quad (4.42)$$

Our total sub-period propagators, U^T , then become

$$U_{k,c}^T = U_{(c+k-1 \bmod n),(0)} (U_{(n-1),(0)})^m (U_{(p \bmod n),(0)})^\dagger : m = \text{int} \left(\frac{k+c}{n} \right) - \left(\frac{c}{n} \right). \quad (4.43)$$

We then use these propagators in the same analysis as in the COMPUTE method to get an improved algorithm which shortens the total simulation time if we need to include γ_{rot} angles.

This method was first elucidated by Hohwy *et. al.* in Ref. [60]. However, the γ_{rot} as a time shift was realized by both Charpentier[61] and Levitt[62]. An implementation of the γ -COMPUTE algorithm is given in Appendix A.2.4.

4.2.5 Non-periodic Hamiltonians

Non-periodic time dependant Hamiltonians are the hardest problems computationally. There is almost no other technique to perform the simulations other than the direct method. The direct method is very slow for problems in general, but it is the only one. Pulse shaping[63], slow molecular motion[64, 65, 66], and chemical exchange[67, 68] are the largest classes of non-periodic Hamiltonians. Molecular motion and chemical exchange are sometimes treated in the fast regime where the time dependence can be averaged away.

4.2.6 Powder Average Integration

There are two extremes in solid-state NMR. The first is a single crystal experiment where each molecule is aligned in the same direction. In this extreme, there is only one angle that describes a molecular frame to the rotor frame. In this picture the γ -COMPUTE

algorithm is invalid as there is only one γ angle. It also only requires us to compute the observed signal once, making it similar in speed to a liquid NMR simulation.

The second extreme is much more common in solid-state NMR, where we have many different crystallites oriented randomly throughout the sample. It is typically assumed that over the billions of crystallites, every possible orientation is present. To get our total signal then requires a sum over all these crystallites over the entire volume.

$$S(t) = \int S(t, \Omega_{rot}) d\Omega_{rot} \quad (4.44)$$

In most cases we do not know the analytical form of $S(t, \Omega_{rot})$, so we are forced to perform the integral as a discrete sum

$$S(t) = \sum_{i,j,k} S(t, \phi_i, \theta_j, \gamma_k). \quad (4.45)$$

Calculation of each $S(t, \phi_i, \theta_j, \gamma_k)$ can be time consuming itself, and to get the proper $S(t)$ we may have to perform many evaluations. Sampling the volume as best as possible using as fewest possible angles is desired to achieve computational efficiency. The γ -COMPUTE algorithm does aid us by attempting to eliminate the γ_k part of the sum, however, both θ_j and ϕ_i remain. Instead of sampling the entire sphere represented by θ and ϕ we can look at the angular dependences of the Hamiltonians. We can divide our spherical distribution into equal volume octants as shown in Figure 4.4. The valid range of $\theta = [0, \pi]$ and the valid range of $\phi = [0, 2\pi)$, and each octant has the range as shown in Figure 4.4. Using our high field expressed Hamiltonian form in Eq 4.40 where $m = 0$ (the only part that survives the truncation)

$$\mathbf{H} = T_0^2 \sum_{m'=-2}^2 \sum_{m''=-2}^2 \left[d_{0,m'}^2(\theta_{lab}) e^{-im'(\omega_r t + \gamma_{rot})} \right] \left[d_{m',m''}^2(\theta_{rot}) e^{-im''\phi_{rot}} \right] \hat{A}_{m''}^2 \quad (4.46)$$

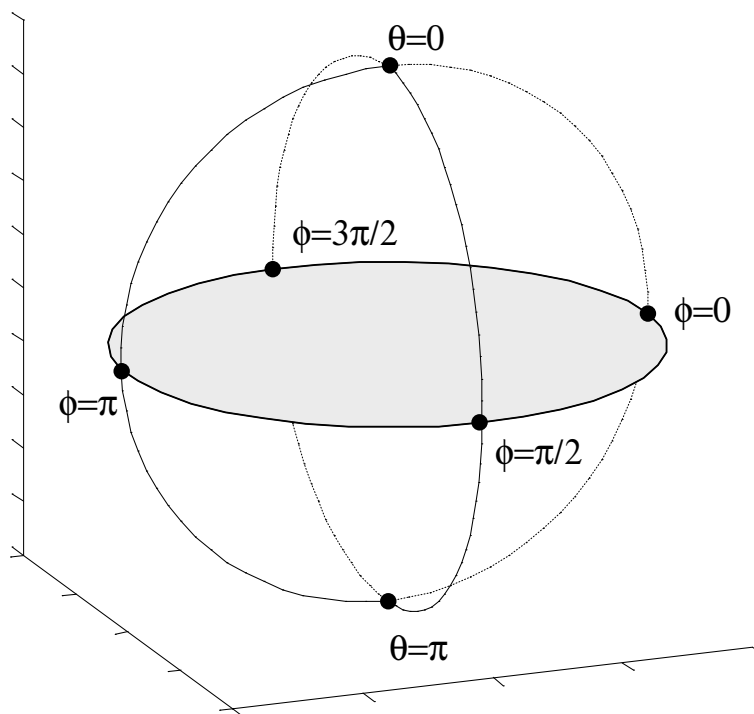


Figure 4.4: Octants of equal volume of a sphere.

Specific Hamiltonians can lead to a simplification of this form and different symmetry sets.

There are essentially 3 different symmetry groups that exist in NMR[69].

- **No Symmetry**—requires all 3 angles to be fully integrated over the entire range.

$$\begin{aligned}
 0 &\leq \theta \leq \pi \\
 0 &\leq \phi < 2\pi \\
 0 &\leq \gamma < 2\pi
 \end{aligned}
 \tag{4.47}$$

- C_i -Inversion symmetric therefore no γ dependence and inversion symmetric therefore requiring only half a sphere. Such cases exist when the eigenstates of the Hamiltonian do not change as the rotor rotates.

$$\begin{aligned}
 0 &\leq \theta \leq \frac{\pi}{2} \\
 0 &\leq \phi < 2\pi
 \end{aligned}
 \tag{4.48}$$

- D_{2h} -Cylindrically symmetric and inversion symmetric therefore no γ dependence.

Inversion symmetric requires only half a sphere, and cylindrically symmetric requires two octants (where $0 \leq \theta \leq \pi$). The two together implies one octant.

$$\begin{aligned} 0 \leq \theta &\leq \frac{\pi}{2} \\ 0 \leq \phi &\leq \frac{\pi}{2} \end{aligned} \tag{4.49}$$

This case is most prevalent when there is no η term in the Hamiltonians, and the molecule frame is the same for all atoms under static conditions[6].

For each three symmetry groups there are a variety of ways of generating points with in the required ranges[70, 71, 72, 73, 74, 75, 76, 77, 78]. Since we cannot sample an infinite number of angles, we wish to optimally choose the angles. From my own experience the best powder averaging schemes for no symmetry are the 3D-*ZCW* schemes[79, 80] using Gaussian quadrature[69]. For the C_i symmetry the 2D-*ZCW* schemes seem to work the best. Static NMR problems do not require that much time (by comparison to spinning simulations). The choice for a D_{2h} average should be handled best by the Lebedev schemes[81, 82].

Powder averaging is the easiest point at which to parallelize a NMR simulation. Each new processor should allow for linear scaling of the problem (i.e. n processors reduces the running time by n). To help create such parallel programs, an master/slave based MPI[83] implementation backbone is given in Appendix A.1.4.

4.3 Conclusions and Comments

Much of the numerical aspects of NMR are treated in the framework of the algorithms presented in this chapter using the specific forms of the equations of motion found

in chapter 3. The algorithms presented here are the sum total of the ones available to the NMR spectroscopist. Given each different algorithm, powder averaging type, rotational evolution, and Hamiltonians there are hundreds of different simulations one can perform. The basic reason to review all the available algorithms is to be able to choose the proper algorithm, interaction set, and other parameters that will perform the simulation at the greatest speed and efficiency. There is no ‘one’ algorithm that will provide the best answer every time, as a result constructing a master program that performs every possible simulation will not prove fruitful. Instead a package that includes the algorithms, data structures, and sub-routines needed to construct the correct program is more useful in general. The next chapter will treat the development and implementation of such a toolkit.

Chapter 5

BlochLib

5.1 Introduction

In the chapter 2, I laid down a foundation for a set of highly optimized data structures needed for NMR, basically the Complex number, the Vector and the Matrix. Everything else in computational NMR is based on these simple data types. Having very fast Vectors and Matrix operations will then determine the speed and functionality of the rest of the code that uses them. For this point on, I will take for granted the fact that we have these fast data structures. C++ and objects allow such creation of nice ‘block-boxes’ that performs specific tasks without any input from the user. This ability to create foundation and foundations upon foundations provide a simple way to construct our total simulation with ease. I will now present a tool kit specific to both classes of NMR, the quantum mechanical and classical.

5.2 The Abstract NMR Simulation

Numerical simulations in NMR (and most other physical systems) can be divided into 2 basic classes: Experimental Evolutions (EEs) and Theoretical Evolutions (TEs). The main object of both is some sort of generated data and both typically require some input of parameters.

5.2.1 Experimental Evolutions (EE)

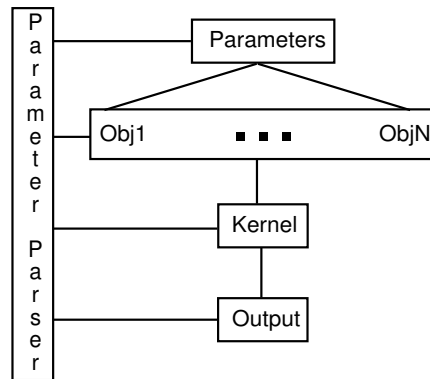
In Figure 5.1a, we find a pictorial representation of an EE. These types of simulations are some of the simplest to construct and generalize. A basic EE simulation/program is one designed to mimic some experimental condition. The basic experimental condition is an RF coil that applies RF pulses and provides a detection mechanism within chemical sample. The main function of an experiment is to apply different sets of pulse sequences to retrieve different sets of information.

Because of the wide variety of different pulse sequences, an EE must first act as a *Parameter Parser*. The Parameter Parser takes in some set of conditions and sets the various mathematical structures (the *Objects*) such that a *Kernel* can perform the proper calculation(s) which produces some sort of data we wish to see, thus *Output*.

5.2.2 Theoretical Evolutions (TE)

The other class of simulation, Theoretical Evolutions (TEs) (see Figure 5.1b), 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 designed to explore the physical properties of the system, even those not assessable

a) Experimental Evolutions



b) Theoretical Evolutions

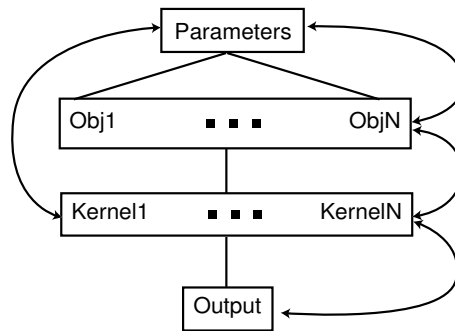


Figure 5.1: Many basic simulations can be divided into two main subgroups, a) Experimental Evolutions (EE), and b) Theoretical Evolutions (TE). EEs tend to use a solid Kernel driver, whereas the TEs can use many different Kernels, feedback upon itself; use the generated data in other kernel, and so forth. 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.

to 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 Figure 5.1). A good overview of the methods desired in NMR can be found in Ref. [84].

5.2.3 Existing NMR Tool Kits

Programs such as *Simpson*[85] have generalized the EE form of NMR simulation 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 EE publicly available.

Currently there is only one TE tool kit available to the NMR spectroscopists, *Gamma*[86]. 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.

5.2.4 Why Create a new Tool Kit?

Complex interactions like the demagnetizing field and radiation damping are becoming important and are best treated classically (see Ref. [87] 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[88, 89, 90] requiring detailed knowledge of magnetic fields in the sample. Low field experiments(see [91] and references there in) are also becoming more common. Pulse shaping[92] and multiple rotor angle liquid crystal experiments[93] are also becoming more frequent. *Gamma* and *Simpson* are ill-equipped to handle these new developments.

To treat all these newer developments (and to use the fast data structures described in chapter 2) I have created *BlochLib* to be the next generation NMR simulation tool kit. The tool kit is quite large and the documentation that describes all of its functionality is well over 1000 pages, I will try in thesis chapter to give a general overview of library itself. The following section will discuss some 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.

5.3 BlochLib 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 eas-

ily create both TEs and EEs for NMR in almost any circumstance. Below, several issues are addressed before the major design of *BlochLib* is discussed.

5.3.1 Existing Numerical Tool Kits

For the quantum mechanical aspects of NMR, the basic operation is matrix multiplication. The same expression template methodology can also be 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 than 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)[94] 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)[95, 83] 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 im-

plementations. One used fairly frequently for its power, speed, multiple types of algorithms is the CERN package MINUIT[96].

5.3.2 Experimental and Theoretical Evolutions for NMR simulations

As stated above TEs tend to require more possible configurations than 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). Figure 5.1 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 the two distinct classes of NMR simulation the quantum mechanical and the classical as described in Chapter 4.

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

5.3.3 *BlochLib* Layout

BlochLib is written entirely in C++. Figure 5.2 shows the basic layout of the 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 5.2). It is designed to be as modular as possible with each main section shown in Figure 5.2 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*. Figure 5.3 shows you some speed tests for the basic quantum mechanical NMR propagation operations. Each code sample (except for Matlab) was compiled using the GNU

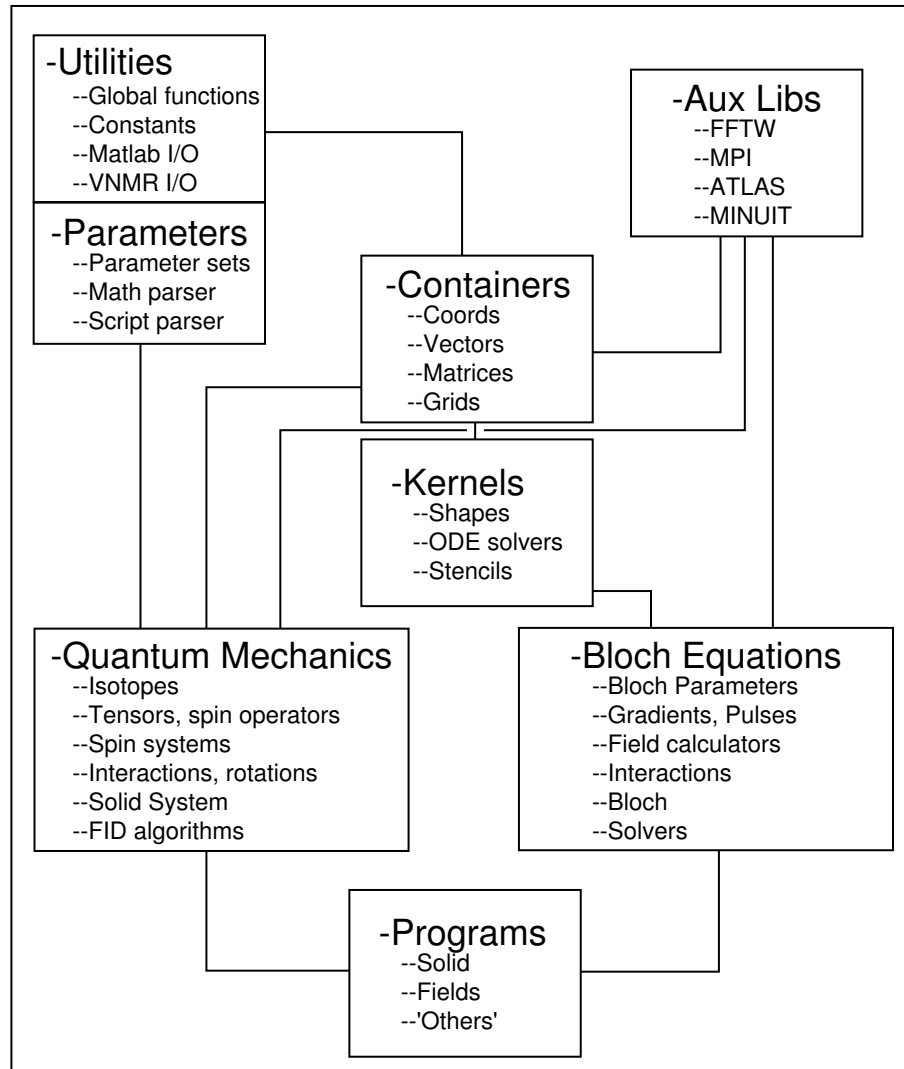


Figure 5.2: The basic design layout of the *BlochLib* NMR tool kit.

compiler (g++) using the same optimizations (-O3 -finline-functions -funroll-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) than 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 .

You may notice that *BlochLib*'s speed is slower than ATLAS's even though the same code is used. The reason for this discrepancy is discussed in section 5.3.4. *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

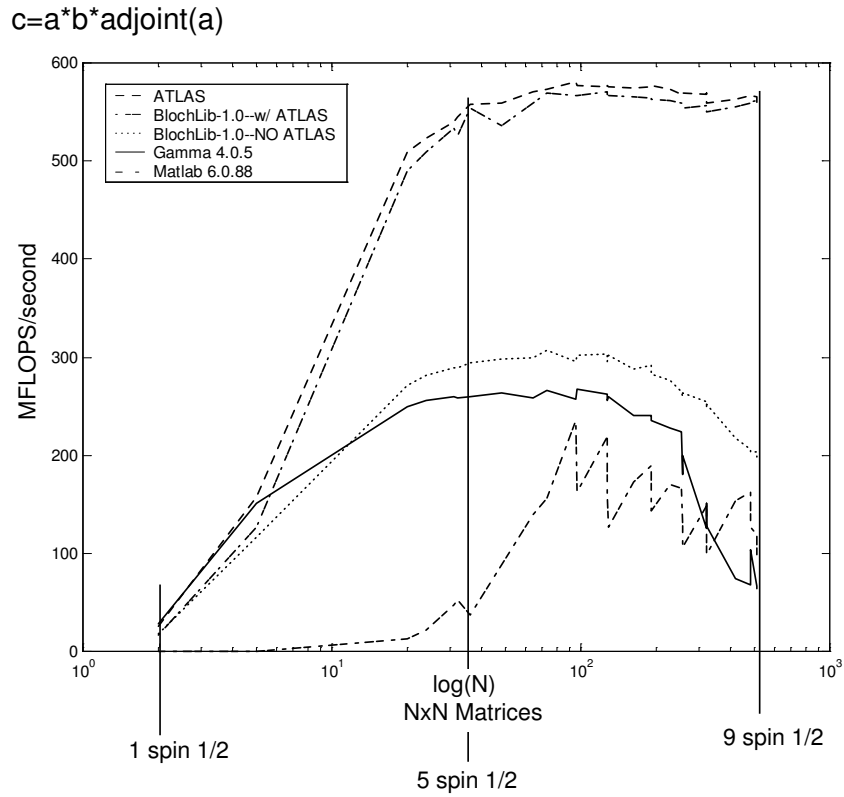


Figure 5.3: This figure shows a speed test for the common NMR propagation expression $c = a * b * a^\dagger$ in Millions of Floating point operations (MFLOPS) per second performed on a 700 MHz Pentium III Xeon processor running Linux (Redhat 7.2).

and coordinate transformations which only function on a 3-space. However, any length is allowed, but as Figure 2.4 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 super-diagonal, and the sub-diagonal 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-Schmidt ortho-normalization, matrix exponentials and matrix logarithms. 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 be 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. Several visualization techniques are best handled in the native format of NMR spectrometer software. A VNMR (Varian) reader and writer of 1D and 2D data is available as well as a XWinNMR (Bruker) and SpinSight (Chemagnetics) 1D and 2D readers are also included. 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 the points within a cylinder and a rectangle, using normal operators and `&&`) and or `(|)`, “`XYZcylinder && XYZrect`”).

The ODE solvers require function generation objects . Available ODE solvers are

listed in section 4.1.2. 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. `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 5th order method (the `ckrk` class) is a basic work horse medium accuracy. It is a good first attempt for attempting to solve ODEs[42, 45]. 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 [43, 41, 44, 45]. The semi-implicit Bulirsch-Stoer extrapolation method is based 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[97, 45]. 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 than two dimensional in *BlochLib* yet, the stencils over grid spaces are treated much differently than they would be over a standard three dimensional array. They are included in this version of *BlochLib* for completeness,

however, 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. 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 as described in section 3.3.

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 “ $T21_0,1 * 56$ ” ($T21_0,1$ is the second rank, $m=1$ spin tensor between spin 0 and spin 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 eigenvalue propagation for static samples and the γ -COMPUTE[60] 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, the demagnetizing field, and diffusion¹ as described in section 3.1.

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 create 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` 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

¹In the current version of *BlochLib*, diffusion is not treated.

Table 5.1: Available Matlab visualization functions in *BlochLib*

Matlab Function	Desicrption
<code>Solidplotter</code>	A GUI that plots many of the NMR file formats
<code>plotter2D</code>	A function that performs generic data plotting
<code>plotmag</code>	Visualization functions for the magnetic field calculators
<code>plottrag</code>	Magnetization trajectories classical evolutions visualizations

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 5.2). Also included are several Matlab visualization functions (see Table 5.1) 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`.

5.3.4 Drawbacks

As discussed before, 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 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 seconds (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 second of compilation time.

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 Figure 5.3 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 compilers cannot handle the memory

requirements for expression template unraveling (CodeWarrior (Metrowerks) crashes constantly because of memory problems from the expression templates). The saving 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.

5.4 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 5.2 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.

Table 5.2: Key examples and implementation programs inside *BlochLib*

Category	Folder	Description
Classical	bulksus	Bulk susceptibility interaction
	dipole	Dipole–dipole interaction over a cube
	echo	A gradient echo
	EPI	an EPI experiment[98]
	magfields	Magnetic field calculators
	rotating_field	Using field calculators and offset interactions
	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[87]	
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
	diffusion	1D diffusion example
	mpiplay	Basic MPI examples

5.4.1 Solid

The program *Solid* represents the basic EE quantum mechanical simulation program. *Solids* basic function is to simulate most 1D and 2D NMR experiments. It behaves much like *Simpson* but is faster for large spin sets as shown in Figure 5.4. *Solid* tends to be slower for small spin sets as explained in section 5.3.4. All simulations were performed on a 700 MHz Pentium III Xeon (Redhat 7.3), compiled with gcc 2.95.3 with donditions the same as those shown in Figure 5 of Ref. [85] for Figure 5.4a. Figure 5.4b shows the speed of the simulation of a C7 with simulations conditions the same as those shown in Figure 6e of Ref. [85]. In both cases the extra spins are protons with random CSAs that have no interactions between with the detected ^{13}C nuclei. *Solid* is essentially a parameter parser which then sends the obtained parameters to the main kernel for evaluation. The EE diagram (Figure

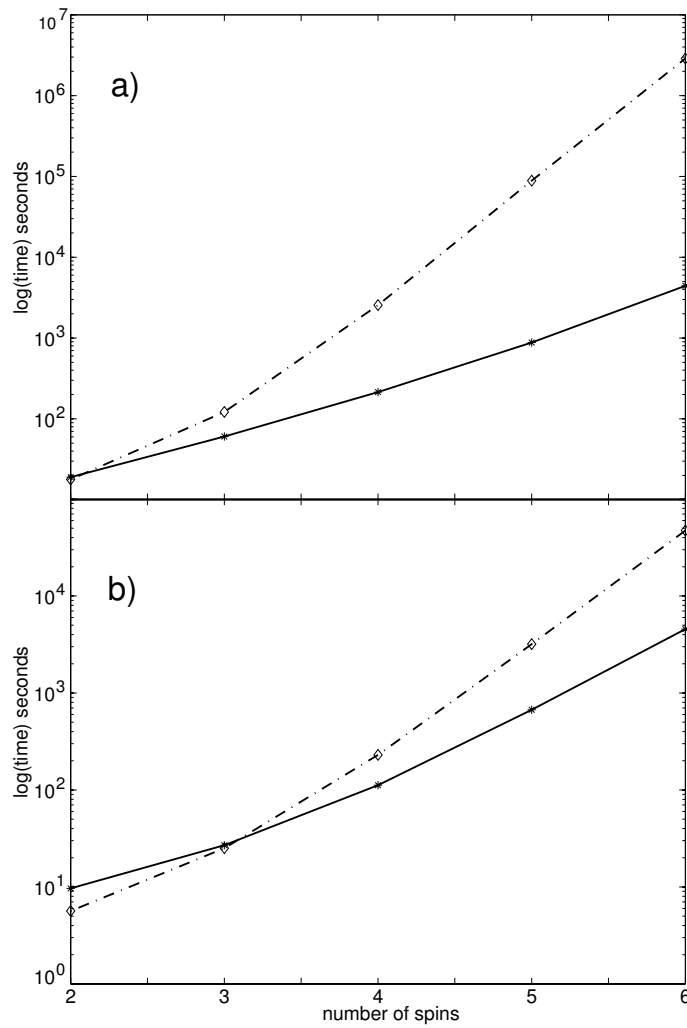


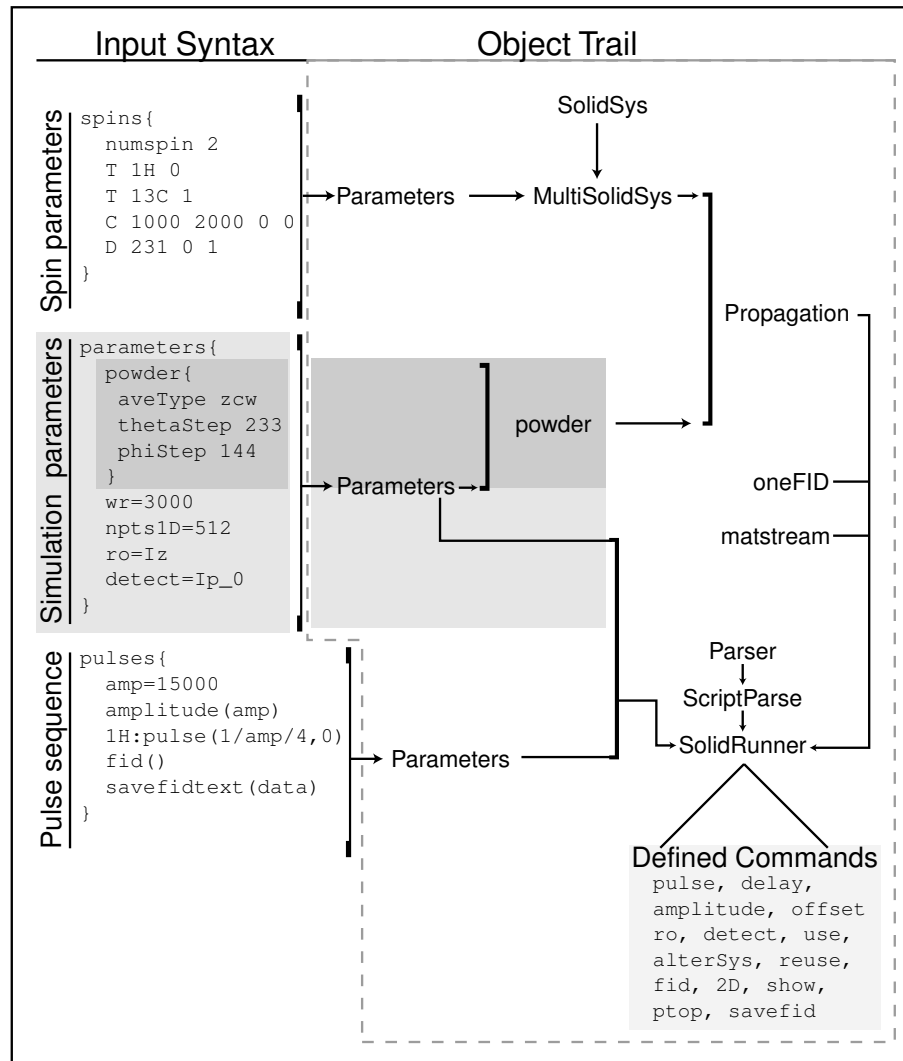
Figure 5.4: Time for simulations of *Solid* (solid line) and *Simpson* (dashed–dotted line) as a function of the number of spins. a) shows the simulation of a rotary resonance experiment on set pair of spins, and b) shows the speed of the simulation of a C7.

5.1) can be extended to more specific object usage used in *Solid* (Figure 5.5). 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.

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 Figure 5.6.

The results of a 2D and point-to-point simulation of the post-C7 sequence[99] are shown in Figure 5.7. Appendix A.3.1 shows the input configuration scripts for the generation of this data.

Figure 5.5: The design of the EE program *Solid* derived from the input syntax.

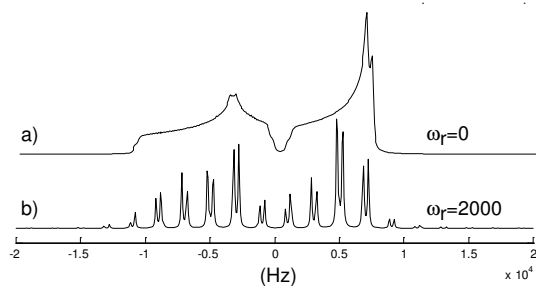


Figure 5.6: A two spin system as simulated by *Solid* where a) is with no spinning, and b) is with a rotor speed of 2000 Hz at the magic angle (54.7 deg). The spins system included 2 CSA's with the first spins parameters as $\omega_{iso} = 5000 * 2\pi$, $\omega_{ani} = 4200 * 2\pi$, and $\eta = 0$, the 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 Appendix A.3.1 for input configuration file.

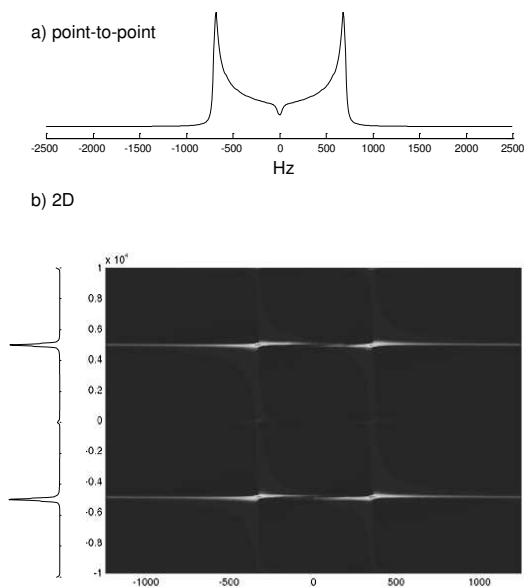


Figure 5.7: A two spin system as simulated by *Solid* of the post-C7 sequence where a) is collected using a point-to-point acquisition, and b) is a full 2D collection. The spins system includes a dipole coupling of 1500 Hz. For both a) and b) 233 powder average points were used. See Appendix A.3.1 for input configuration file.

5.4.2 Classical Program: Magnetic Field Calculators

Included in *BlochLib* is the ability to calculate magnetic fields over arbitrary coil geometries. The main field algorithm calculates a discrete integral of Ampere's equation for the magnetic field.

$$B(r) = \frac{\mu_o}{4\pi} \int \frac{I(r') \times dl(r')}{|r - r'|^2} \quad (5.1)$$

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, $dl(r')$, 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 2 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. Figure 5.8 shows the basic design of the field calculator using the `MultiBiot` object.

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

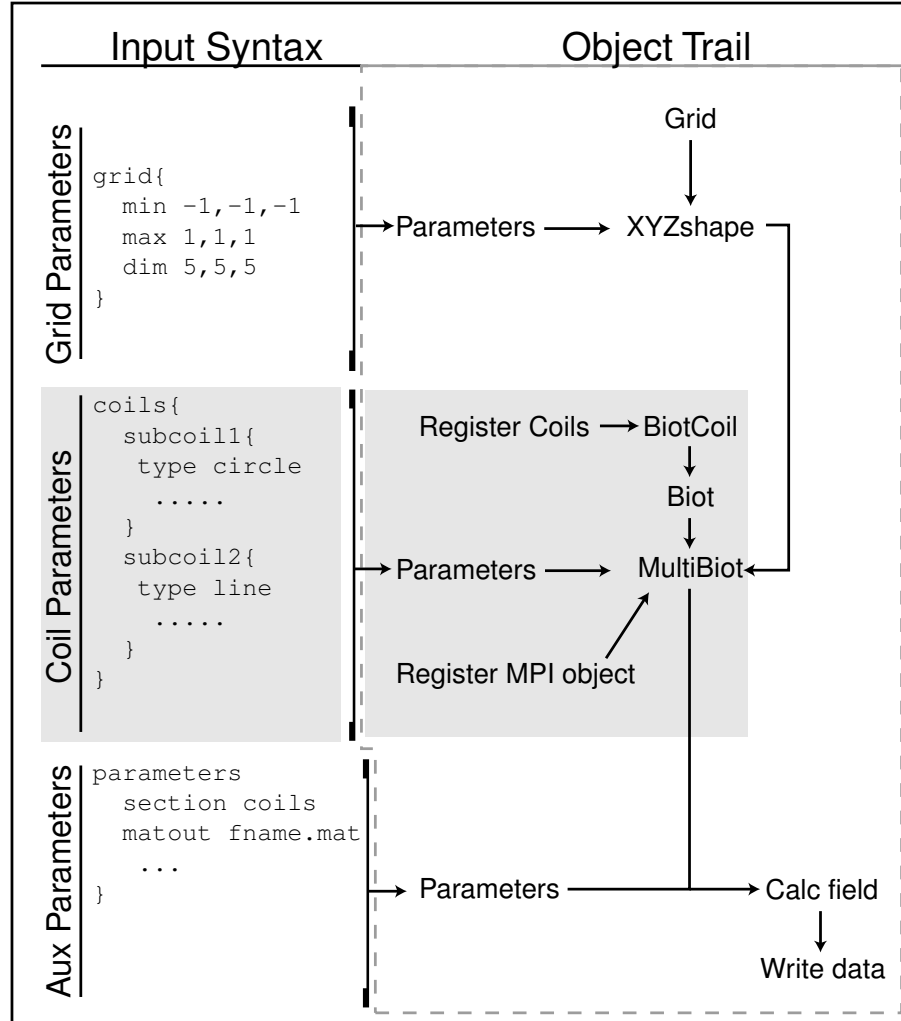


Figure 5.8: The basic design for the Field Calculator program.

XYZshape and MultiBiot objects. Parallelization can be implemented simply by defining the MPIworld object and passing it to the MultiBiot object. The data is written in 2 formats; into one readable by Matlab for visualization and into a file readable by the MultiBiot object.

This program is included in *BlochLib* under the `magfields` directory (see Table 5.2). Figure 5.9 shows the data generated by the program. The input file for this program can be seen in Appendix A.3.2. It should be noted that the convergence of the integral in

Eq. 5.1 is simply a function of the number of line segments you choose for the coils.

5.4.3 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 Figure 5.10. 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.

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 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.

Bulk Susceptibility

One such implementation attempts to simulate the result obtained in Ref. [100] Figure 2. This is a HETCOR (Heteronuclear Correlation) experiment between a proton and a phosphorous. The delay in the HETCOR sequence (see Figure 5.11a) allows the offset of the 1H to evolve. Next the 1H magnetization is placed back on the z-axis. The z-

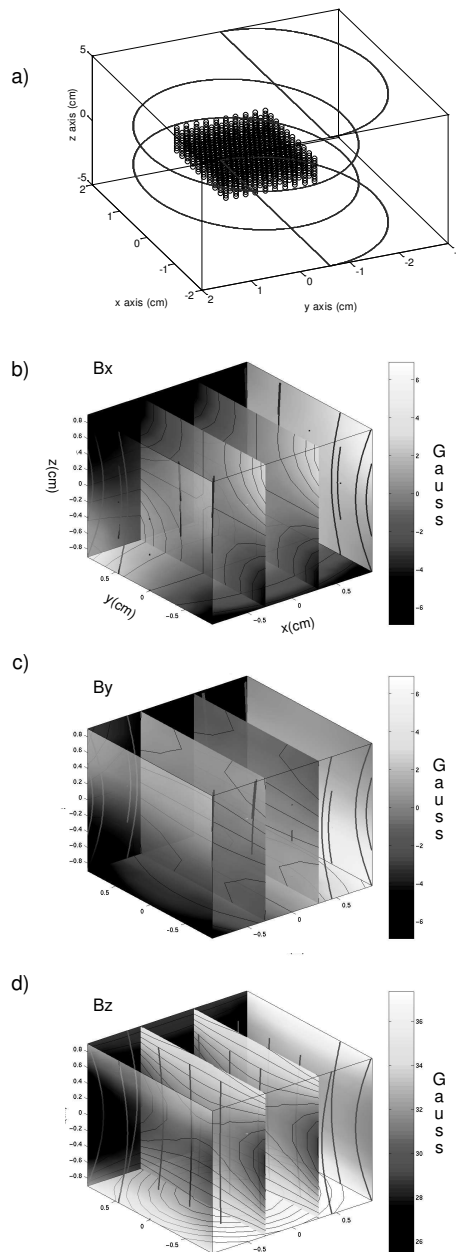


Figure 5.9: The magnetic field calculated by the program shown in Figure 5.8. The configuration file is shown in Appendix A.3.2 The Matlab function, `plotmag`, was used to generate these figures (see Table 5.1). The coil and the sampled grid are shown in a), the fields along the x, y, z directions are shown in b) –d) respectively.

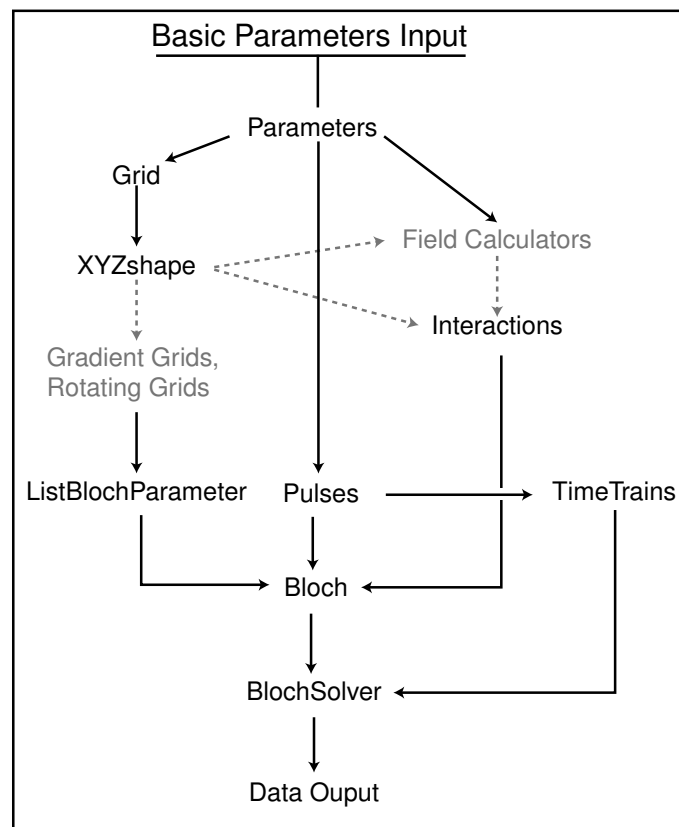


Figure 5.10: A rough design for a classical Bloch simulation over various interactions.

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 ^1H 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 Figure 5.11b and matches the result obtained in [100]. In order to correctly replicate the figure, the ^1H 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 ^1H also had to be altered to 0.002 seconds (the reference quotes 0.015 seconds, however the diagram shows a much faster decay then this time). The code for this diagram is in the `bulksus` folder of the distribution.

Radiation Damping

Another interesting implementation attempts to emulate the result obtained by Y.Y. Lin, *et.al*[87]. 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 (Figure 5.12b) matches Figure 2 in reference [87]. The result is a nonlinear partial resurrection of magnetization. The input parameters are those in the reference [87]. The data was plotted using `plottrag` in the distribution. The code for this diagram is in the `yylin` folder of the distribution and can be seen in Appendix A.3.5.

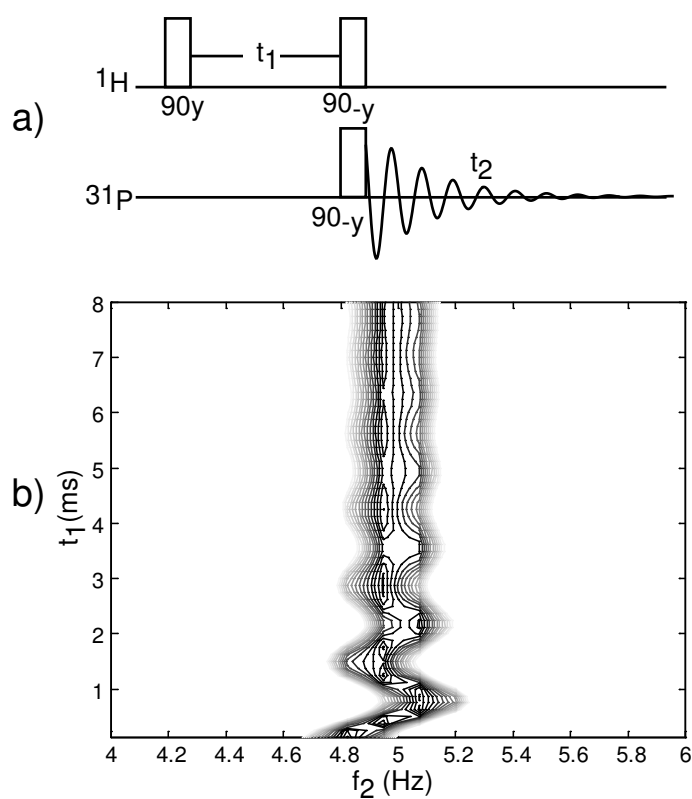


Figure 5.11: Simulated data from a HETCOR (Heteronuclear Correlation) experiment showing the effect of bulk susceptibility on the offset of the ^{31}P . a) shows the simulated pulse sequence and b) shows the simulated data.

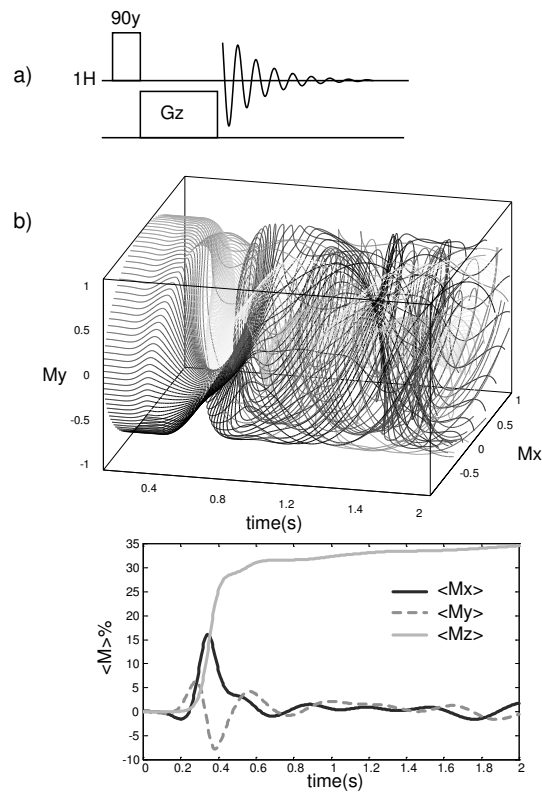


Figure 5.12: Simulated resurrection of magnetization after a crusher gradient pulse in the sequence shown in a). b) shows the effect of radiation damping and the modulated local field.

Probe Coils

The final example involves analyzing an NMR probe coil design. Dynamic Angle Spinning (DAS)[101] 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[102]. To optimize the split solenoid design one needs to see factors like inhomogeneities and effective power within the sample area. Figure 5.13 shows a split solenoid design as well the inhomogeneity profile along the xy-plane (the high field removes any need for concern about the z-axis).

Compared with a normal solenoid, Figure 5.14, the field profile is much more distorted, also given the same current in the two coils, the solenoid has 6 times more field in the region of interest than 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 contributor to the inhomogeneity across the sample. Correcting this by a U shape (or equivalent) should aid in correcting the profile.

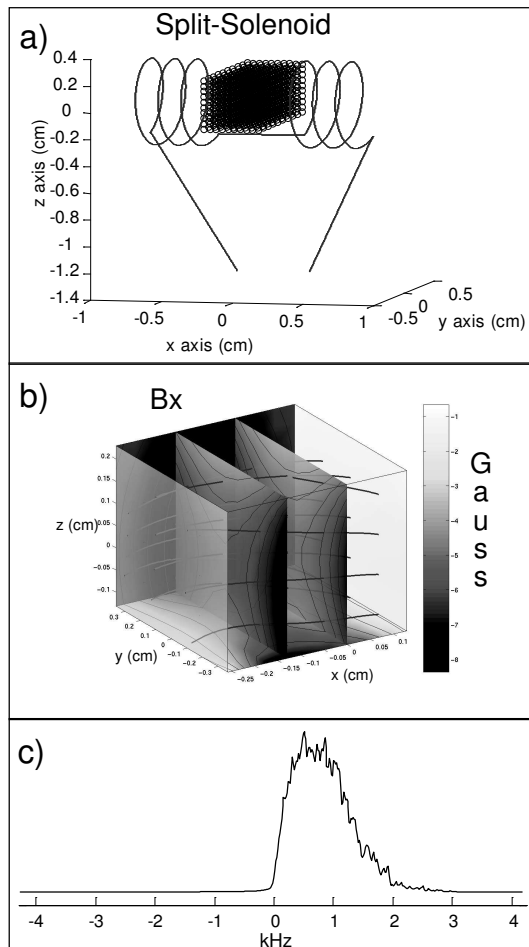


Figure 5.13: The magnetic field profile of a split solenoid (3 turns/cm with a radius of 0.3175 cm and a splitting of 0.6 cm), a practical coil to work around the Dynamic Angle Spinning problem of the solenoid coil (see text). a) shows the coil as well as the region of interest for the magnetic field (black points). b) shows the field profile along the x-direction given 3 amps of current. c) shows the effective inhomogeneity of such a coil for a proton. 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).

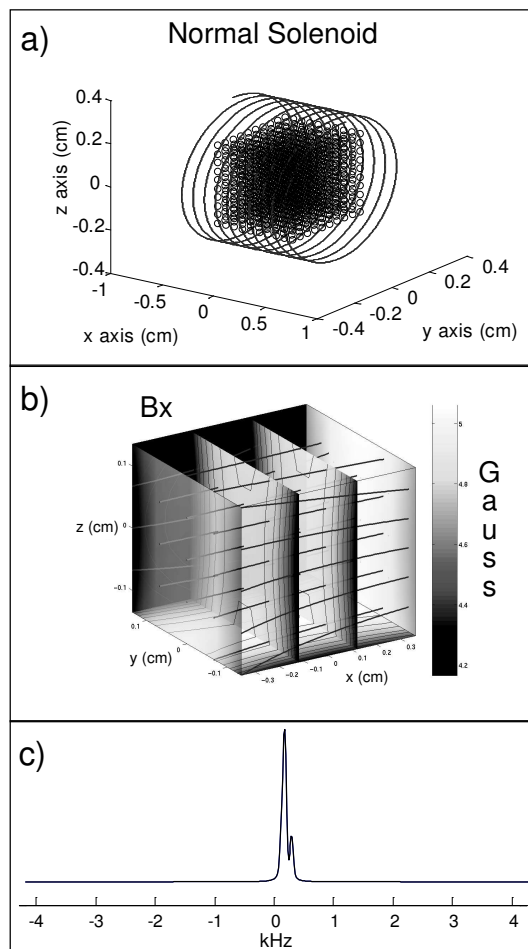


Figure 5.14: The magnetic field of a standard Solid State NMR probe detection coil (the solenoid, 10 turns/cm with a radius of 0.3175 cm). a) shows the coil as well as the region of interest for the magnetic field (black points). b) shows the field profile along the x direction given 0.5 amps of current. c) shows the effective inhomogeneity of such a coil for a proton. The small peak to the right of the main peak is the edges of the sampled rectangle close to the coil. The average field of the coil was subtracted from the result in c).

5.5 Conclusions

Throughout this chapter, 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: speed using expression-templates and ease of use using C++ and objects/operators. *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 than 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/>, or so I hope it remains after I leave. If it not there, I hope to maintain a copy and updates at <http://theadddones.com/>.

Chapter 6

Massive Permutations of Rotor Synchronized Pulse Sequences

6.1 Introduction

Given that we have a large set of fast computational tools, I will now go into an application beyond the typical scope of the ‘usual’ NMR simulation. The usual NMR simulation consists of a pulse sequence that either needs to be simulated to validate an experiment or add subtle corrections. Such simulations abound in the NMR literature to the point that even including a reference list for such types of simulation, I would probably have to reference 80% of the NMR publications around. In this chapter I wish to develop a general frame work to optimize any rotor synchronized pulse sequence over long time evolutions with an explicit example applied to the post-C7[99] sequence.

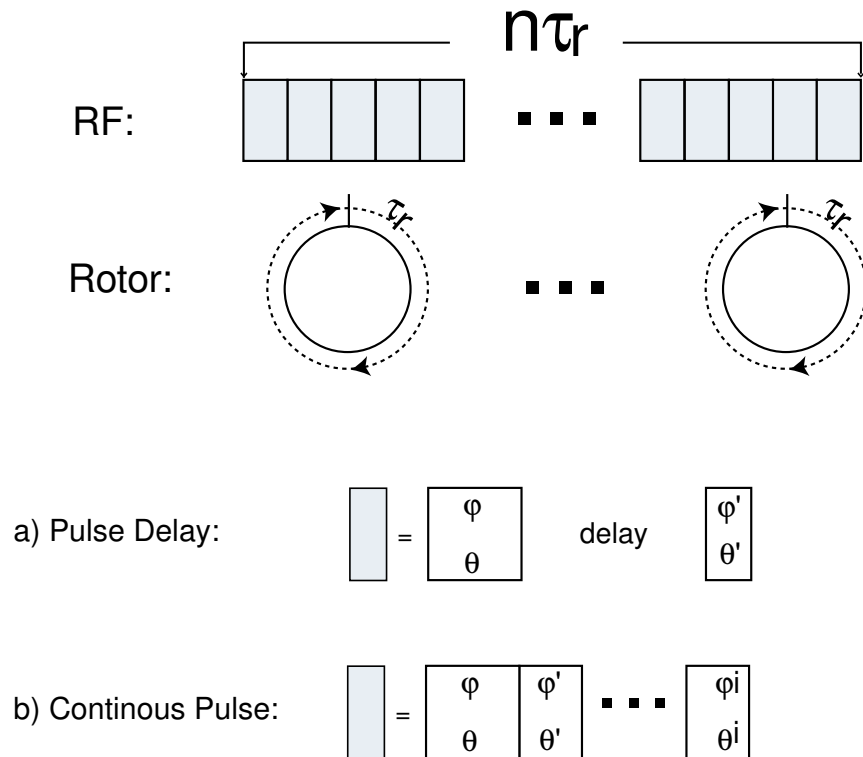


Figure 6.1: A general rotor synchronized pulse sequence a) using pulses and delays, and b) using a quasi continuous RF pulse.

6.1.1 Rotor Synchronization

Rotor synchronized sequences (RSS) are designed to create a specific average Hamiltonian when observed in multiples of the synchronized parameter, n . n is the amount of rotor periods the sequence takes to complete. Within the n cycle, there are typically two different types of sequences. The first contains a series of pulses and delays as in Figure 6.1a. The second applies continuous RF radiation during the rotor cycles as in Figure 6.1b. These RF sequences are designed to manipulate the spin space degree of freedom (the $T_{l,m}$ tensors) in conjunction with the spatial rotation the rotor performs (the $A_{l,m}$ tensors).

The RF pulses can be of arbitrary amplitude, phases and duration; however, the rotor can only be spun in a constant direction and is not adjustable in terms of duration

or phase during the course of an experiment. The speed of its rotation is manipulable, however, doing so during the course of an experiment is unstable and unrealizable due to the physical aspects of the rotation. For this reason, we typically say the rotor has a fixed spinning rate, ω_r and a corresponding period of $\tau_r = 2\pi/\omega_r$. The synchronized parameter then refers to n multiples of τ_r .

Almost all RSS sequences use continuous pulses. Each continuous pulse version has a pulse–delay–pulse version counterpart. The pulse–delay–pulse versions have better properties theoretically, but experimentally, they are extremely hard to implement. The limitation is the assumption of hard pulses, for many experimental parameters and real spin systems, a hard pulse is extremely hard to implement correctly.

6.2 Background Theory

6.2.1 Average Hamiltonian

Before we can really to describe specific RSSs, we need Average Hamiltonian Theory (AHT)[103] to attempt to understand the dynamics of such sequences. AHT uses the periodicity of the Hamiltonian to render a time dependant problem into an *approximate* time independent form. Given a periodic Hamiltonian of period τ_r , we can write the integrated Hamiltonian at time τ_r as series of time independent Hamiltonians

$$\int_0^{\tau_r} H(t) \partial t = \bar{H}^0 + \bar{H}^1 + \bar{H}^2 + \dots \quad (6.1)$$

where

$$\begin{aligned}
\bar{H}^0 &= \frac{1}{\tau_r} \int_0^{\tau_r} H(t) \partial t \\
\bar{H}^1 &= -\frac{i}{2\tau_r} \int_0^{\tau_r} \int_0^t [H(t), H(t')] \partial t \partial t' \\
\bar{H}^2 &= \frac{-1}{6\tau_r} \int_0^{\tau_r} \int_0^t \int_0^{t'} [H(t), [H(t'), H(t'')]] \\
&\quad + [H(t''), [H(t'), H(t)]] \partial t \partial t' \partial t''.
\end{aligned} \tag{6.2}$$

As an example, I will show the effect of the average Hamiltonian of a simple dipole under a rotor rotation of ω_r spinning at the rotor axis θ_r . The physical rotation only effects the spatial tensors. The Hamiltonian for a disordered sample in a high field reduces to

$$H = \left[\sum_m \exp[-im\omega_r t] d_{0,m}^2(\theta_r) \sum_{m'} D_{m,m'}^2(\Omega_{rot}) A_m^2 \right] T_0^2 \tag{6.3}$$

where $\Omega_{rot} = (\phi, \theta, \gamma)$ is the powder rotation. Given that there is no molecular orientation to worry about, in a disordered solid, we easily expand the sum to get

$$\begin{aligned}
H &= \frac{\delta_z}{16} \sqrt{\frac{3}{2}} (1 + 3 \cos(2\theta)) (1 + 3 \cos(2\theta_r)) T_0^2 + \\
&\quad \frac{3\delta_z}{2} \sqrt{\frac{3}{2}} e^{-i(\phi+\omega_r t)} \cos \theta \cos \theta_r \sin \theta \sin \theta_r T_0^2 + \\
H &= \frac{3\delta_z}{2} \sqrt{\frac{3}{2}} e^{i(-\phi+\omega_r t)} \cos \theta \cos \theta_r \sin \theta \sin \theta_r T_0^2 + \\
&\quad \frac{3\delta_z}{8} \sqrt{\frac{3}{2}} e^{-i(2\phi+2\omega_r t)} \sin^2 \theta \sin^2 \theta_r T_0^2 + \\
&\quad \frac{3\delta_z}{8} \sqrt{\frac{3}{2}} e^{i(2\phi+2\omega_r t)} \sin^2 \theta \sin^2 \theta_r T_0^2.
\end{aligned} \tag{6.4}$$

When integrated over a period τ_r , we get

$$\bar{H}^0 = \frac{\delta_z}{16} \sqrt{\frac{3}{2}} (1 + 3 \cos(2\theta)) (1 + 3 \cos(2\theta_r)) T_0^2. \tag{6.5}$$

The rest of the average Hamiltonian orders, \bar{H}^n , are all zero because this Hamiltonian commutes with itself at different times

$$\begin{aligned}
\bar{H}^1 &= -\frac{i}{2\tau_r} \int_0^{\tau_r} \int_0^t [A(t)T_0^2, A(t')T_0^2] \partial t \partial t' \\
&= -\frac{i}{2\tau_r} \int_0^{\tau_r} \int_0^t A(t)A(t')T_0^2T_0^2 - A(t)A(t')T_0^2T_0^2 \partial t \partial t' = 0.
\end{aligned} \tag{6.6}$$

Using Eq. 6.5 it is then easily to see we can pick θ_r such that $\bar{H}^0 = 0$. This angle, called the magic angle, is $\arccos(1/\sqrt{3}) = 54.7^\circ$. Magic angle spinning therefore removes any isolated second order spatial tensors. For dipoles and 1st order quadrupoles, this completely removes the interaction, for CSAs and J couplings it removes the anisotropic shift. By isolated we mean there there is *NO* other interaction in the main Hamiltonian that does not commute with the rest of the interactions. In real systems, we typically have CSAs and dipoles and J couplings to consider over multiple spins. Then our commutation rule breaks down and we must consider higher order AHT terms to get the proper answer. It also should be noted that the solution here is only valid for the time $t = \tau_r$, at any other time, the other components of Eq. 6.4 do not integrate to zero. The typical NMR experiment does not observe every $t = \tau_r$ but something less, the extra terms the introduce ‘side-bands’ in the resulting eigen-spectrum.

6.2.2 Recoupling RSS

RSSs have two main uses in NMR 1) to remove unwanted terms from the AHT (decoupling) [104, 105, 106, 107, 108, 109] and 2) to selectively reintroduce certain terms into an AHT (recoupling) [110, 111, 112, 113, 114, 115, 116, 99, 117, 118, 119, 120, 121, 122, 123, 124, 116]. Here we will focus on recoupling, but the methods introduced here are easily extendable to decoupling.

Most RSS recoupling methods have been categorized very nicely by their symmetries. The reader is encouraged to look to Carravetta’s [110], Eden’s [105], and Brinkmann’s [112] papers for more information, here we will simply give the results. RSSs are broken into two different main classes, the *R* and *C* classes. Figure 6.2 show the main difference between the two. Thus an entire RSS is made up from *C* or *R* subcomponents. The char-

acteristics of the C sub element is that they perform a total rotation of 2π . The R subunit rotates the spin states by π . To classify them even further Eden and Carravetta introduce 2 more symmetry elements along with n . N represents the time duration of a single sub element as $n\tau_r/N = \pi$ for the R class and $n\tau_r/N = 2\pi$ for the C class. The next factor ν along with N represent the phase iteration from one sub element to the next as shown in Figure 6.2.

Carravetta and Eden showed that given proper selection of N , n , and ν that one can select almost arbitrary elements to comprise the \bar{H}^0 element of the AHT given a certain set of interactions (dipoles, CSAs, etc). To correct for higher order AHT terms one can apply further symmetry if we know which tensor elements we desire.

Given a real experimental system, the basic $C7$ as shown in Figure 6.2 will fail to produce desired results because of higher order commutators. Another aspect of a real experimental system is that RF pulses are not perfect and resonant on only one frequency. To make a C or R cycle robust towards inhomogeneity and offset of the RF pulse itself, each C or R cycle must be internally compensated, in effect reversing the damage done by a bad RF pulse. For the C cycles, this is easily performed by applying the same amplitude, but reversing the RF phase by π , which reverses any problems, to first order, caused by offsets and inhomogeneity problems. For the R cycles, we apply a second R with the phase equal to $-\phi$ from the last (as we are treating a π pulse). We have assumed that over the course of a single C/R cycle that the real system Hamiltonian can be considered constant, which is why this only works to first order. Each C/R cycle gets an extra C/R attached as shown in Figure 6.3a and b. This technique is called *compensation*. For the C cycles, we have a total rotation of 4π , which can be divided into $(\pi/2)_\phi - (2\pi)_{\phi+\pi} - (3\pi/2)_\phi$ which

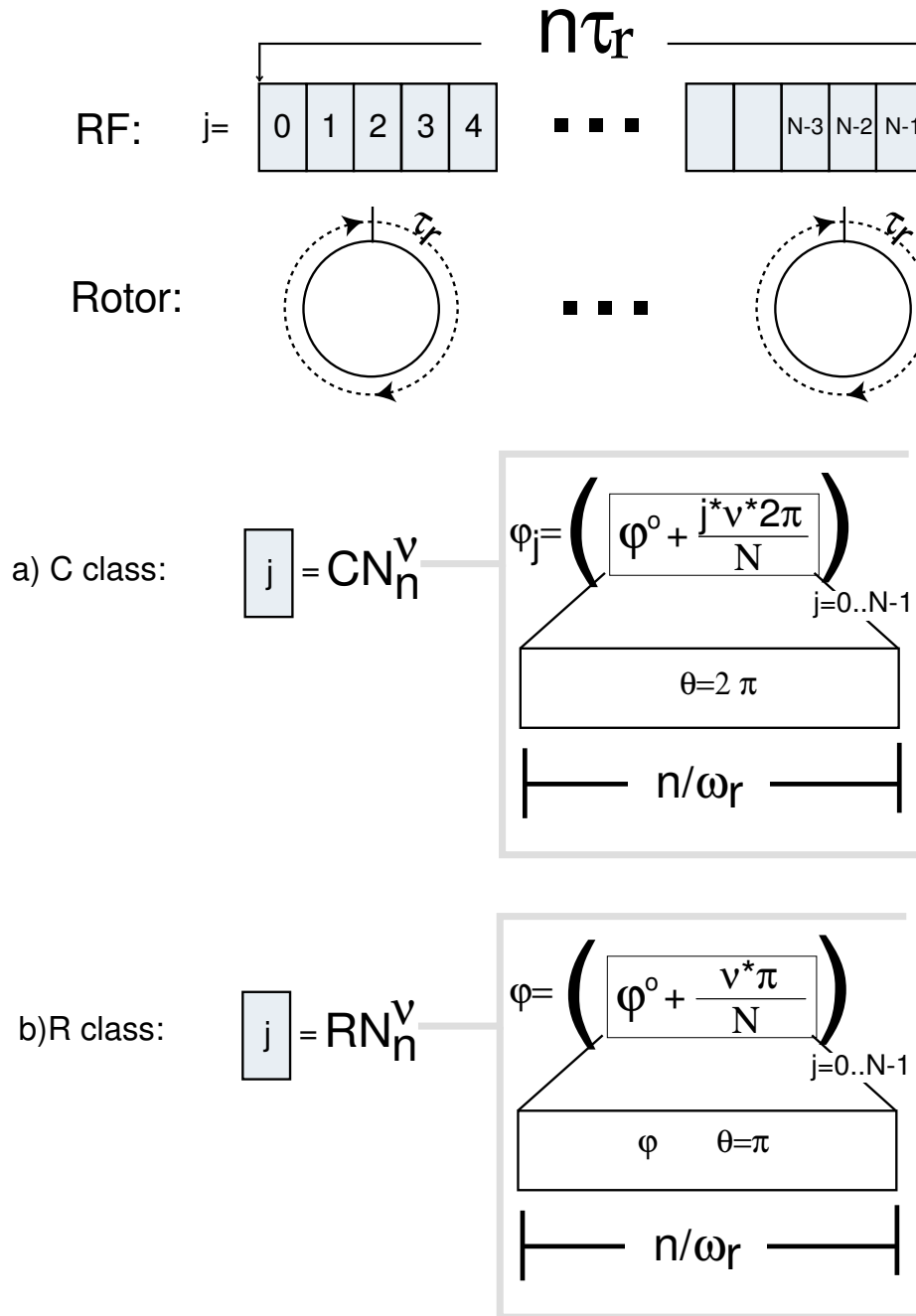


Figure 6.2: The two RSS classes C (a) and R (b).

has better offset and inhomogeneity properties as demonstrated in Ref. [99]. This simple reordering of the 4π is called ‘posting’ hence the name *post-C7*. The R sequences have now a *total* rotation of 2π which can be split arbitrarily into many forms like a $(\theta_1)_\phi - (2\pi - \theta_1)_{\phi+\pi} - (\theta_1)_{-\phi} - (2\pi - \theta_1)_{-\phi-\pi}$ sequence. Figure 6.3c and d show this posting explicitly.

The RSS sequences produce desired tensor elements from a system Hamiltonian of the form

$$\bar{H}^0 = \sum_l \sum_{m,m'} g_{m,m'} A_{l,m} T_{l,m'} + c.c. \quad (6.7)$$

where *c.c.* means the complex conjugate and $g_{m,m'}$ is a scaling factor. The scaling factors is also an important aspect of these sequences as they determine the apparent spectral distribution, and are needed when using these RSS in larger pulse sequences designs[113]. Calculation of the scaling factors for the R or a C type is a simple application of AHT.

A general rotation through the Euler angles (ψ, θ, ϕ) , on an arbitrary tensor can be represented using the notation for tensor rotation

$$\overline{F_{l,m'}} = \sum_{m=-l}^l e^{-im'\psi} d_{m',m}^l(\theta) e^{-im\phi} F_{l,m}, \quad (6.8)$$

where $\overline{F_{l,m'}}$ is the rotated tensor and $F_{l,m}$ is the original tensor. A given dipolar or CSA interaction contains two unique tensor elements, the spatial part, $A_{2,0}$, and a spin part, $T_{2,0}$. The RSS sequences rotate the spatial part by $(0, \theta_r, \omega_r t)$ where θ_r the angle of the rotor, and ω_r is the spinning speed ($\omega_r t$ then represents the total phase). They also rotate the spin part through the Euler angles $(0, \omega_{rf} t, \phi)$ where ω_{rf} is the pulse amplitude (thus $\omega_{rf} t$ represents the total rotation angle) and ϕ_j is the pulse phase. Our dipolar Hamiltonian under such a rotation becomes

$$A_{2,0} T_{2,0} \rightarrow \sum_{m=-2}^2 d_{m,0}^2(\theta_r) e^{-im2\pi\omega_r t} A_{2,m} \sum_{m=-2}^2 d_{m,0}^2(2\pi\omega_{rf} t) e^{-im\phi_j} T_{2,m} \quad (6.9)$$

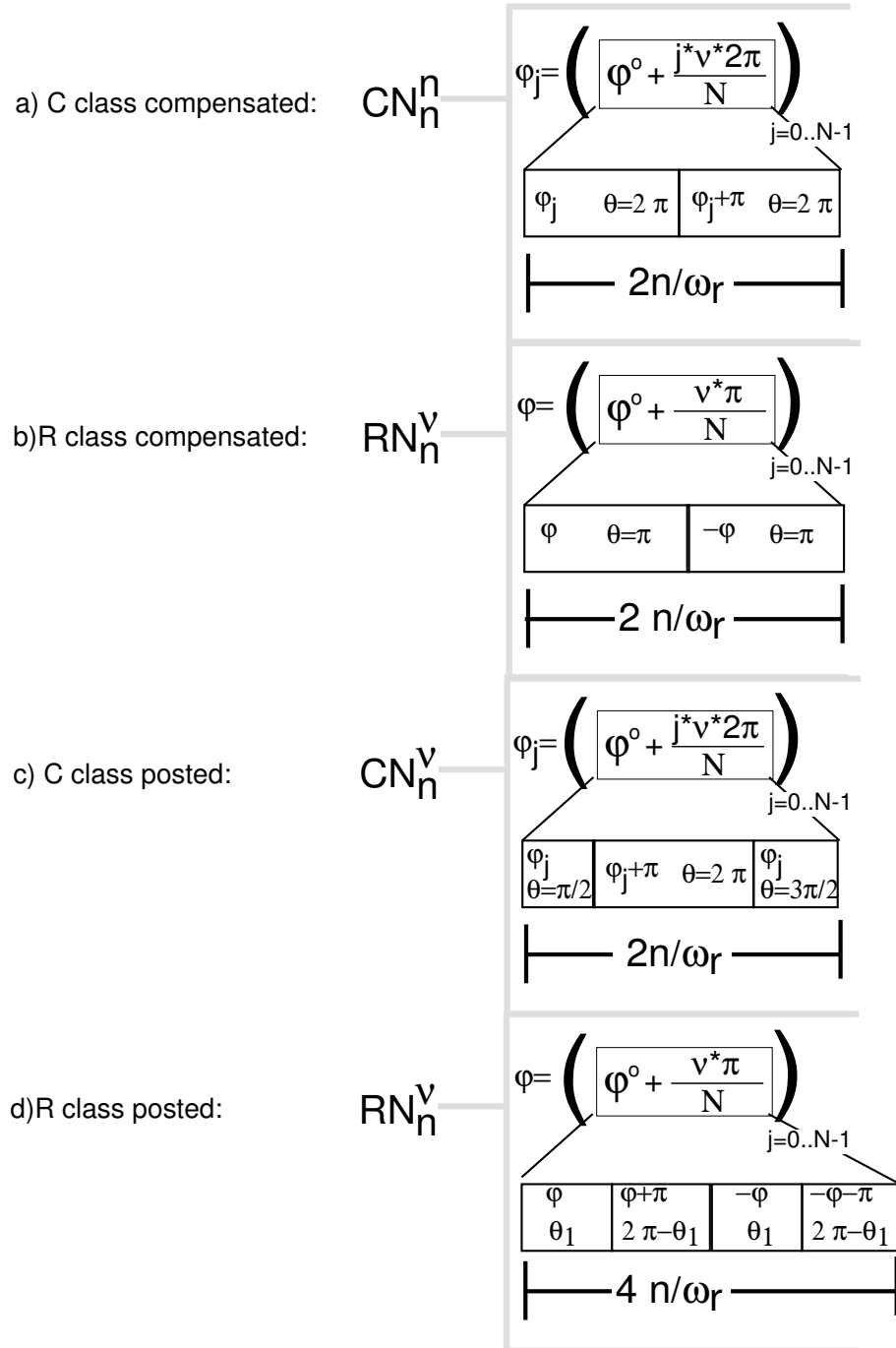


Figure 6.3: Compensated C (a), R (b) and posted C (c), R (d) RSS sequences.

Assuming that the symmetry pulse phases, ϕ_j , selects only terms with $l = 2, m, m'$ then our rotated Hamiltonian becomes

$$A_{2,0}T_{2,0} \rightarrow (d_{2,0}^2(\theta_r)e^{-i4\pi\omega_r t}A_{2,m})(d_{2,0}^2(2\pi\omega_r f t)e^{-i2\phi}T_{2,m'}) + c.c. \quad (6.10)$$

In order to calculate what the scaling factor in front of the $A_{2,\pm m}T_{2,\pm m'}$ terms we need to make some assumptions about the applied sequence. First, the entire sequence is applied over a multiple of a rotor cycle such that the sequence is periodic and AHT can be used to continue the calculation. Second, the rotor cycle is divided into N subsections such that during the j^{th} subsection only the pulse phase, ϕ_j is varied and all subsections are the same length in time, $\tau = \frac{n}{\omega_r N}$. The zeroth order average Hamiltonian is then

$$\bar{H}^0 = \sum_{j=0}^{N-1} \frac{1}{\tau} \int_{j\tau}^{(j+1)\tau} d_{m,0}^2(\theta_r)e^{-im\pi\omega_r t} d_{m',0}^2(2\pi\omega_r f t)e^{-im'\phi_j} dt A_{2,m}T_{2,m'} + c.c. \quad (6.11)$$

The scaling factor, g , is given as the total constant that multiplies $A_{2,m}T_{2,m'}$ in the above equation.

$$g_{m,m'} = \sum_{j=0}^N \frac{1}{\tau} \int_{j\tau}^{(j+1)\tau} d_{m,0}^2(\theta_r)e^{-im\pi\omega_r t} d_{m',0}^2(2\pi\omega_r f t)e^{-im'\phi_j} dt \quad (6.12)$$

6.2.3 C7

From this point I will discuss the $C7_2^1$, the $C7$ [115] as it was originally named before Eden's paper. The post version was also tagged before Eden's paper in Ref. [99]. This sequence produces a double quantum (DQ) AHT to zeroth order using the dipolar Hamiltonian

$$\bar{H}^0 = g_{-1,2}A_{2,-1}T_{2,2} + g_{1,-2}^*A_{2,1}T_{2,-2}. \quad (6.13)$$

This Hamiltonian can then be used to determine approximate distance information between atoms as it selectively evolves only terms from the dipolar Hamiltonian, creating

DQ coherences between them. The accuracy of the distances is related directly to how well the $C7$ performs under all the various experimental and molecular considerations.

A good measure of the $C7$ ability to create DQ coherences, is to measure the transfer between two spins. The transfer can be measured directly starting from an initial polarization on spin 1 ($\rho_o = I_z^1$), application of the sequence, then a measurement of the polarization on the second spin ($I_{det} = I_z^2$). Application of the sequence n times where $n \gg 1$ results in a steady state transfer of the coherence. Figure 6.4 shows the transfer for different dipole couplings between the two spins. Introduction of CSA terms reduce the effect of the transfer as also shown in Figure 6.4. The sequence used in the Figures is shown in Figure 6.5a.

The dipole coupling determines the rate of transfer: a large rate means a closer distance. One can easily see from Figure 6.4 that introduction of large CSA terms cause the steady state transfer to fail. In multi-spin systems, this then leads to confusion of the distance information. The effect is most pronounced at small dipole couplings where we would like to achieve the best data for longer range distance determination. It would be beneficial to remove as many higher order average Hamiltonians as possible in these RSS type sequences.

6.2.4 Removable of Higher Order Terms

Usage of the RSS sequences is restricted to multiples of the rotor period. So long as we create the zeroth order average Hamiltonian at the final observation point, the number of rotor periods is arbitrary. The higher order terms of the $C7$ sequence introduce a variety of other unwanted tensor components due to the commutators between the CSA and dipolar Hamiltonians during each cycle. For a simple two spin system, the only other terms that

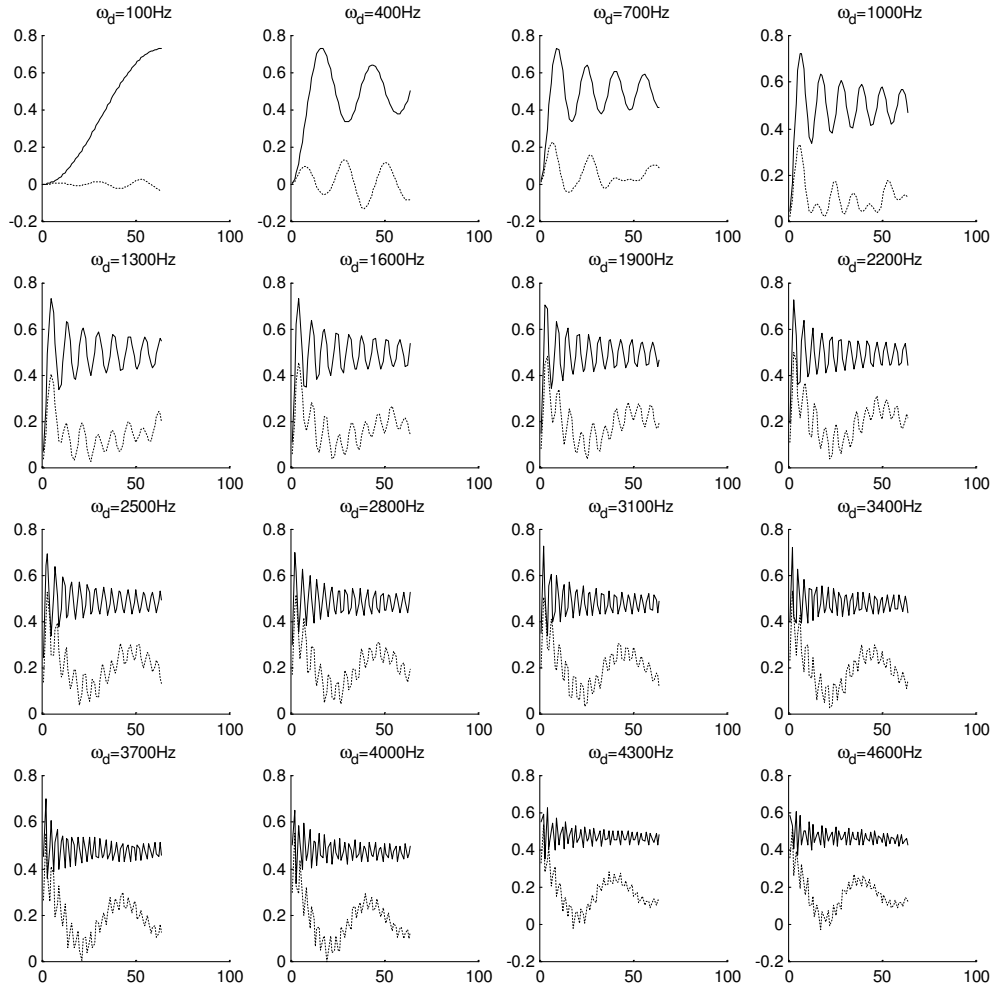


Figure 6.4: Post- $C7$ transfer efficiencies on a two spin system with $\omega_r = 5k\text{Hz}$ for various dipolar coupling frequencies (ω_d). The dashed lines indicate a CSA on spin one of ($\delta_{iso} = 12000\text{Hz}$, $\delta_{ani} = 8700\text{Hz}$, $\eta = 0$) and on spin two of ($\delta_{iso} = -5400\text{Hz}$, $\delta_{ani} = 12300\text{Hz}$, $\eta = 0$).

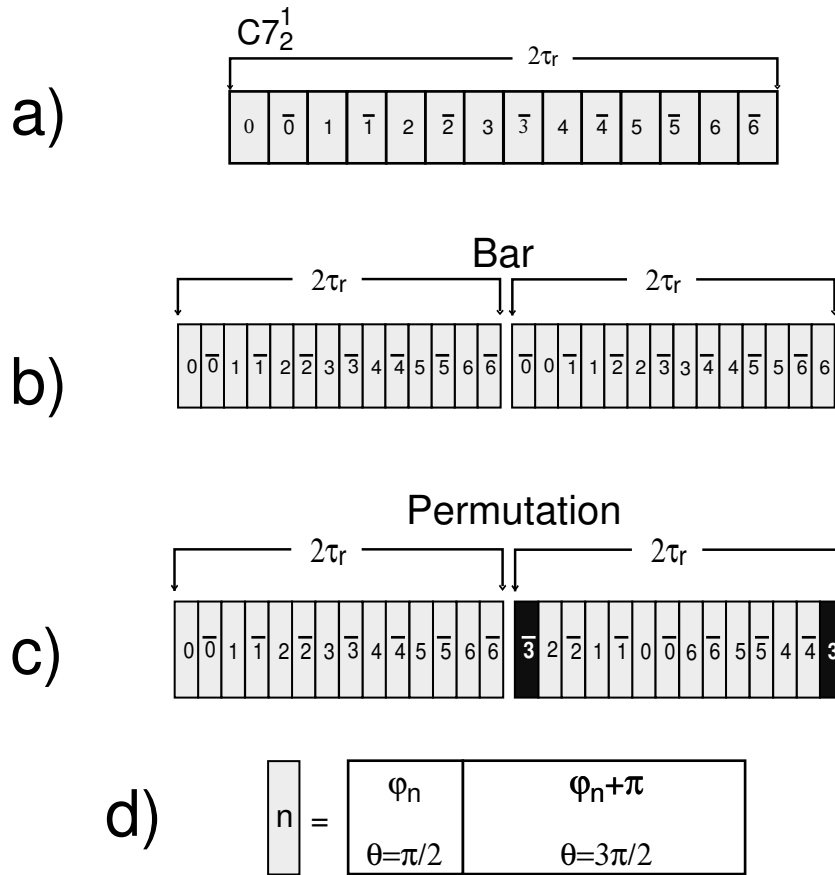


Figure 6.5: Different base permutations on the post- $C7$ sequence: a) the basic sequence, b) the barred sequence, c) permuted sequence, and d) the composition of a sub element.

can appear are combinations of $T_{2,0}$, $T_{2,\pm 1}$ and $T_{2,\pm 2}$. Because we only wish to observe the I_z terms in the evolved density matrix any solo $T_{2,0}$ terms can be ignored as $[T_{2,0}, I_z] = 0$. Thus the only tensor component we wish to remove are the $T_{2,\pm 1}$ terms. These terms have the property that they are odd under a parity transformation where as the $T_{2,\pm 2}$ are not. In the first application of a RSS sequence generates any $T_{2,\pm 1}$ terms, the same sequence with a global π phase shift (called *barring*) from the last will ‘reverse’ any evolution from the $T_{2,\pm 1}$ while maintaining the evolution under $T_{2,\pm 2}$ terms. We can then easily alter the original $C7$ sequence to include two $C7$ sequences with one barred relative to the last as shown in Figure 6.5b.

Even in the 2-cycle $C7$ sequence errors can still accumulate from even higher order terms, thus we can even bar the next 2 $C7$ relative to the last 2 resulting in a sequence like $C7 - \bar{C}7 - \bar{\bar{C}}7 - C7$. We can continue this process, which is called *super-cycling*, until the signal has decayed beyond observation. This super-cycling process was initially used most dramatically in liquid state broadband decoupling[125, 126, 127, 128, 129, 130, 131, 132] but is becoming more prevalent in solid-state NMR as the hardware and theory improve[112, 113].

Problems with super-cycling

The removal of higher order terms comes typically with a penalty. The sequence is usually altered to include super cycles which can make the sequence very long. Most solid-state samples have a relatively fast decay rate (T_2 is on the order of mille-seconds where as in liquids it can be seconds) meaning that super cycles cannot get very long. Perhaps not long enough to remove all the terms necessary for efficient use of the sequence. Not only do we have a time constraint, but even super-cycling for anisotropic samples can lead to

diminishing returns due to hardware imperfections[133, 134].

This leads us to ask whether or not we can improve on the basic RSS in a framework that allows us to measure the effectiveness of a particular super-cycle. We still would like to use the basic symmetry principles that do provide useful removal of terms, but use them in a time constrained way.

6.3 Permutations

The problem of determining the best set of RSS sub-cycles to use for a super-cycle was one best handled using the symmetries of the underlying Hamiltonians. This technique works very well for liquid samples where the Hamiltonians are isotropic and have nice symmetry properties. Techniques like MLEV[131, 130], WALTZ [127, 126], SPARC[125], and GARP[128] use the symmetries to decouple liquid state spins in a highly optimal way. The anisotropic and spinning environment of a solid-state experiment makes such application of super-cycles hard and less effective. We now wish to see if there is a way for us to determine the best set of cycles where the symmetry principles seem to fail. In order to investigate the effect of different cycles, we will use the symmetry principles as our starting point, but from there, the problem is open ended. We will use simple permutation to determine the best sequence.

6.3.1 The Sub-Units

Any particular defined ordering of RSS sequences is a subunit. We will use a particular naming scheme: the basic RSS sequence is labeled ‘o’ (lowercase ‘o’) as in Figure 6.5a, a barred RSS sequence is labeled ‘O’ (capital ‘O’) as in Figure 6.5b, an internal

Table 6.1: A list of some sub-units for a $C7$ permutation cycle.

sub-units
o
O
w
W
oO
oOOo
wW
wWWw
oOWw
OooO
WwwW

permutation (a reordering internal to a single RSS cycle) of a RSS sequence is given the label ‘w’ (lowercase ‘w’) as in Figure 6.5c, the bared version of the internal permutation is given the label ‘W’ (capital ‘W’). A sub unit can be constructed from other subunits as well so long as the subunit return the Average Hamiltonian to the desired result. Table 6.1 lists a few of the sub units for a $C7$.

6.3.2 The Measure

To determine the best sequence, we need some sort of measure that typifies the RSS sequence. For a $C7$ sequence we desire $T_{2,\pm 2}$ terms above all other elements. So the measure of a good sequence will have large $T_{2,\pm 2}$ terms while minimizing any other terms. Since we are performing exact numerical simulations of the propagator of the various $C7$ cycles, we need to use the effective Hamiltonian

$$H_{eff} = \frac{-i}{\tau} \log(U) = \sum_l \alpha_{l,m} T_{l,m} + \sum_{l,l'} \beta_{l,m} [T_{l,m}, T_{l',m'}] + \dots \quad (6.14)$$

where τ is the total time of the super-cycle, α and β are complex constants. Because of the commutation relations of $T_{l,m}$ the higher order terms, when expanded, will reduce to terms

proportional to a single tensor, thus numerically the sum is reduced to a sum of only the $\alpha_{l,m}$ components. $\alpha_{l,m}$ is easily extracted by the trace operation

$$|\alpha_{l,m}| = \left| \int \frac{\text{Tr} [H_{eff}(\Omega) (T_{l,m})^\dagger]}{\text{Tr} [T_{l,m} (T_{l,m})^\dagger]} d\Omega \right|. \quad (6.15)$$

In Eq. 6.15 the integral over all the power angles as the effective Hamiltonian is Ω dependent. We are also only interested in the magnitude as any sign changes are easily phased away. For the $C7$ we can now define two measures. Given n different sequences, the ratio of the magnitudes of the i^{th} sequences $\alpha_{2,\pm 2}^i$ to the original ‘non-permuted’ sequence $\alpha_{2,\pm 2}^0$ should be large for good signal-to-noise in an experimental setting, M_{mag}^i . The second, and better theoretical measure is the ratio of the $\alpha_{2,\pm 2}^i$ terms to the rest the undesired tensor terms, the maximum being the best sequence, M_R^i .

$$\begin{aligned} M_{mag}^i &= \frac{|\alpha_{2,\pm 2}^i|}{\alpha_{2,\pm 2}^0} \\ M_R^i &= \frac{|\alpha_{2,\pm 2}^i|}{\sum_{l \neq 2, m \neq \pm 2} |\alpha_{l,m}^i|} \\ M_{mag} &= \max(M_{mag}^i) \\ M_R &= \max(M_R^i) \end{aligned} \quad (6.16)$$

The goal for any given master cycle is to maximize both of these measures, using M_R as the master measure. Some of the $\alpha_{l,m}$ terms are not relevant to the M_R measure. For instance, because the $T_{2,0}$ terms do not effect the evolution, they should not be counted in the sum. Also other generated tensors are more harmful to the evolution then others. For the $C7$, extra I_z terms are worse then $I_{x,y}$ terms, so the M_R should weight the z terms more. A revised version of the M_R^i is given as

$$M_R^i = \frac{|\alpha_{2,\pm 2}^i|}{\sum_{l \neq 2, m \neq \pm 2} b_{l,m} |\alpha_{l,m}^i|} \quad (6.17)$$

where $b_{l,m}$ are the relevant weighting factors.

6.3.3 Algorithmic Flow

There are many algorithmic sub components to optimally generate and measure each master cycle. Because we have a time constrained problem, there will be a maximum number of sub-units that we can apply, P . An arbitrary permutation can be constructed from any subset of the available sub-units. The number of sub-units in a given subset is called N . The task is to generate all the valid permutations of length P from subset of length N . P is not necessarily a factor of N (i.e. $P/N \neq 0$), we then must select the largest subset, N' , of the N sub-units that are factors of P . Then to generate all the valid permutations we need to generate all the K-subsets of N of length N' , then generate all the permutations of length P . Using the K-subsets can produce similar permutations from another K-subset, so we need to remove any duplicates. The time symmetry all the RSS sequences indicates that a master cycle will give exactly the same results as the reverse of that master cycle, so when we generate the permutation list we must remove all the duplicates and reverse duplicates (i.e. [1234] is the same as [4321]). The general method to generate the permutations lists can be summarized below.

1. Determine the number of distinct sub-units, N .
2. Determine the sequence length (the total number of individual sub-units to apply), P .
3. Generate all the possible K-subsets given N sequences and the length P . For instance if $N = 2$, and $P = 3$, the available K-subsets are $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3\}$. K-subsets that are the reverse of another K-subset are also removed from further calculation.
4. For each K-subset, generate all the permutations. For instance a K-subset of $\{1, 2, 3\}$

and $P = 3$ has these permutations: $\{1, 2, 3\}$, $\{1, 3, 2\}$, $\{2, 1, 3\}$, $\{2, 3, 1\}$, $\{3, 1, 2\}$, and $\{3, 2, 1\}$.

5. Since there can be duplicate permutations within different K -subset permutation lists, remove all duplicates.
6. Remove all the reverse permutations from the master list. For instance the permutation $\{2, 3, 1\}$ would be removed because $\{1, 3, 2\}$ is already included.

To save much computational effort, the removal of reverse permutations and duplicates can occur at the time each permutation is generated. The removal of the reversed permutations for large permutations sets (sets larger than 20 items) is a computational limitation because of the searching problem. On a 700 MHz Pentium III Xeon, the generation of unique non-reversed, permutations for a length 20 system takes 2 hours. The generation of a length 40 list proved too memory intensive as over 10^{47} permutations would need to be searched for duplicates and reversed duplicates. Given a list that large (even if it was cut in half) would also prove prohibitively time demanding to calculate the effective Hamiltonians for all $10^{47}/2$ sequences. In practice, the simulations used either $N = 2$ or $N = 4$ distinct sequences. The maximal length, P , for the sequences was found to be 20 for $N = 2$ and 12 for $N = 4$ that can be handled without huge memory and time requirements. We will apply the label $N \times P$ for each of the calculated permutation data segments. For a 2×20 segment there are 92504 unique non reversible permutations and for a 4×12 segment there are 184800 unique non reversible permutations.

As an algorithmic point, only integers representing each sub-unit are necessary to perform the permutation calculation. This saves much time when both generating the permutations and comparing them for duplication.

Table 6.2: Sequence Permutation set for the effective Hamiltonian calculations of the post- $C7$ sequence.

N	P	basic Units (length)
2	(2,4,8,12,16,20)	o, O (2,4,8,12,16,20) w, W (2,4,8,12,16,20) oO, wW (4,8,16,24,32,40) wO, oW (4,8,16,24,32,40) ow, OW (4,8,16,24,32,40) oOOo, wWWw (8,16,32,48) oOWw, OowW (8,16,32,48) oOOo, WwwW (8,16,32,48)
4	(4,8,12)	o, O, w, W (4,8,12) oOOo, wWWw, Ww, Oo (12, 40)

Because the 2×20 and 4×12 was our computer limitation, and many of the desired sequences are applied for many more cycles than 20 or 12, the third stage of the program allows the ability to use any number of sub permutations for each index N. To calculate all the effective Hamiltonians and their spin operator components in a 2×20 system for 2 spins spinning at a rotor speed of 5000 Hz for 1154 powder average points took 5 days on a single processor. The program is able to distribute the powder average over multiple workstations to allow linear scaling of the calculation. Table 6.2 shows the calculated sequences calculated for the post- $C7$ permutations.

The next stage is generating all of the spin tensors desired to figure out the tensor components. The tensors themselves can be generated using similar permutation techniques as the $N \times P$ by labeling each spin by an integer and each direction as an integer. Table 6.3 shows which tensors were used for this study.

Table 6.3: Spin operators and tensors generated to probe the effective Hamiltonians

Type	Form
1 st order Cartesian	$I_r^i, (r = x, y, z)$
2 nd order Cartesian	$I_r^i I_{r'}^{i'}, (r = r' = x, y, z)$
1 st order spherical	$T_{l,m}^{(i,i')}, (l = 1, 2, m = -l...l)$
2 nd order spherical	$T_{l,m}^{(i,i')} T_{l',m'}^{(i'',i''')}, (l = l' = 1, 2, m = m' = -l...l)$

Table 6.4: Spin System parameters for the three sets of permutations. All units are in Hz

System Label	Spin parameters
SS_1	$^{13}C_1$ CSA ($\delta_{iso} = 1254, \delta_{ani} = 12345, \eta = 0$)
	$^{13}C_2$ CSA ($\delta_{iso} = -1544, \delta_{ani} = 8552, \eta = 0$)
	$^{13}C_1$ - $^{13}C_2$ dipole ($\delta_{ani} = 2146$)
SS_2	$^{13}C_1$ CSA ($\delta_{iso} = 1254, \delta_{ani} = 12345, \eta = 0$)
	$^{13}C_2$ CSA ($\delta_{iso} = -1544, \delta_{ani} = 8552, \eta = 0$)
	$^{13}C_1$ - $^{13}C_2$ dipole ($\delta_{ani} = 2146$)
	1H_3 - $^{13}C_1$ dipole ($\delta_{ani} = 4506$)
	1H_3 - $^{13}C_2$ dipole ($\delta_{ani} = 7564$)
SS_3	$^{13}C_1$ CSA ($\delta_{iso} = 1254, \delta_{ani} = 12345, \eta = 0$)
	$^{13}C_2$ CSA ($\delta_{iso} = -1544, \delta_{ani} = 8552, \eta = 0$)
	$^{13}C_1$ - $^{13}C_2$ dipole ($\delta_{ani} = 2146$)
	1H_3 - $^{13}C_1$ dipole ($\delta_{ani} = 4506$)
	1H_3 - $^{13}C_2$ dipole ($\delta_{ani} = 7564$)
	1H_4 - $^{13}C_1$ dipole ($\delta_{ani} = 2150$)
	1H_4 - $^{13}C_2$ dipole ($\delta_{ani} = 4562$)
1H_3 - 1H_4 dipole ($\delta_{ani} = 15546$)	

6.4 Data and Results

6.4.1 Sequence Measures

There were over 500000 different $C7$ master cycles simulated and measured. The next few figures will show the data, giving both the M_R and M_{\max} for all the sequences of a given number of $C7$ cycles as well as the best one showing you the tensor components. There are three different sets of data corresponding to 3 different numbers of nuclei. Table 6.4 shows the spin system parameters for each set.

Table 6.5: Relevant weighting factors for Eq. 6.17

Tensor	weight($b_{l,m}$)
I_z^i	0.2381 (a factor of 5)
$I_{x,y}^i$	0.0952 (a factor of 2)
$T_{2,0}^{i,j}$	0 (a factor of 0)
$T_{2,\pm 1}^{i,j}$	0.0476 (a factor of 1)

The spin parameters were chosen to avoid any rotational resonance conditions with either the spinning rate or the RF amplitude. They were also chosen as a representative organic molecule so dipole and CSA values are consistent with peptides and amino acids (although no one amino acid was used). The couplings were chosen to be all the same order of magnitude as the spinning frequency $\omega_r = 5kHz$ as to be in the regime of truly non-ideal conditions where the benefits of the permutation cycles would show more dramatically. If the spinning rate (and consequently the RF power) are high, then the average Hamiltonian series converges must faster as each order falls off like $(1/\omega_r)^n$. As with most of the RSS sequences, an experimental limit is usually reached with an RF power of 150kHz. For the $C7$ this implies a maximum rotor speed of about 20 kHz. For other CN sequences ω_r is much less, so 5000 kHz is a good value to investigate the properties of the sequences.

To handled the data more effectively, only the first order tensors of Table 6.3 were considered in the M_R measure. The higher order tensors were recorded, but as stated before, the commutation relations of 2 spin 1/2 nuclei reduce them all to first order tensors. The higher order tensors can give better insight as to the coherence pathways the error terms follow, which could potentially be used to construct sequences and phase cycles that remove these pathways. The relevant weighting factors for Eq. 6.17 are given in Table 6.5.

Figures 6.6–6.14 show the data recorded for the SS_1 system for a total sequence length of 4,8,12,16,20,24,32,40,48 respectively. Figures 6.15–6.20 show the data recorded for the SS_2 system for a total sequence length of 4,8,12,16,24,32 respectively. Figures 6.21–6.26 show the data recorded for the SS_3 system for a total sequence length of 4,8,12,16,24,32 respectively.

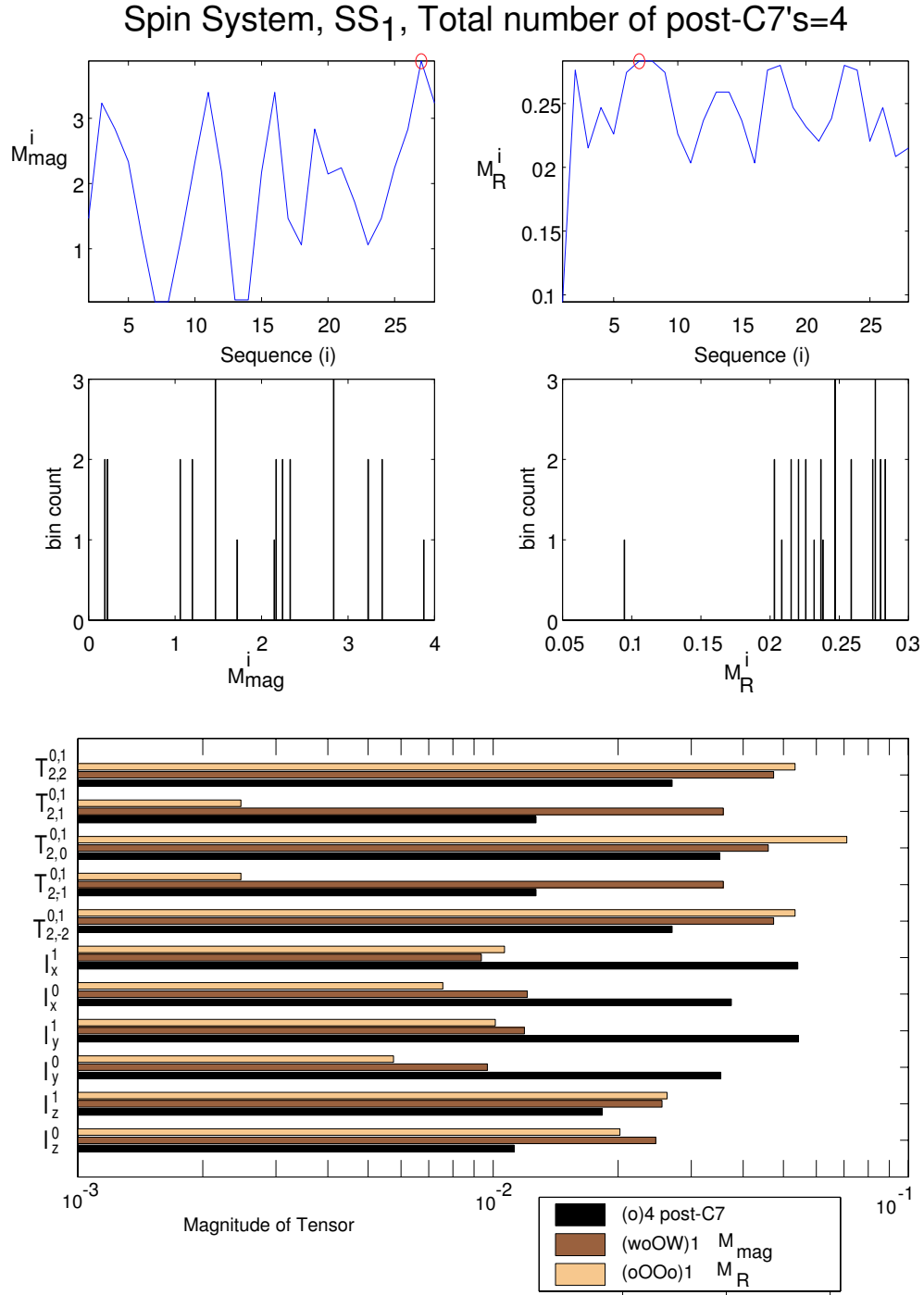


Figure 6.6: Spin system SS_1 with 4 total number of $C7$'s applied.

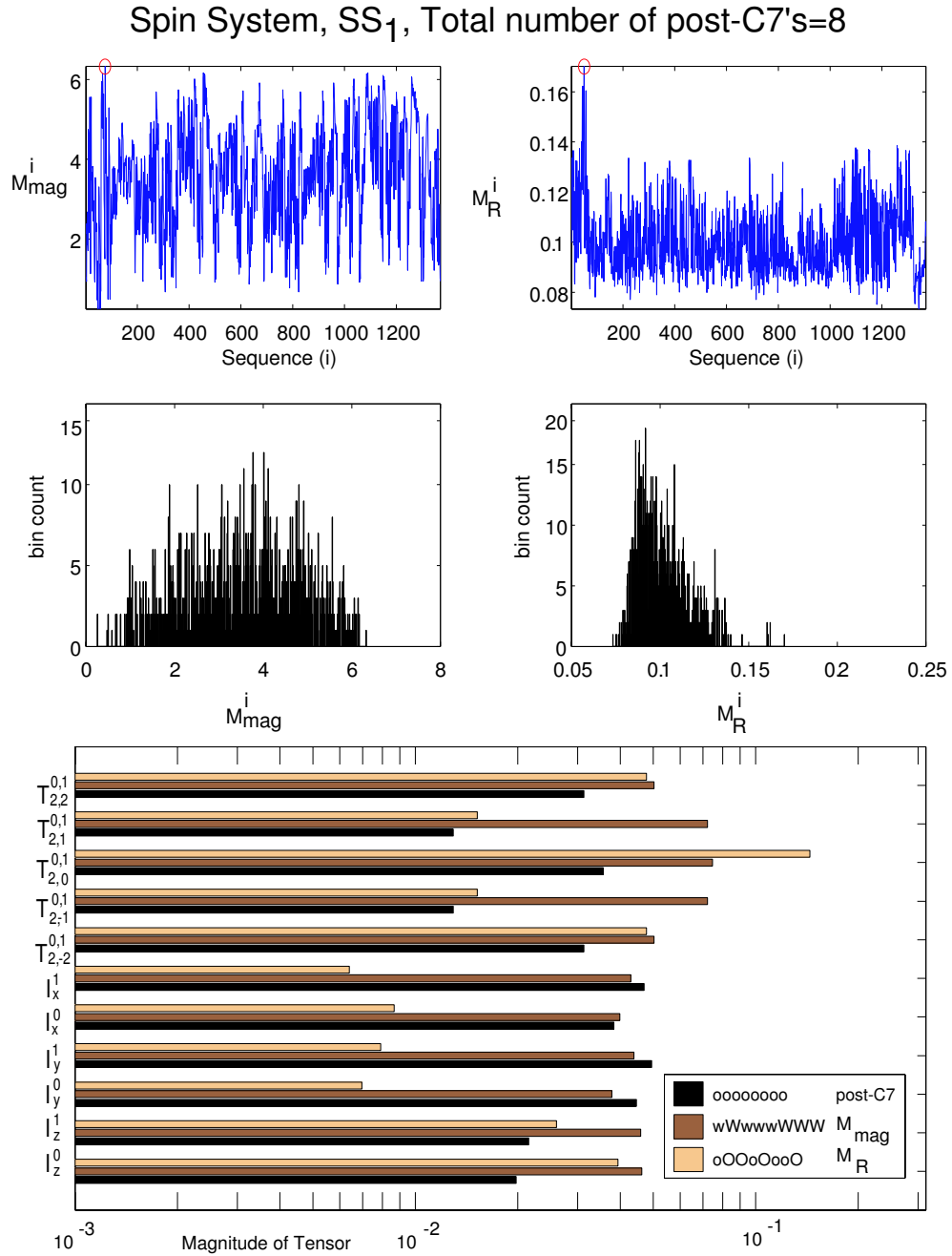


Figure 6.7: Spin system SS_1 with 8 total number of $C7$ s applied.

Spin System, SS_1 , Total number of post-C7's=12

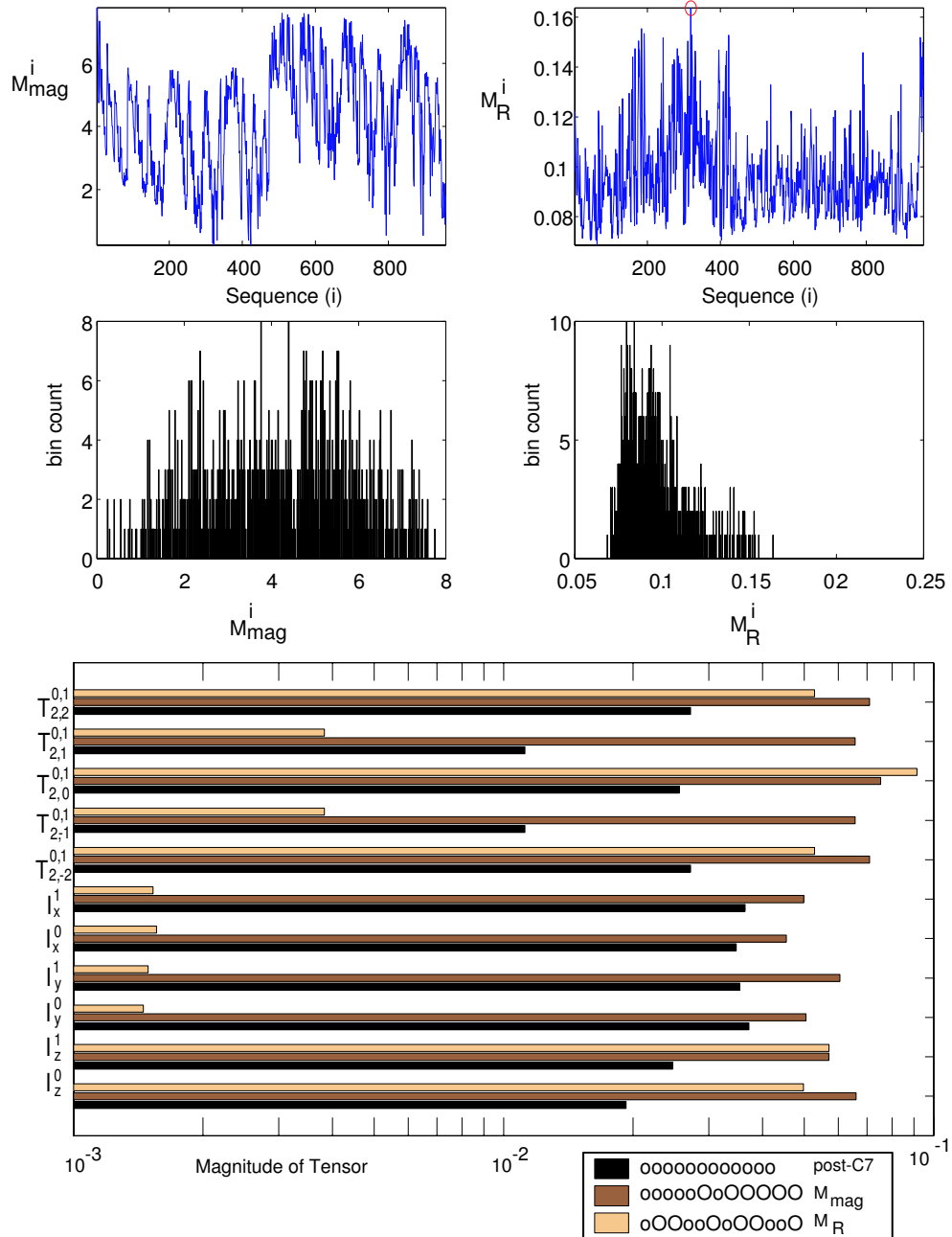


Figure 6.8: Spin system SS_1 with 12 total number of $C7$ s applied.

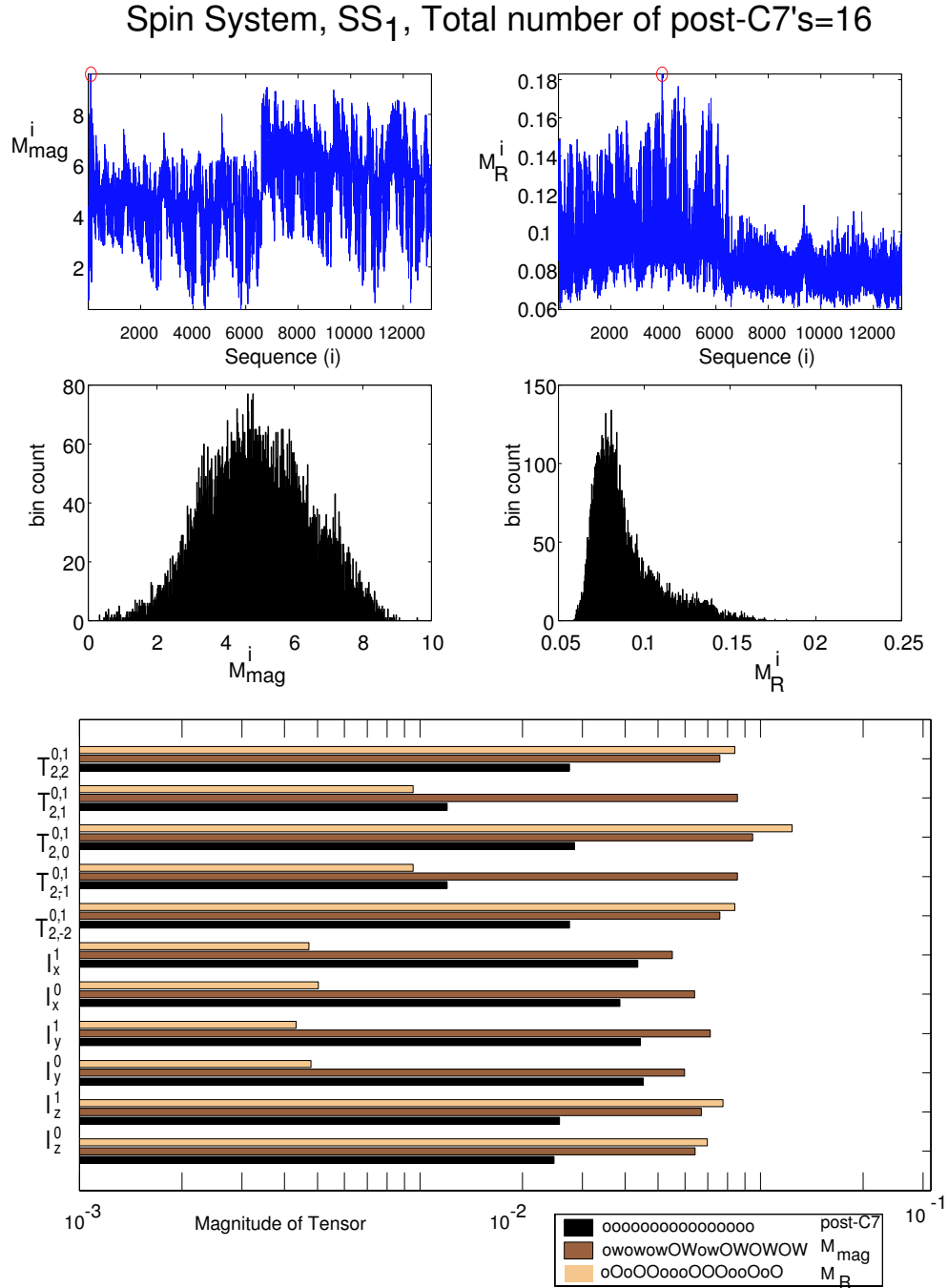


Figure 6.9: Spin system SS_1 with 16 total number of $C7$ s applied.

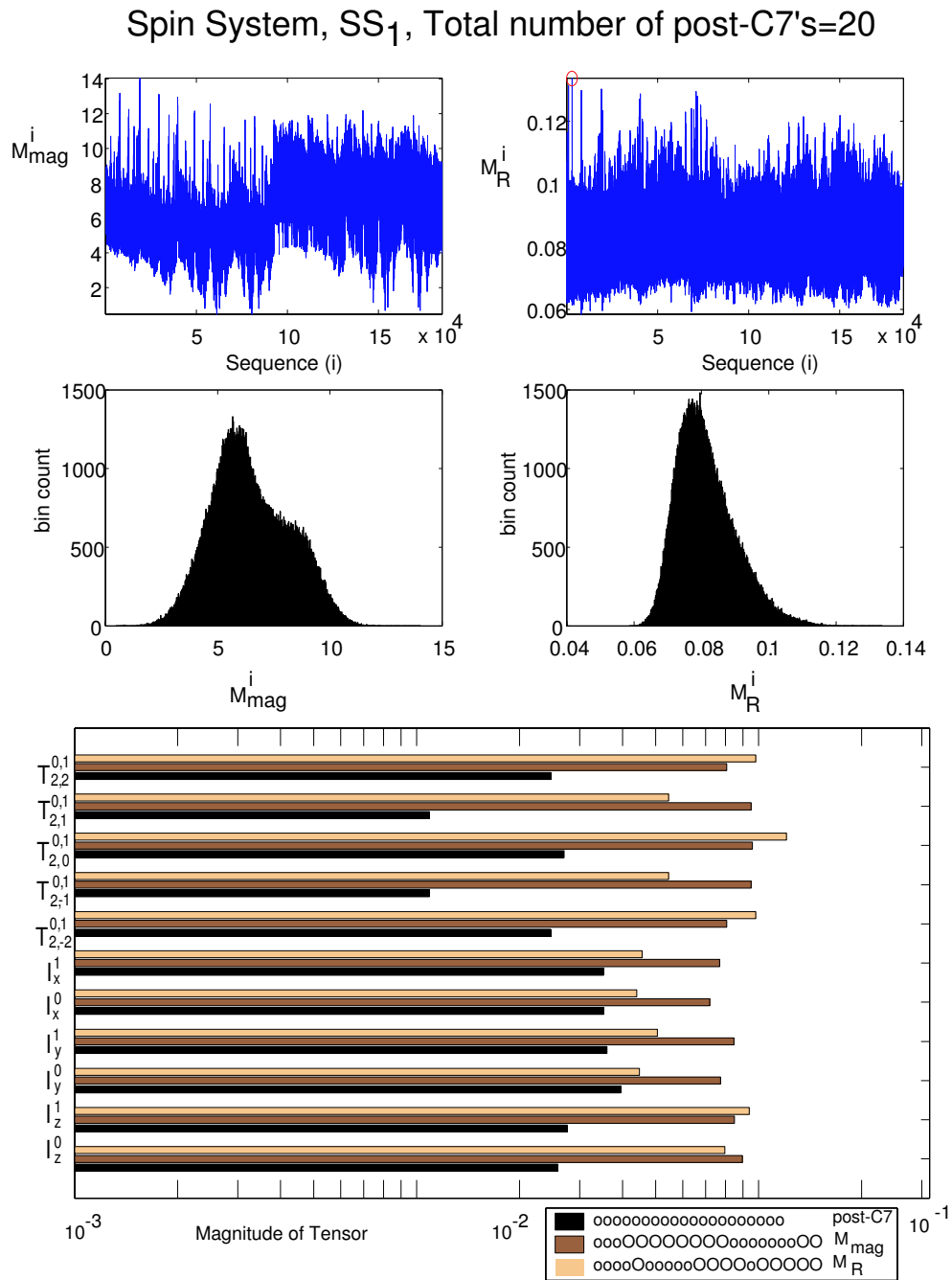


Figure 6.10: Spin system SS_1 with 20 total number of $C7$ s applied.

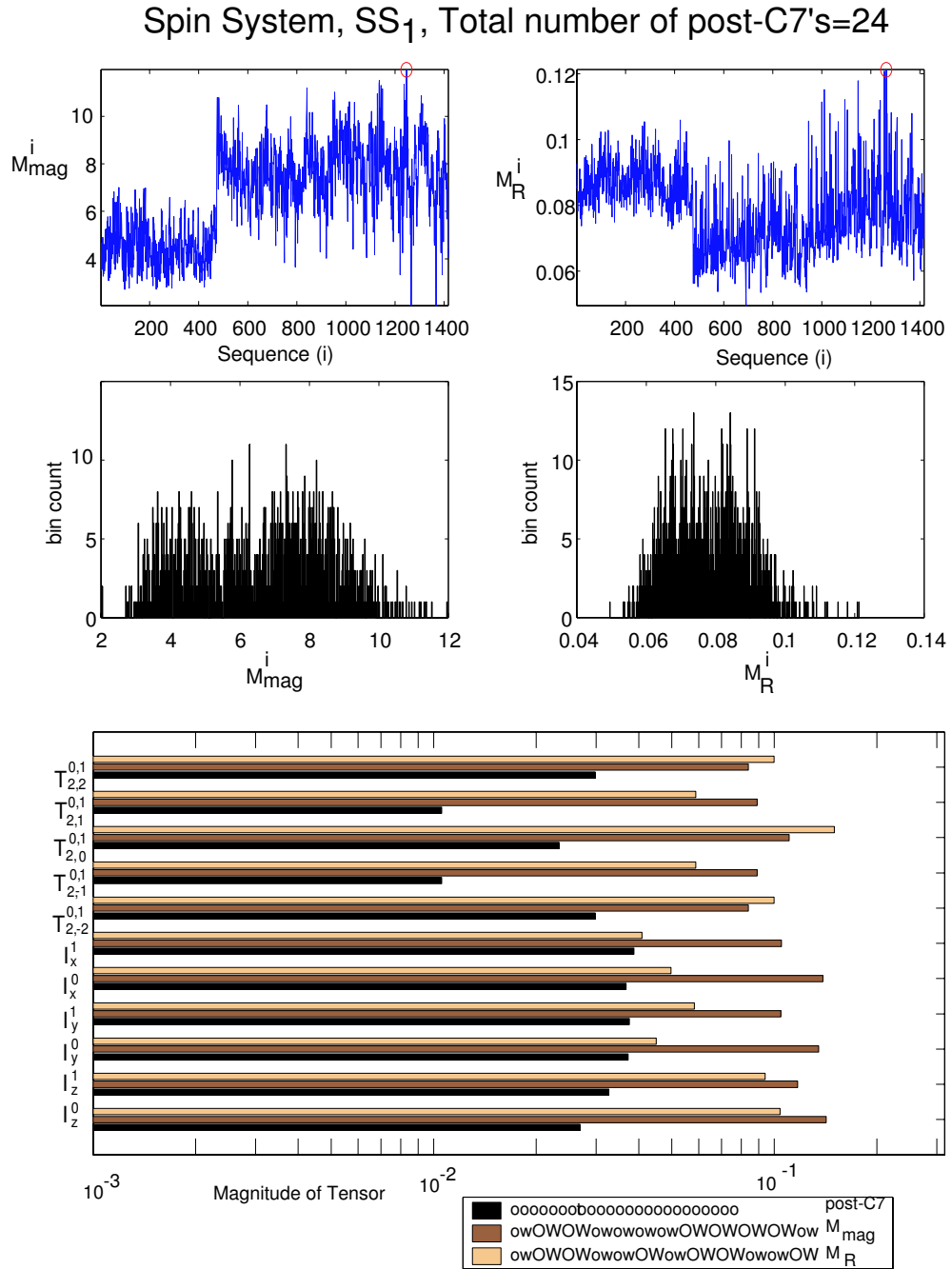


Figure 6.11: Spin system SS_1 with 24 total number of $C7$ s applied.

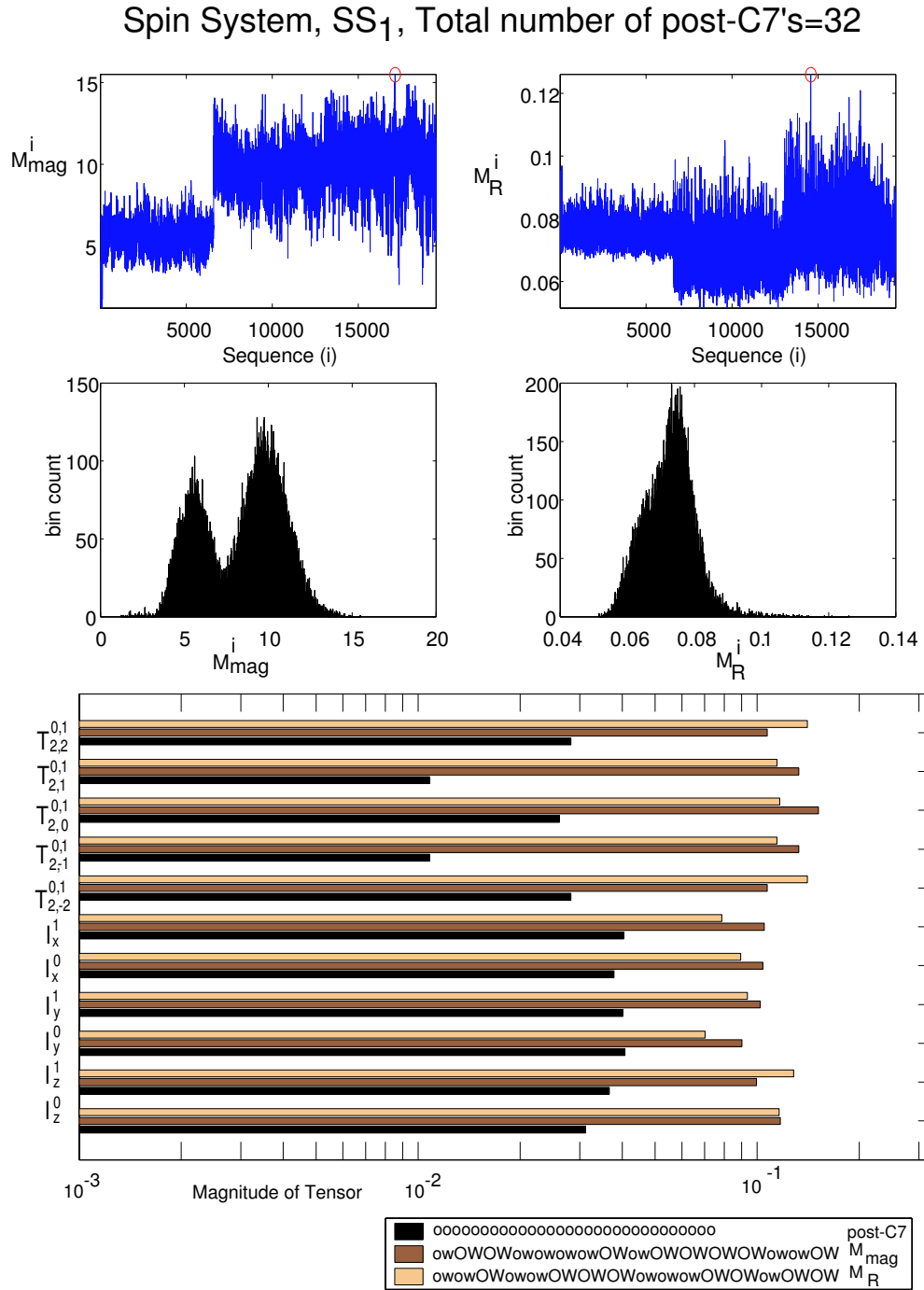


Figure 6.12: Spin system SS_1 with 32 total number of $C7$ s applied.

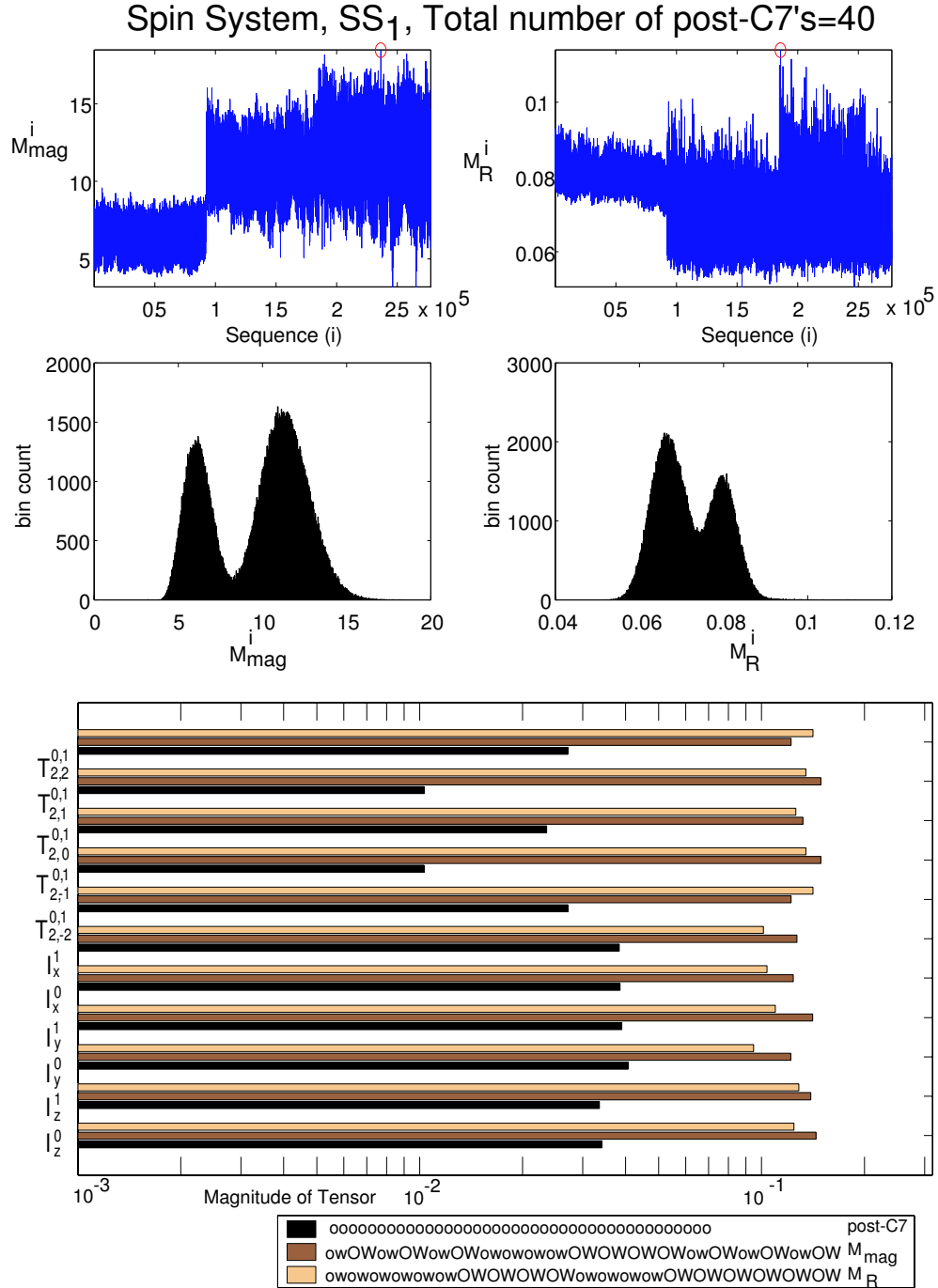


Figure 6.13: Spin system SS_1 with 40 total number of $C7$ s applied.

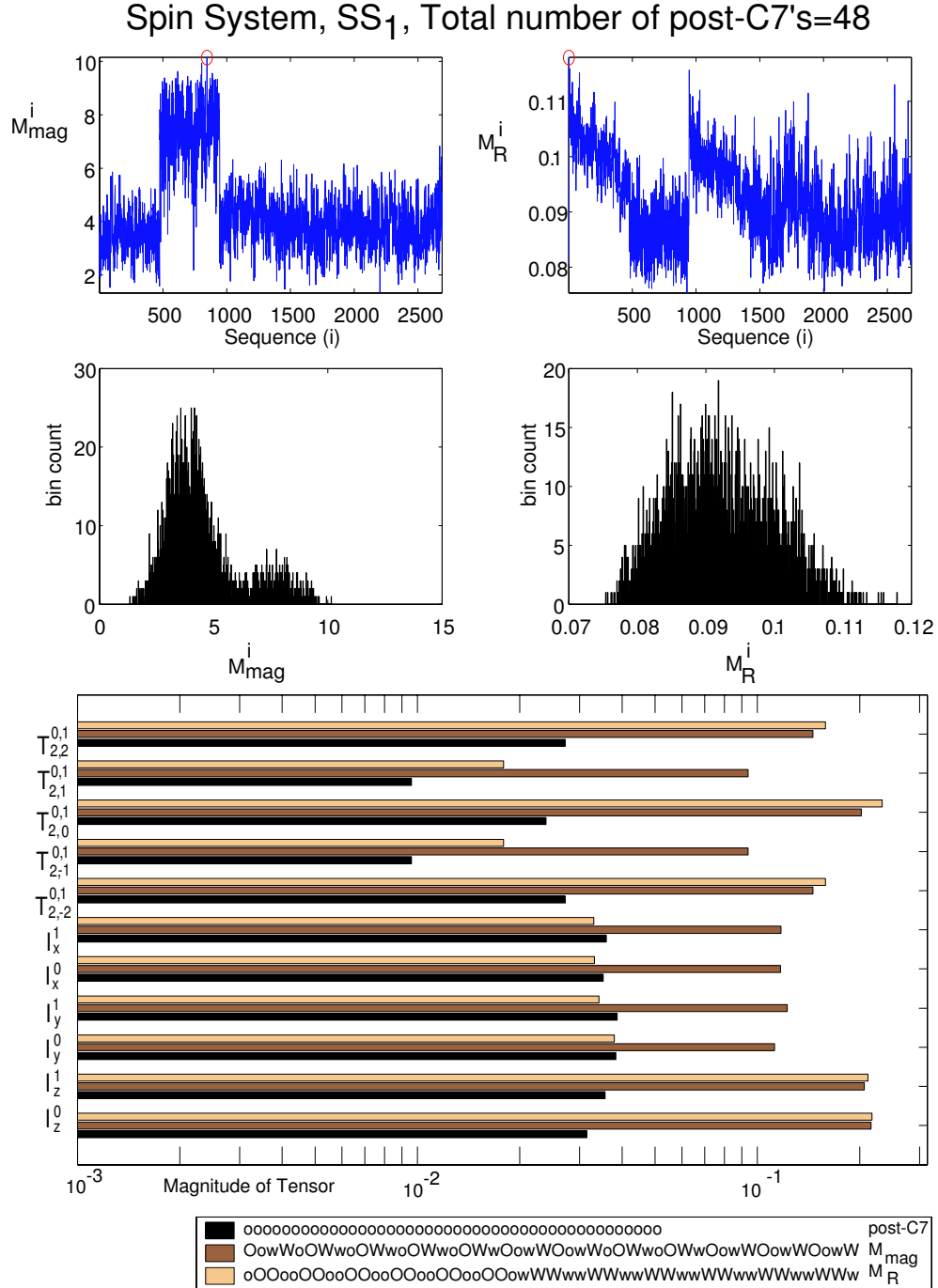


Figure 6.14: Spin system SS_1 with 48 total number of $C7$ s applied.

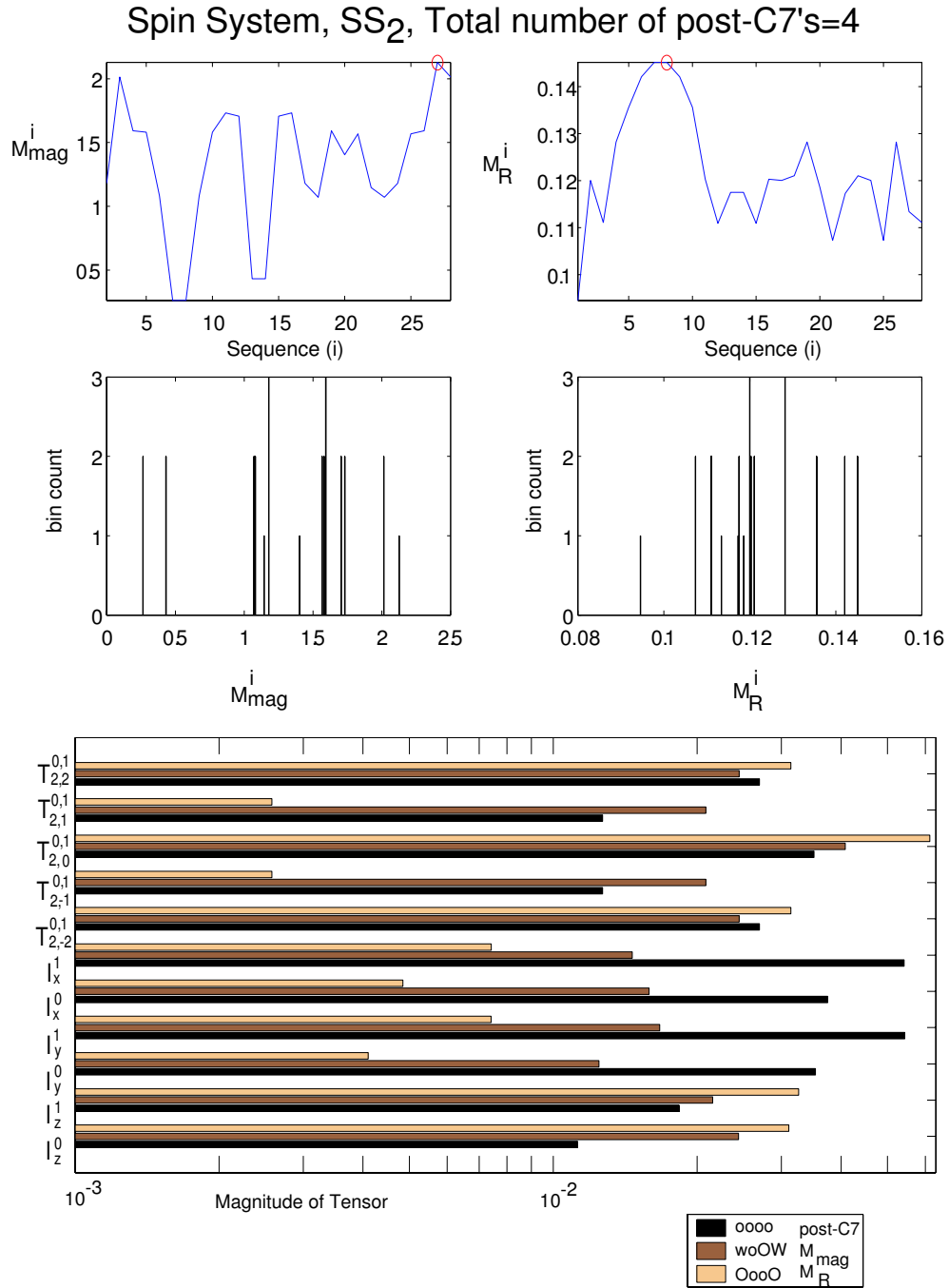


Figure 6.15: Spin system SS_2 with 4 total number of $C7$ s applied.

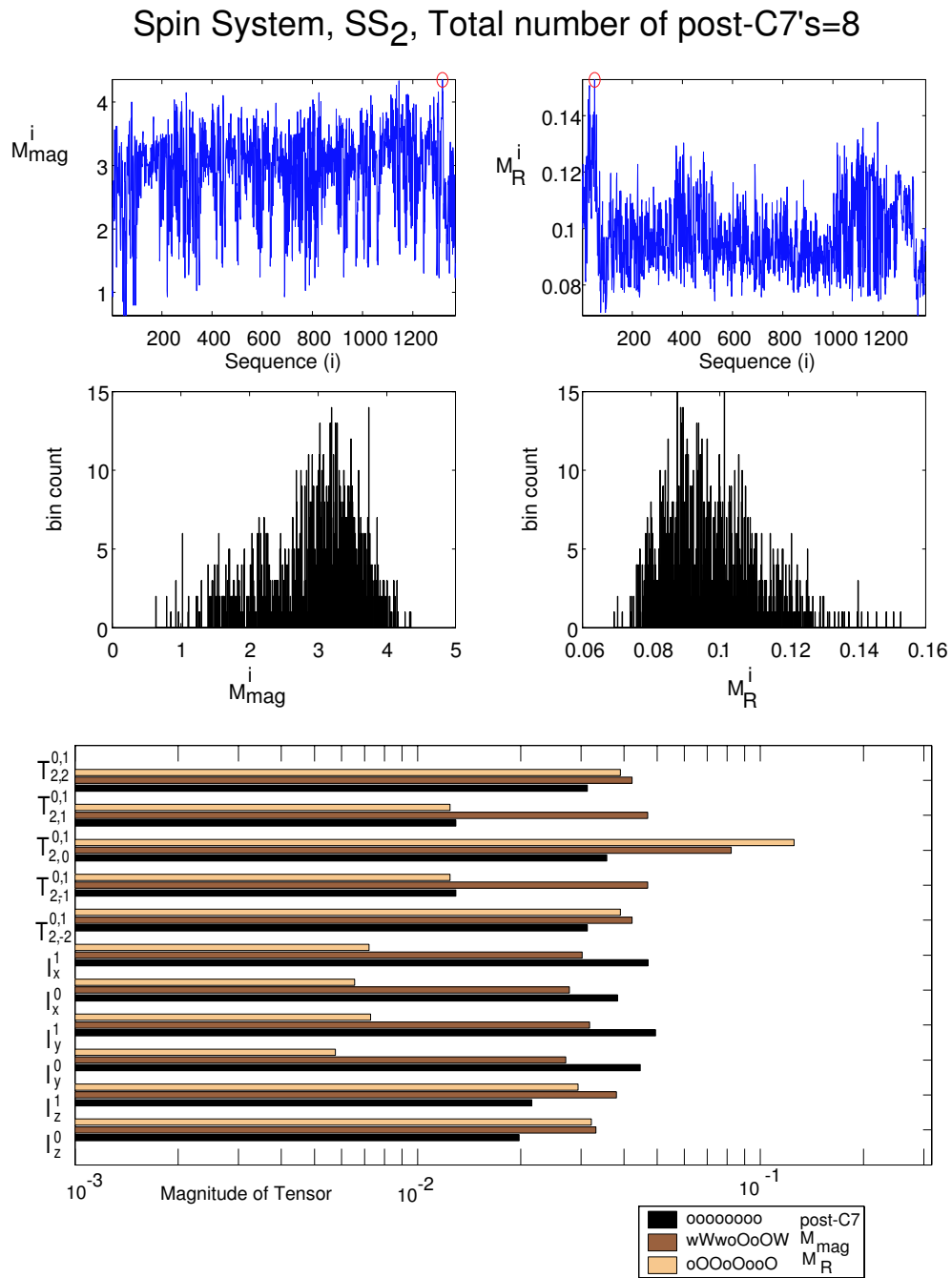


Figure 6.16: Spin system SS_2 with 8 total number of $C7$ s applied.

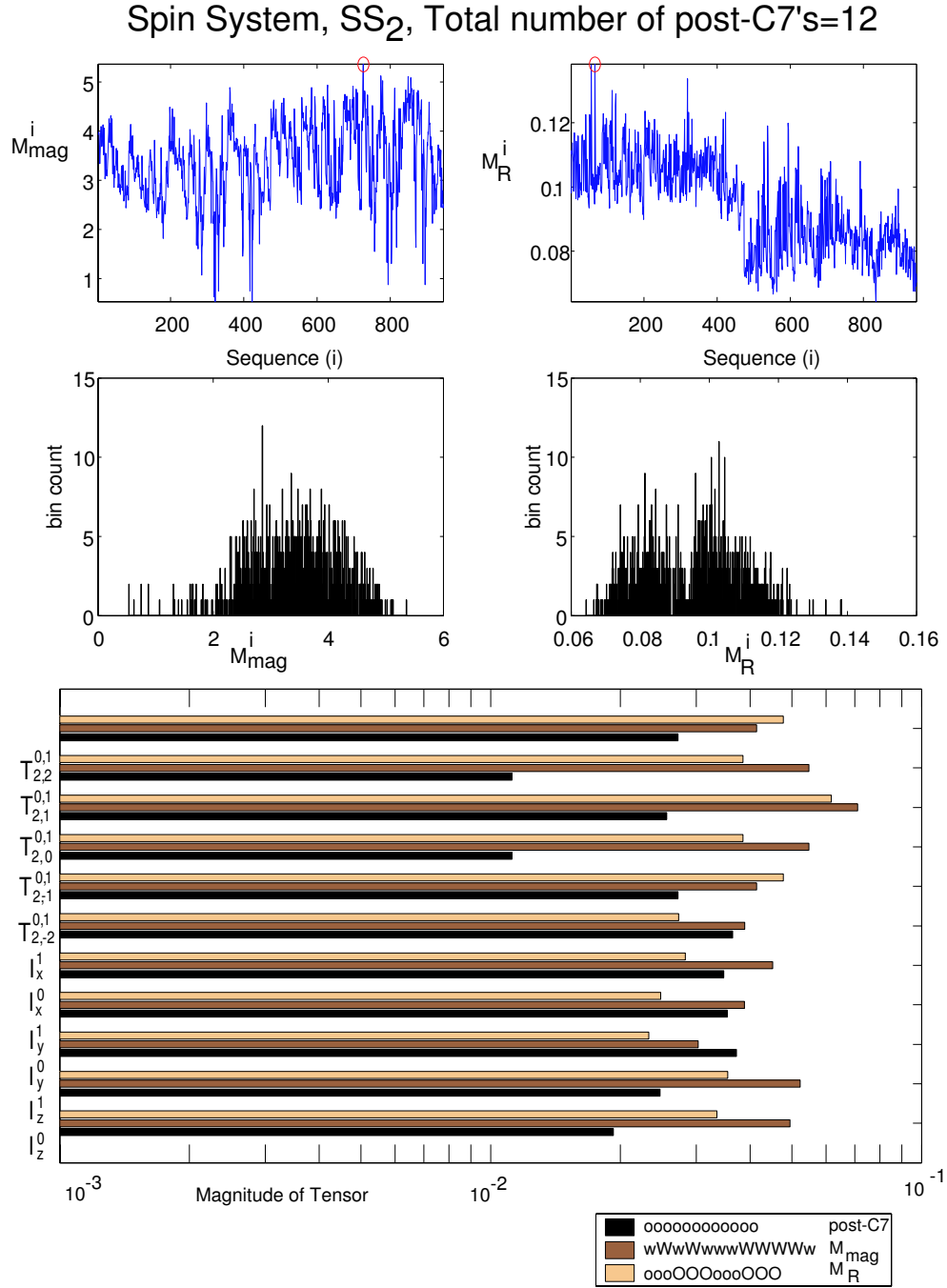


Figure 6.17: Spin system SS_2 with 12 total number of $C7$ s applied.

Spin System, SS_2 , Total number of post-C7's=16

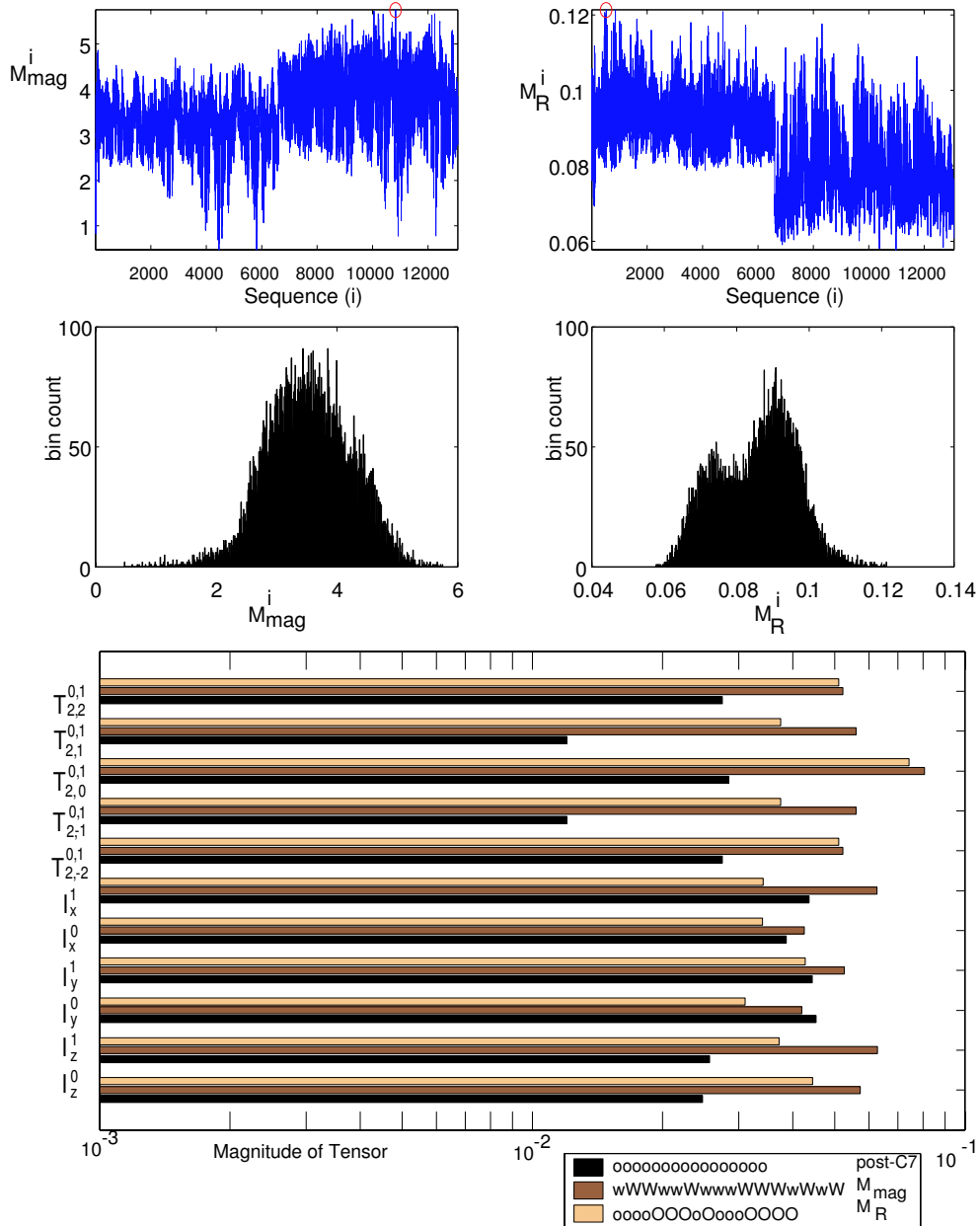


Figure 6.18: Spin system SS_2 with 16 total number of $C7$ s applied.

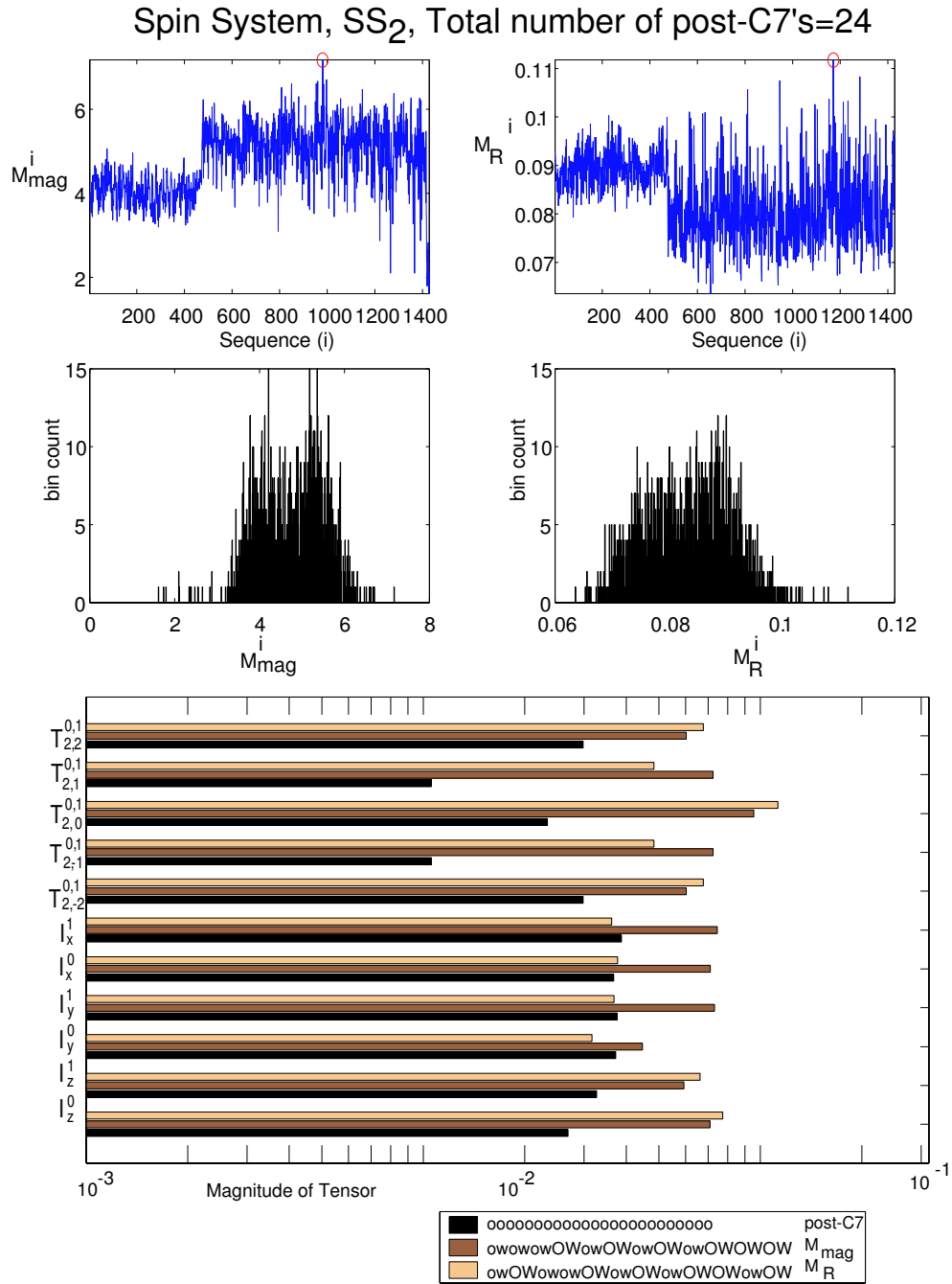


Figure 6.19: Spin system SS_2 with 24 total number of $C7$ s applied.

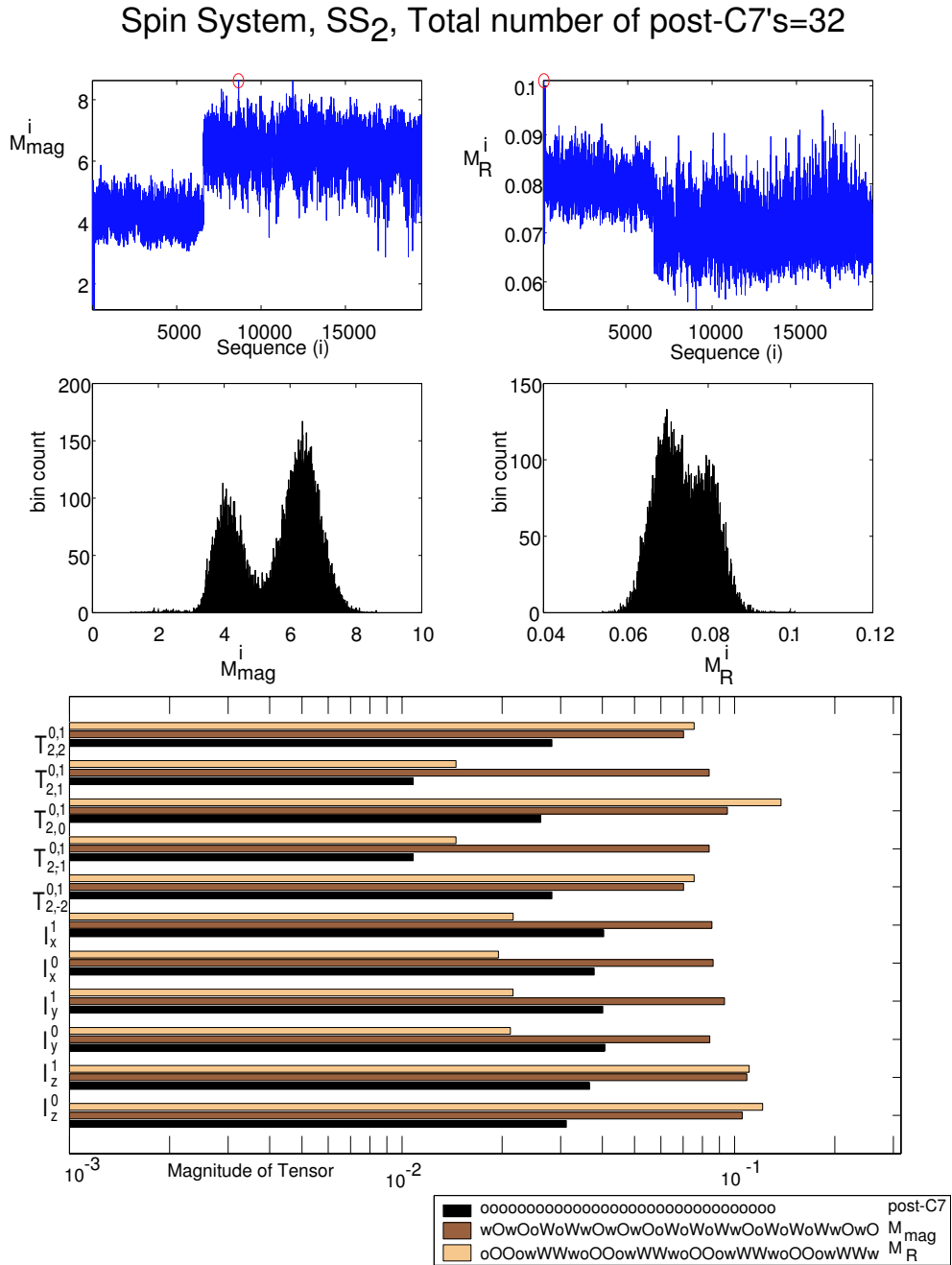


Figure 6.20: Spin system SS_2 with 32 total number of $C7$ s applied.

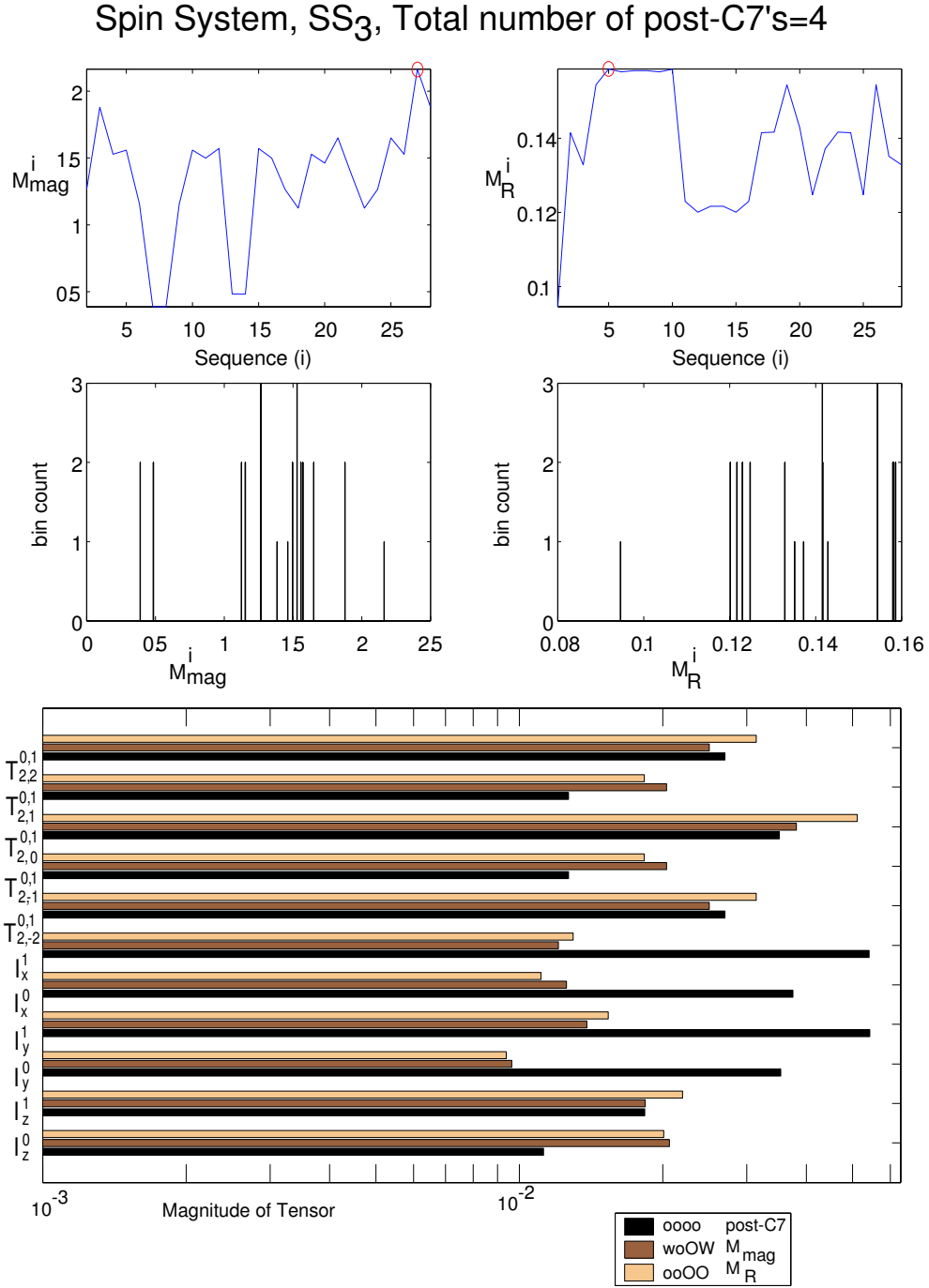


Figure 6.21: Spin system SS_3 with 4 total number of $C7$ s applied.

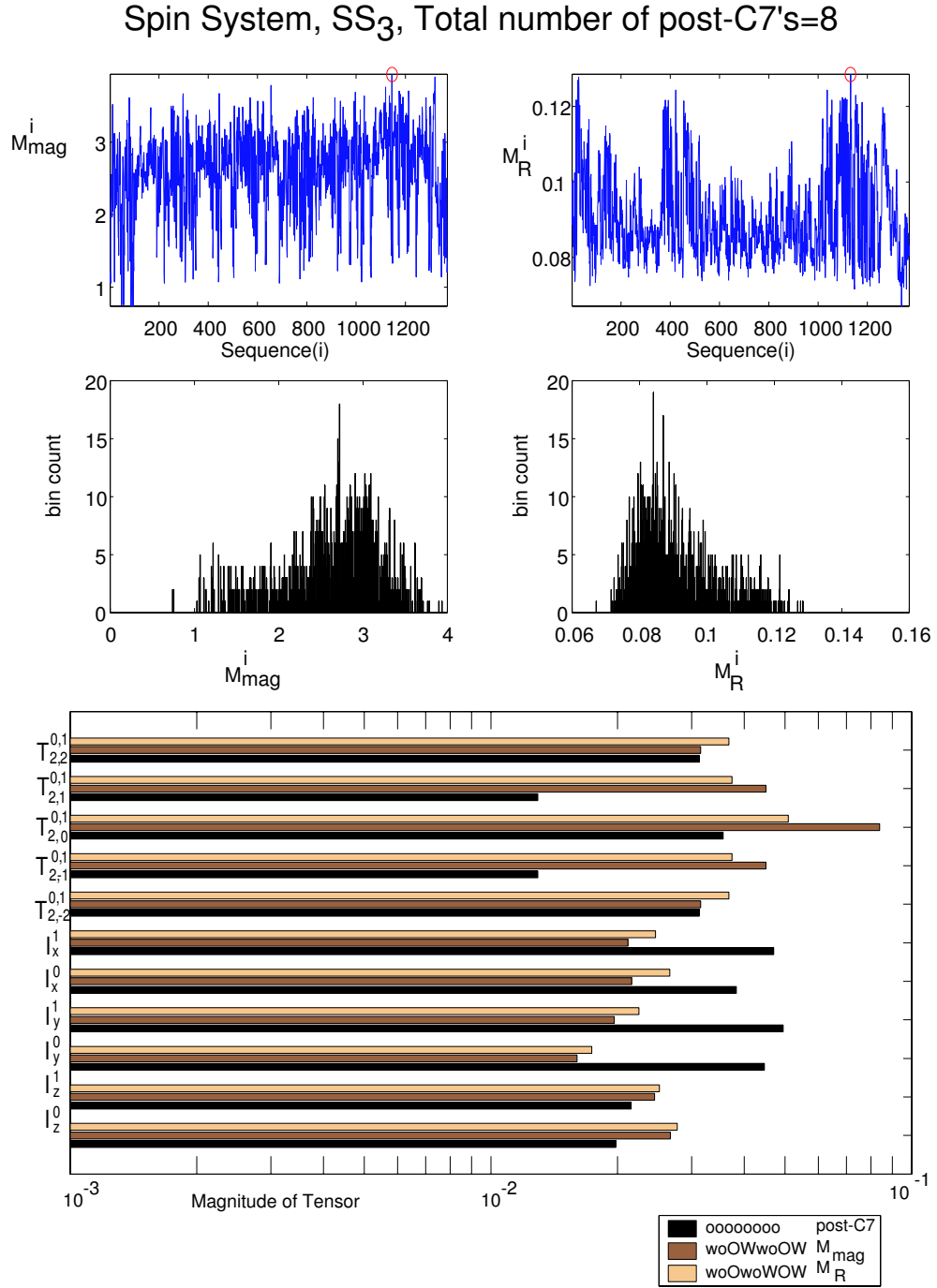


Figure 6.22: Spin system SS_3 with 8 total number of $C7$ s applied.

Spin System, SS_3 , Total number of post-C7's=12

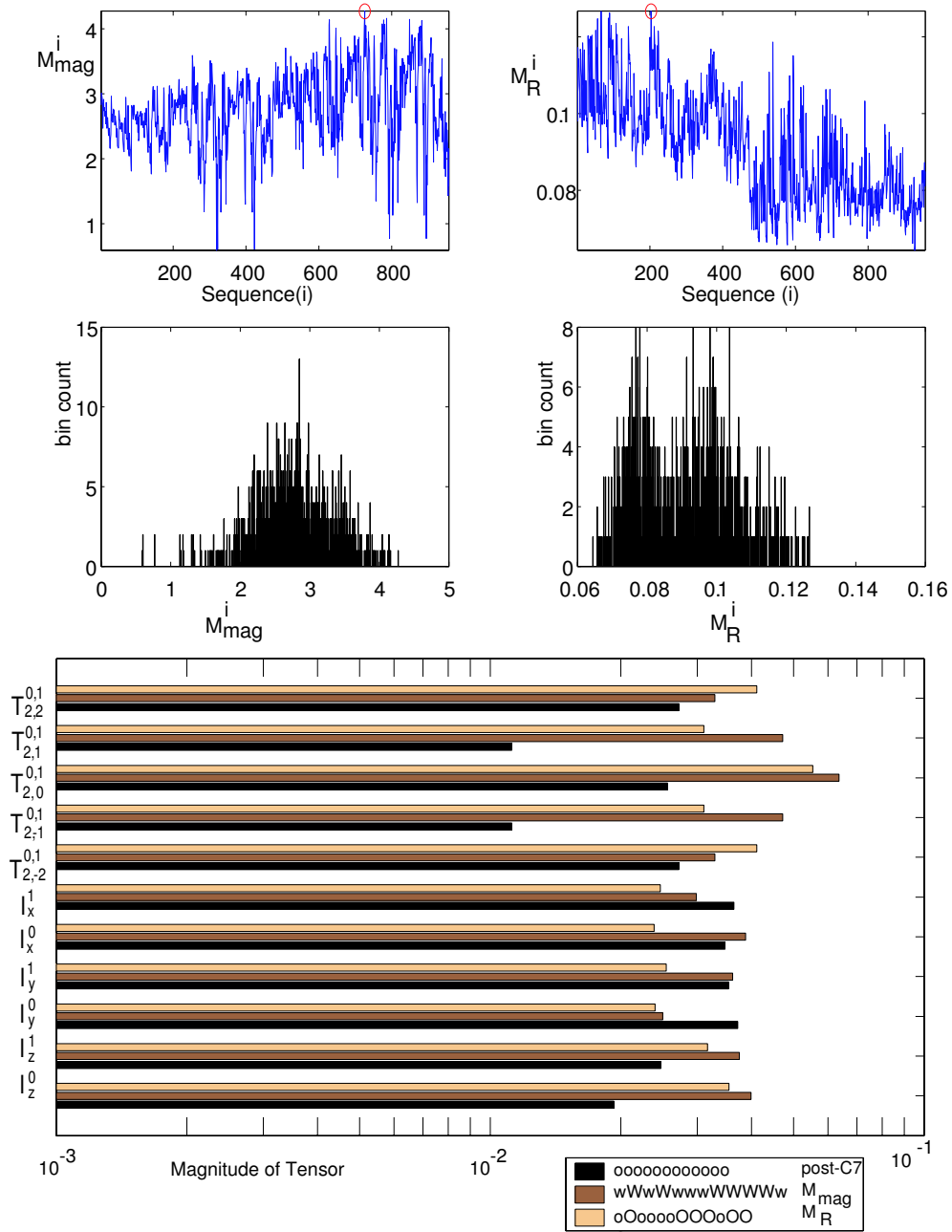


Figure 6.23: Spin system SS_3 with 12 total number of $C7$ s applied.

Spin System, SS_3 , Total number of post-C7's=16

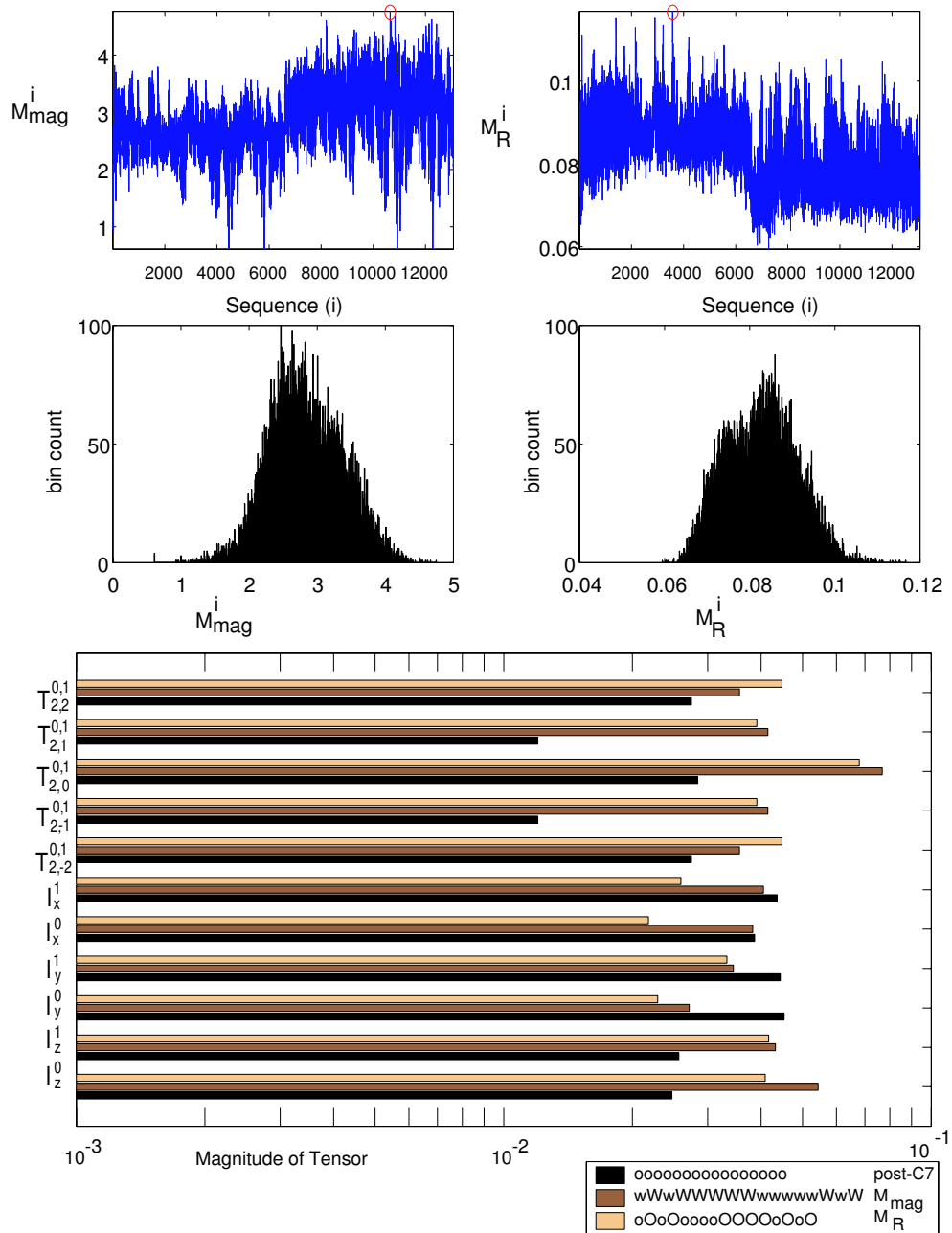


Figure 6.24: Spin system SS_3 with 16 total number of $C7$ s applied.

Spin System, SS_3 , Total number of post-C7's=24

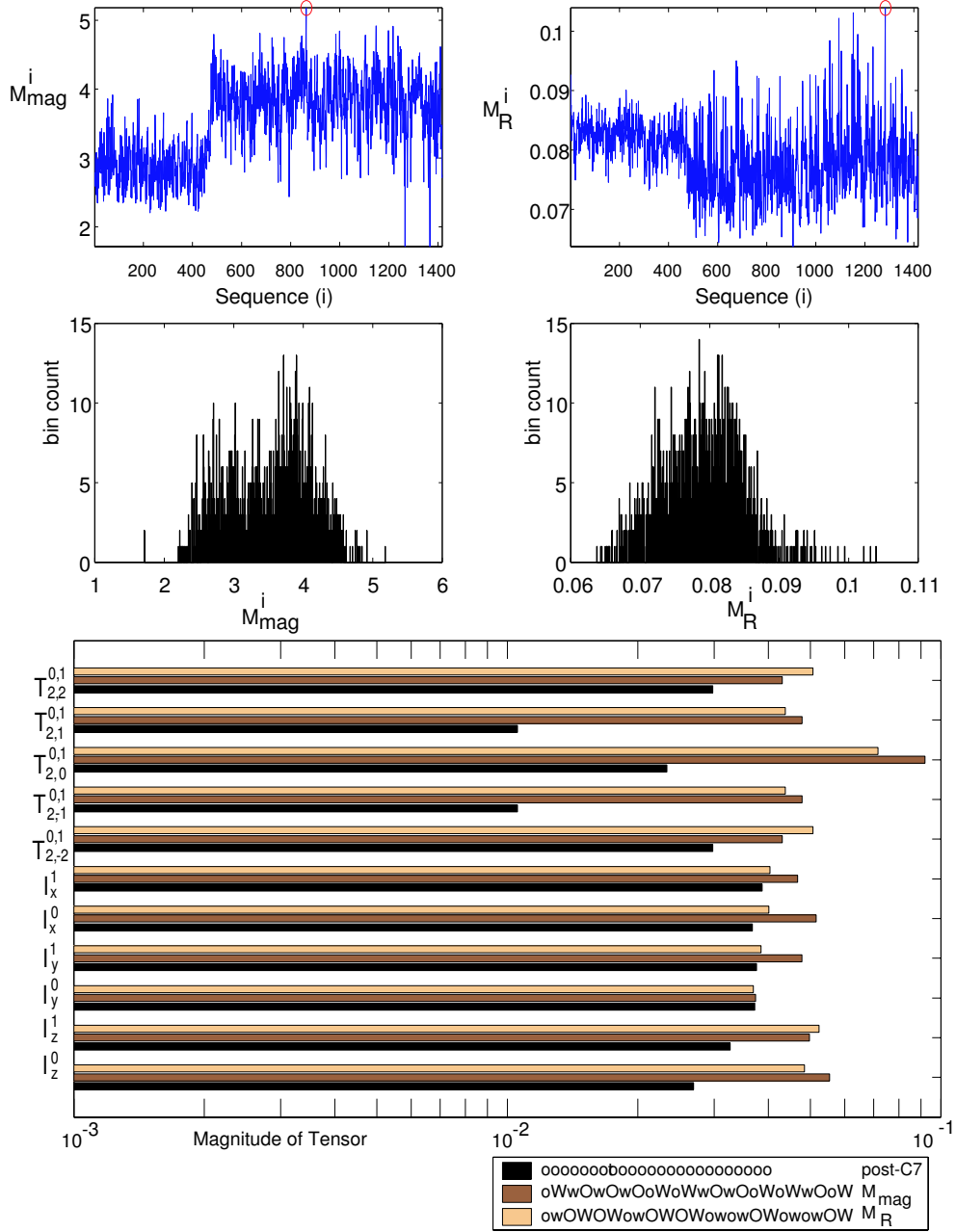


Figure 6.25: Spin system SS_3 with 24 total number of $C7$ s applied.

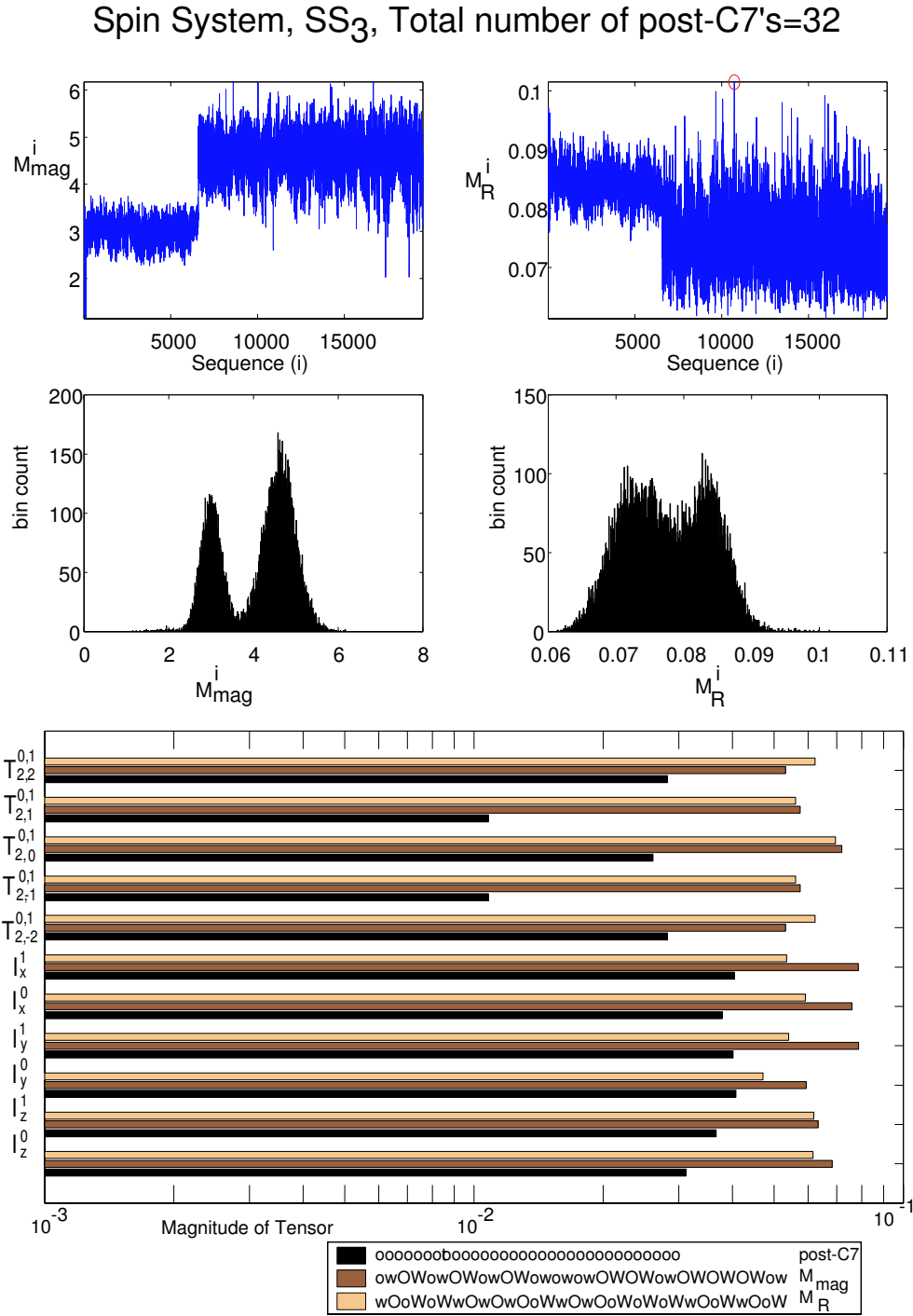


Figure 6.26: Spin system SS_3 with 32 total number of $C7$ s applied.

6.4.2 Transfer Efficiencies

The amount of data is quite overwhelming however; the results can be consolidated into a single sentence. Small permutation cycles (i.e. the total number of $C7$ was less than 8) give expected results from symmetry considerations, any master cycles where the total number of $C7$ s is greater than 8 give results seemingly uncorrelated results using only the basic symmetry principles. This is in fact what we were looking for. It does not necessarily mean that for the longer sequences, that symmetry considerations could not have produced the desired sequences. In fact the generated sequences are the best conditions to cancel higher order terms, which if the full average Hamiltonian sequence was generated could have designed by hand. Of course calculating the average Hamiltonian sequence is a much harder problem than simply probing the effectiveness of a given sequence. Table 6.6 lists the best permutation cycles found for each spin system and total number of $C7$ calculated.

In both the SS_2 and the SS_3 data sets, the protons were not decoupled from the ^{13}C nuclei, and the generated sequences are thus different for each spin system. Ideal decoupling of the protons would give the same sequences as SS_1 . There is a problem using decoupling in RSS sequences. As we are applying a continuous rotation to the carbons, the RF power applied to the protons must be larger than the power applied to the carbons for decoupling otherwise we would effectively synchronize the motions of the two nuclei, creating more recoupling than decoupling. The power ratio condition has been empirically found to be $\omega_{rf}^{decoupling} > 3 * \omega_{rf}^{^{13}C}$ [108, 135, 136]. For large N or large spinning rates in RSS sequences this is very hard to satisfy experimentally. Because $\omega_{rf}^{^{13}C}$ can be large itself, we should be able to use this as our decoupling field. One can then use similar the symmetry

Table 6.6: Best $C7$ permutation sequences for each spin system and $C7$ cycle length.

System	permutations	
SS_1	length	best permutation
	4	oOOo
	8	oOOoOooO
	12	oOOooOoOOooO
	16	oOoOOoooOOOooOoO
	20	ooooOoooooOOOOoOoooo
	24	owOWOWowowOWowOWOWowowOW
	32	owowOWowowOWOWOWowowowOWOWowOWOW
	40	owowowowowowOWOWOWOWowowowowOWOWOWOWOWOW
48	oOOooOooOOooOOooOOooOOowWWwwWWwwWWwwWWwwWWwwWWww	
SS_2	length	best permutation
	4	OooO
	8	oOOoOooO
	12	oooOOOoooOOO
	16	ooooOOOoOoooOOOO
	32	owOWowowOWowOWowOWOWowOW oOOowWWwoOOowWWwoOOowWWwoOOowWWw
SS_3	length	best permutation
	4	ooOO
	8	woOwoWOW
	12	oOooooOOOoOO
	16	oOoOooooOOOOoOoO
	32	owOWOWowOWOWowowOWowowOW wOoWoWwOwOwOoWwOwOoWoWoWwOoWwOow

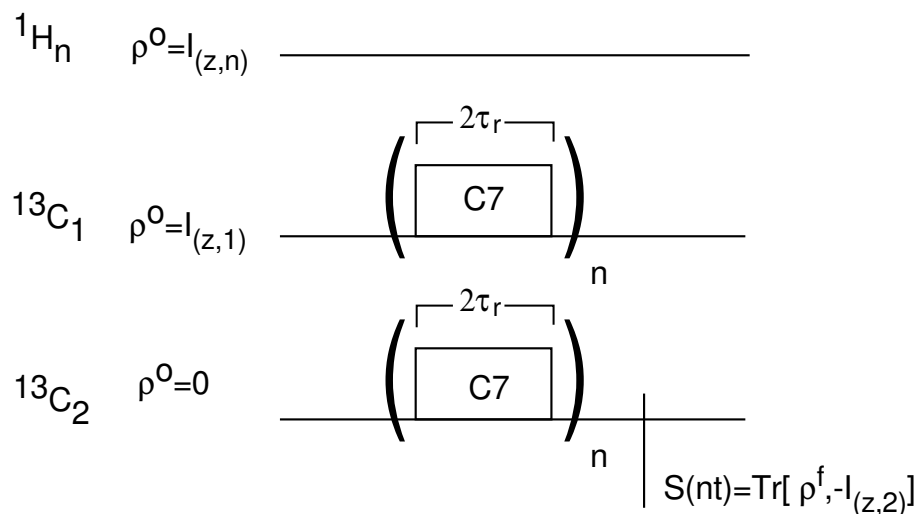


Figure 6.27: Pulse sequence, initial density matrices and detection for a transfer efficiency measurement.

considerations to also remove higher order ${}^1\text{H} - {}^{13}\text{C}$ cross terms. For systems SS_2 and SS_3 the search found permutation sequences that minimized these as well simply because the larger a $T_{2,\pm 2}$ term the less the ${}^1\text{H} - {}^{13}\text{C}$ cross terms as our polarization is conserved¹.

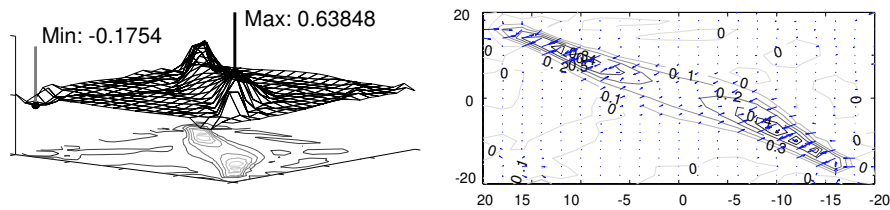
To investigate the effectiveness of the generated sequences, we looked at the transfer efficiencies over a range of offset conditions. The applied pulse sequence is shown in Figure 6.27. The efficiencies for the original $C7$ and the post- $C7$ are shown in Figure 6.28 for the SS_1 system changing only the offset parameters of ${}^{13}\text{C}_1$ and ${}^{13}\text{C}_2$. The basic $C7$ is only effective when the difference between two offsets is zero, with dramatic increases when a rotational resonant condition is met ($|\delta_{iso}^1 - \delta_{iso}^2| = n\omega_r$). The post- $C7$ is effective over a much wider range of offsets, with a sharp drop after a positive offset difference over the spinning rate.

The next few figures will show the transfer efficiencies for each of the best sequences as determined from the total $C7$ length of 4, 8, 12, and 16, comparing them to the original

¹Unitary evolution cannot increase the polarization of the system.

Transfer Efficiencies for SS_1 spin System vs. Offsets
For the Basic C_7 sequences

C_7 after 4 applications



post- C_7 after 4 applications

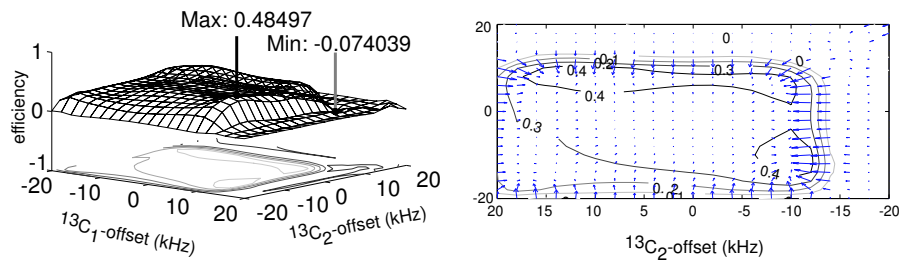


Figure 6.28: Transfer efficiencies for a 4 fold application of the basic C_7 and the post- C_7 for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

post- $C7$ sequence given a length of 4, 8, 12, and 16. There are two different views for each data set. The first is the 3D profile, which gives a better view of the form of transfer function, the second is the gradient–contour plot for numerical representations. Data for spin system SS_1 are shown in Figures 6.29 and 6.30. Data for spin system SS_2 are shown in Figures 6.31 and 6.32, and data for spin system SS_3 are shown in Figures 6.33 and 6.34.

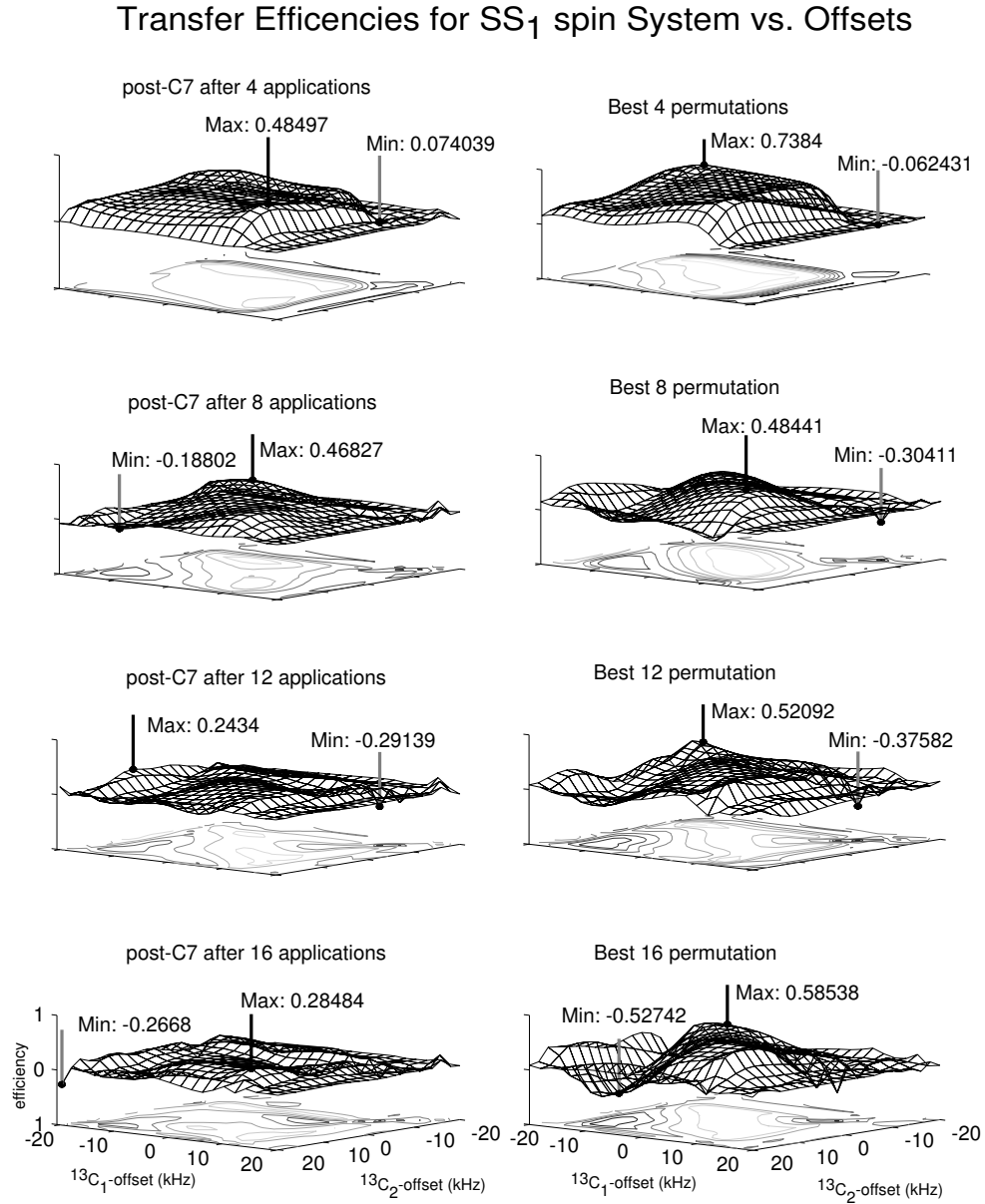


Figure 6.29: 3D transfer efficiencies plots for a 4,8,12,16 fold application of the post- C_7 and the best permutation cycles for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

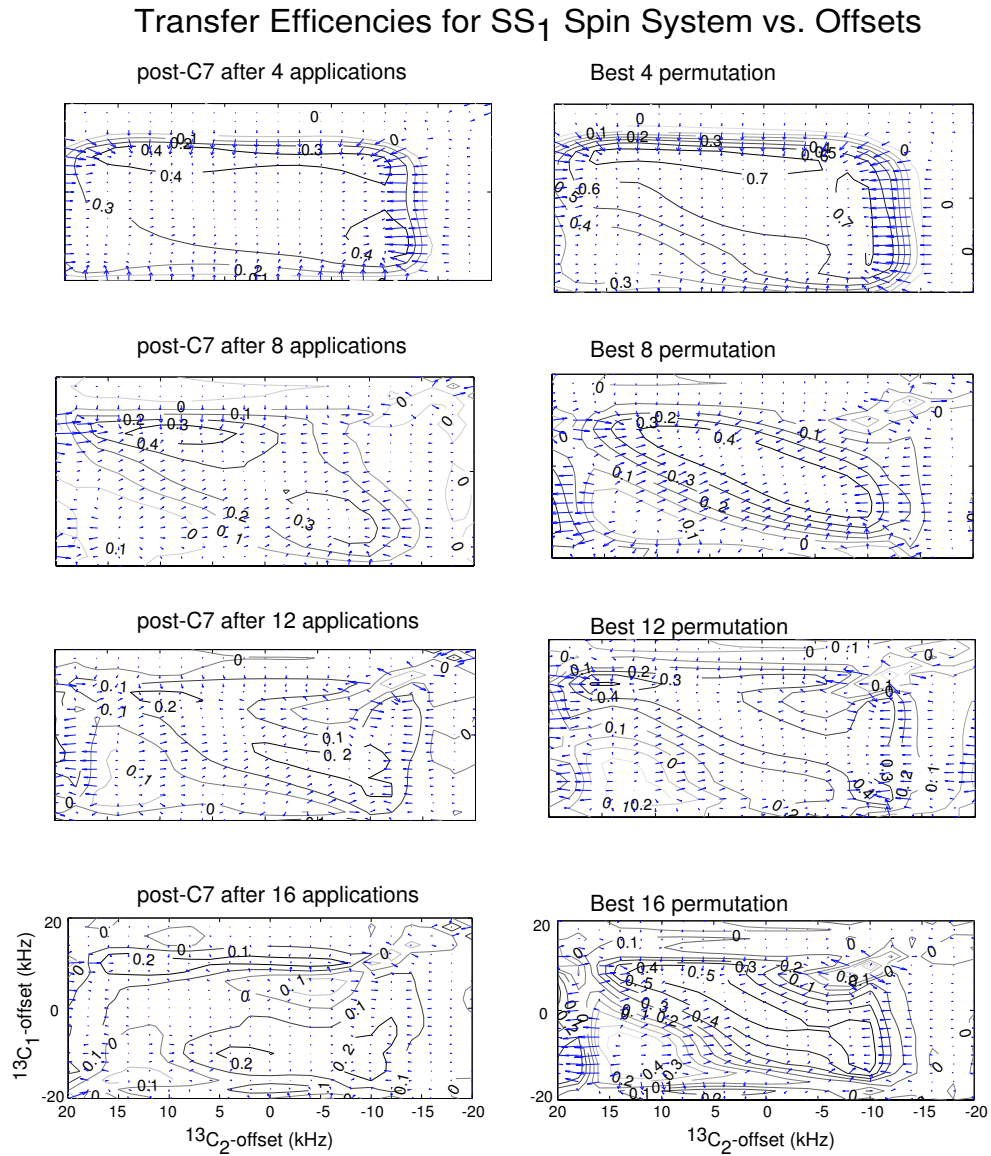


Figure 6.30: Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- C_7 and the best permutation cycles for the SS_1 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

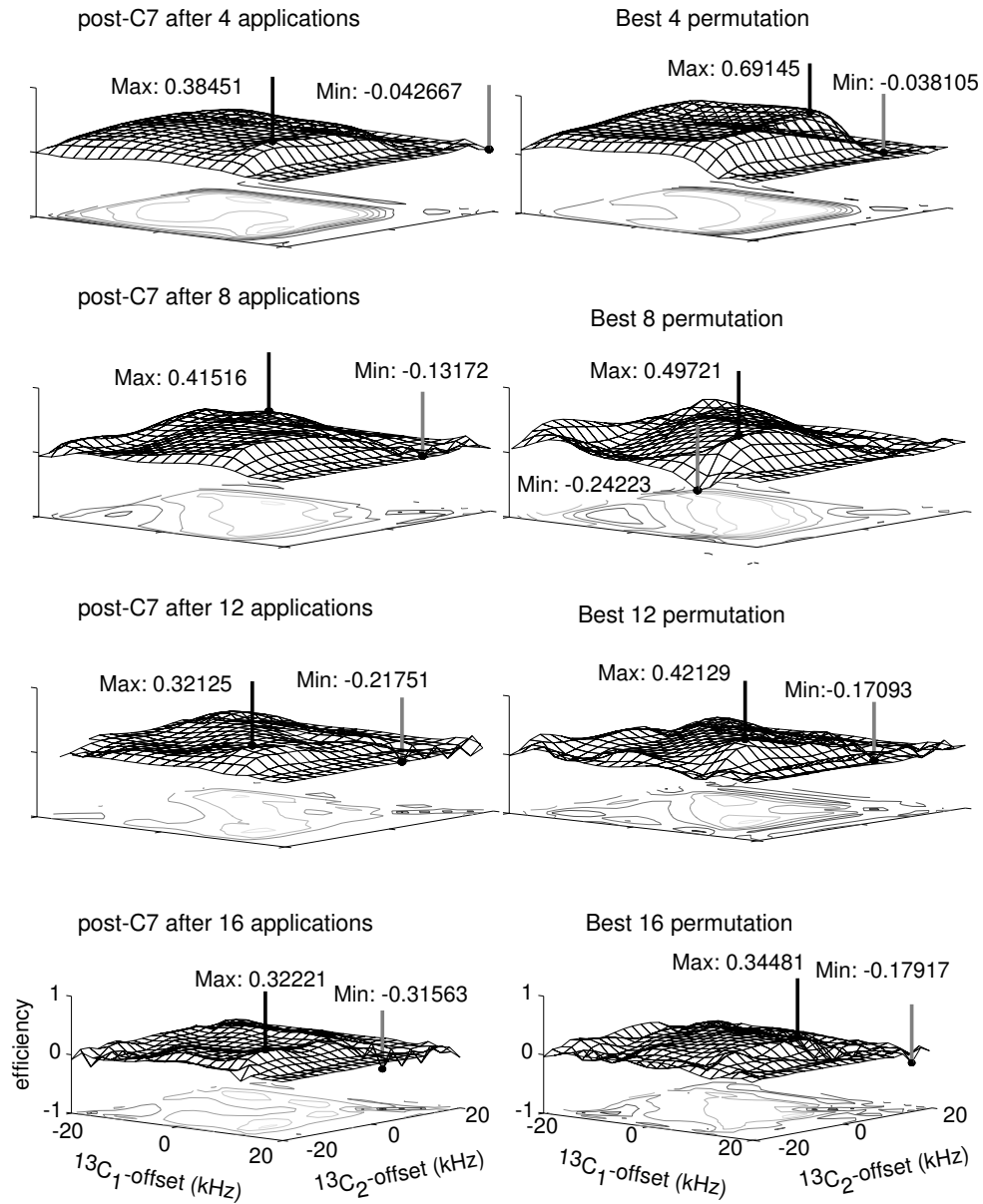
Transfer Efficiencies for SS_2 spin System vs. Offsets

Figure 6.31: 3D transfer efficiencies plots for a 4,8,12,16 fold application of the post- $C7$ and the best permutation cycles for the SS_2 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

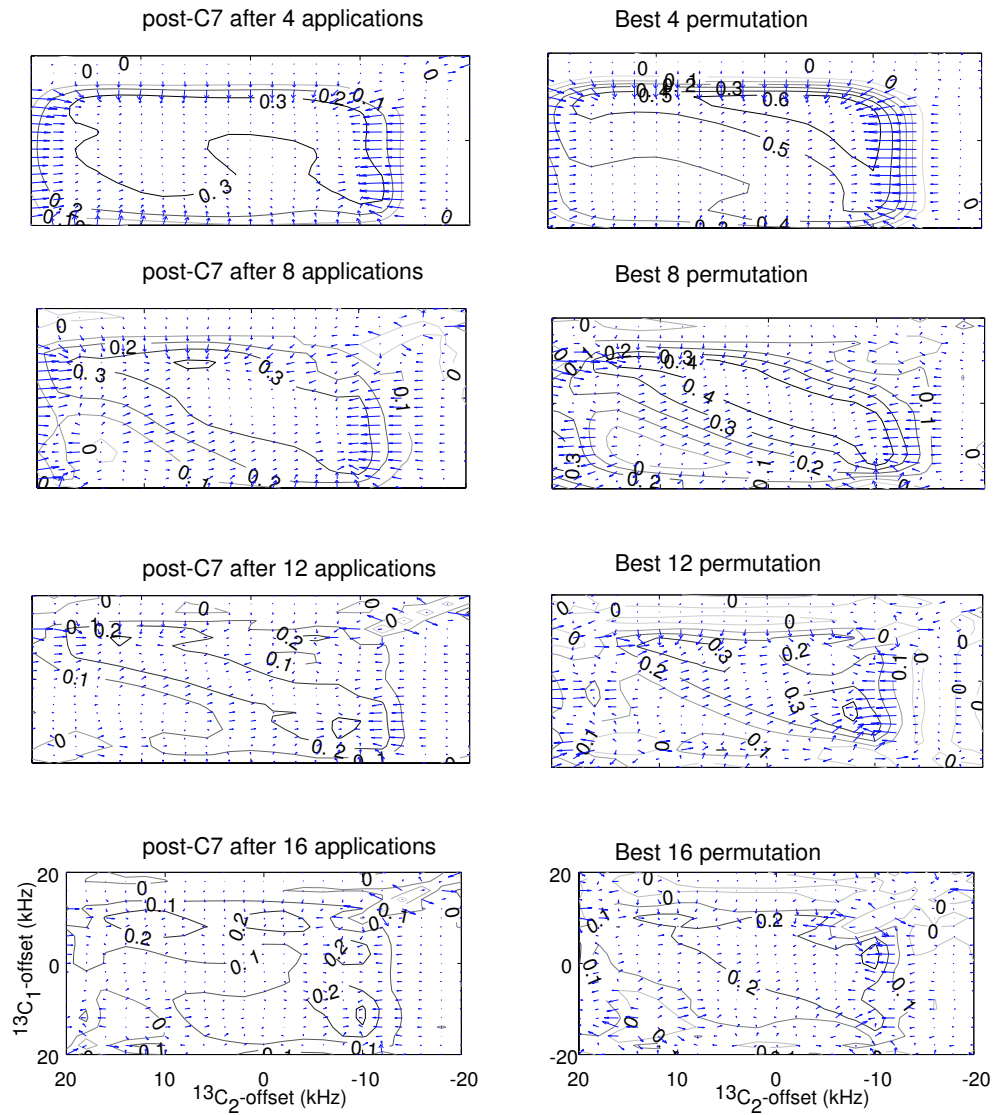
Transfer Efficiencies for SS_2 Spin System vs. Offsets

Figure 6.32: Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- C_7 and the best permutation cycles for the SS_2 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

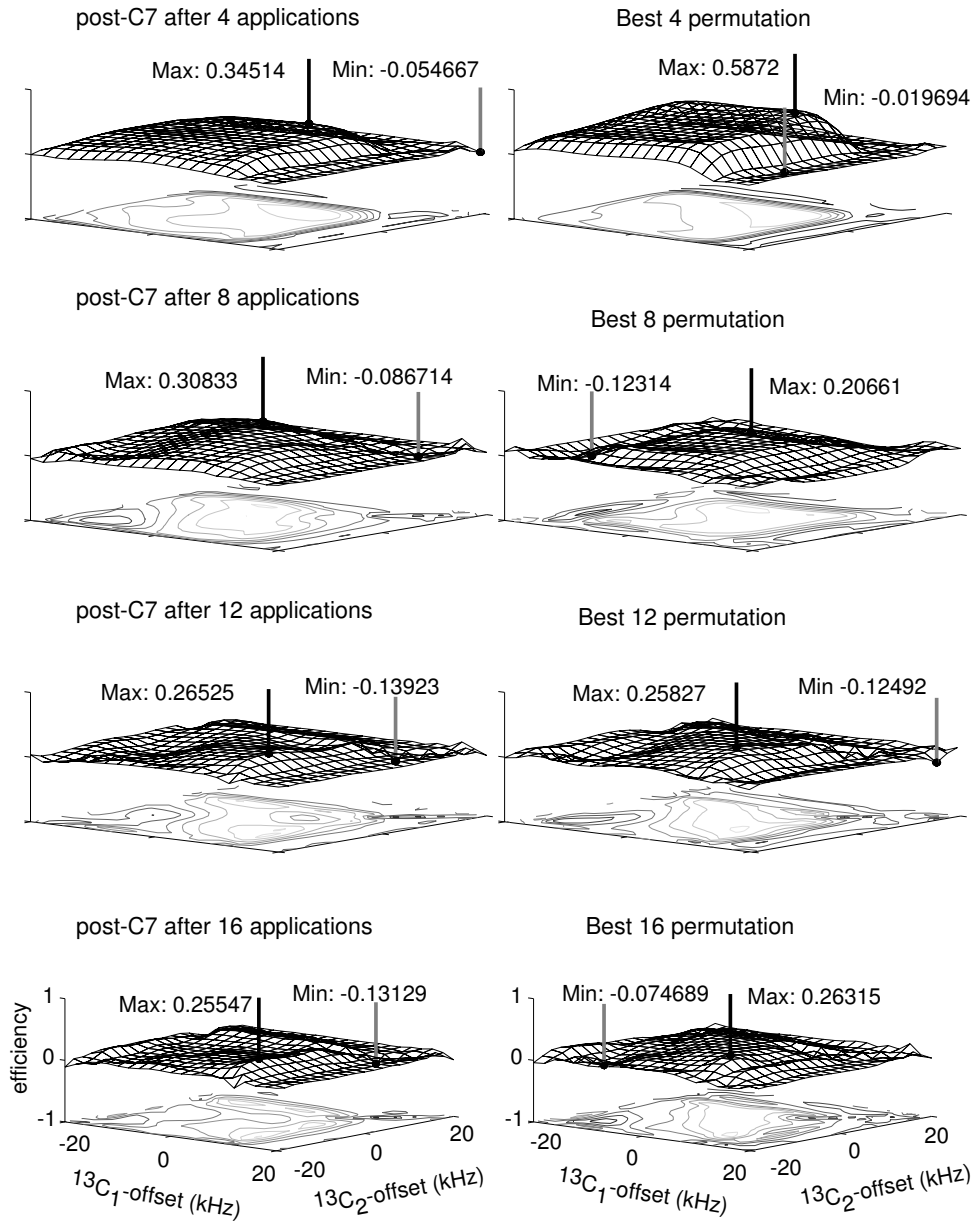
Transfer Efficiencies for SS_3 Spin System vs. Offsets

Figure 6.33: 3D transfer efficiencies plots for a 4,8,12,16 fold application of the post-C7 and the best permutation cycles for the SS_3 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5\text{kHz}$.

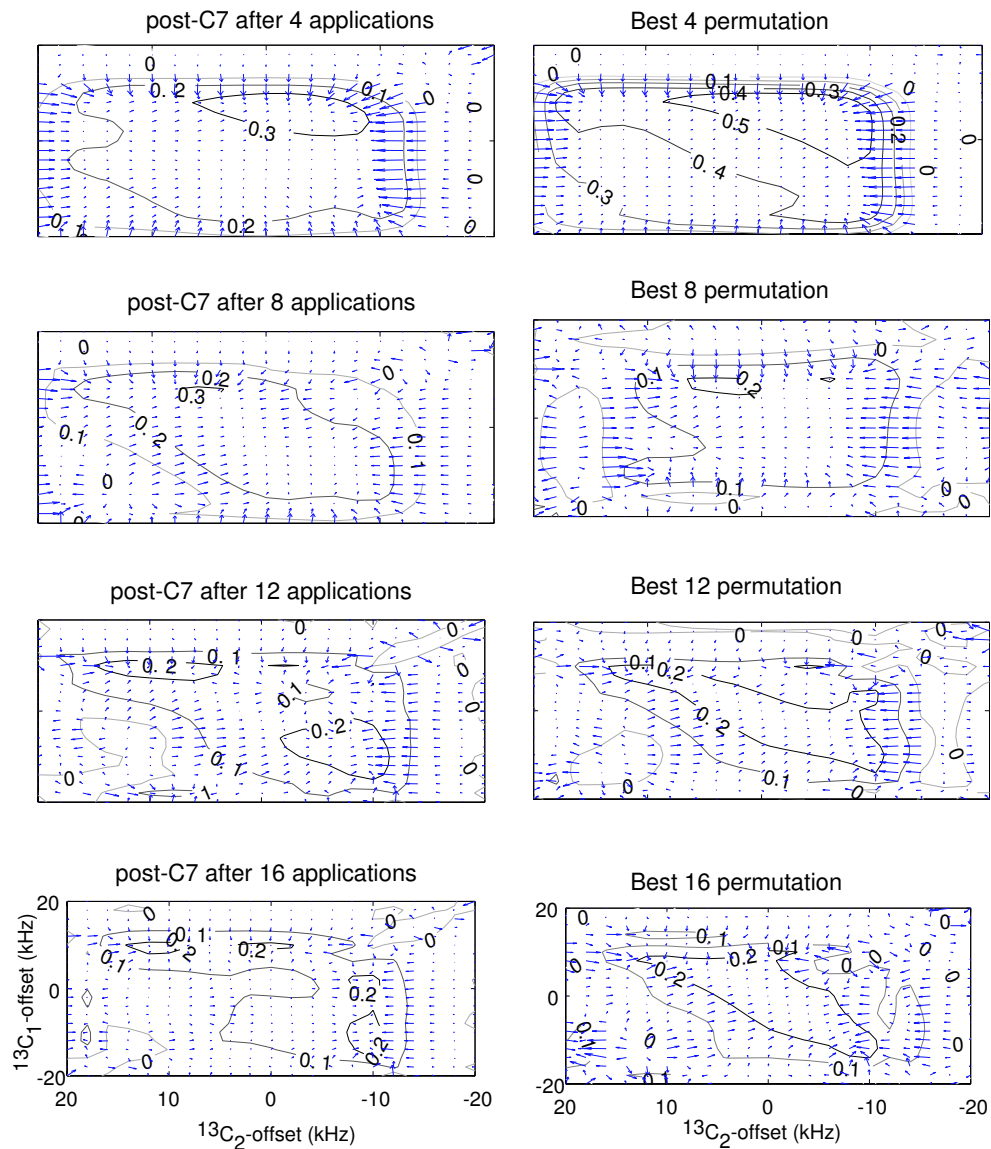
Transfer Efficiencies for SS_3 Spin System vs. Offsets

Figure 6.34: Contour–gradient transfer efficiencies plots for a 4,8,12,16 fold application of the post- C_7 and the best permutation cycles for the SS_3 system as a function of $^{13}C_1$ and $^{13}C_2$ offsets at $\omega_r = 5kHz$.

As the Figures clearly show, the permuted sequences are always better than the standard post-C7 sequences. We can generate similar views as in Figure 6.4 by taking slices along each of the above figures using each the best permutation cycle. These complete transfer diagrams are shown in Figures 6.35-6.37 for systems SS_1 , SS_2 and SS_3 respectively. The resulting transfers are on average 50% better in efficiency transfer and 25% more stable (the standard deviation across specific offset value) than the original sequence.

6.5 Conclusions

RSS sequences represent a large class of the pulse sequences used in solid-state NMR. They rely on generation of a specific zeroth order average Hamiltonian. However, in real systems the desired effect this zeroth order average Hamiltonian is destroyed by many experimental and system specific parameters. To correct for these problems, the symmetry of the system and the zeroth order average Hamiltonians are used to cancel other terms in the expansion. The implementation of symmetry is usually broken into two parts. Internal compensation and posting techniques as designed to act on a small part of the total sequence. Super-cycling takes these internally compensated sequences through application of phase shifts, attempts to compensate for errors in the total sequence. This process, for small number of sequence applications tends to work very well to compensate for errors. As the sequence becomes longer and longer, this simple approach breaks down as higher and higher order terms and errors accumulate. Determination of the best super-cycle for these longer sequences becomes a tedious task as many orders of the average Hamiltonian must be calculated and analyzed for weaknesses, a task for the general spin system is nearly impossible analytically. We can, however, use a permutation approach to the problem as

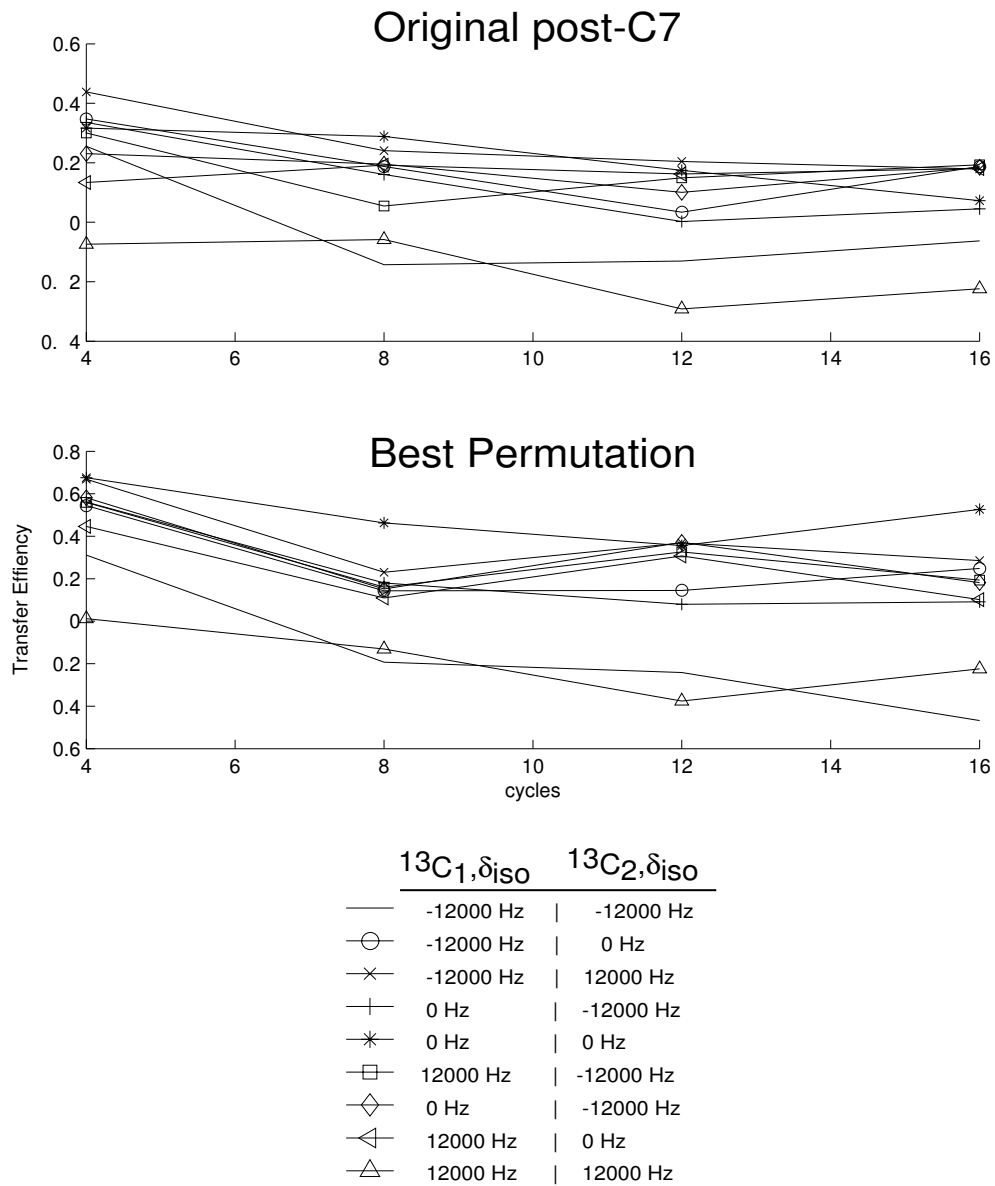


Figure 6.35: Transfer Efficiencies using the post-C7 and the best permuted cycles across over different cycles for the SS_1 spin system.

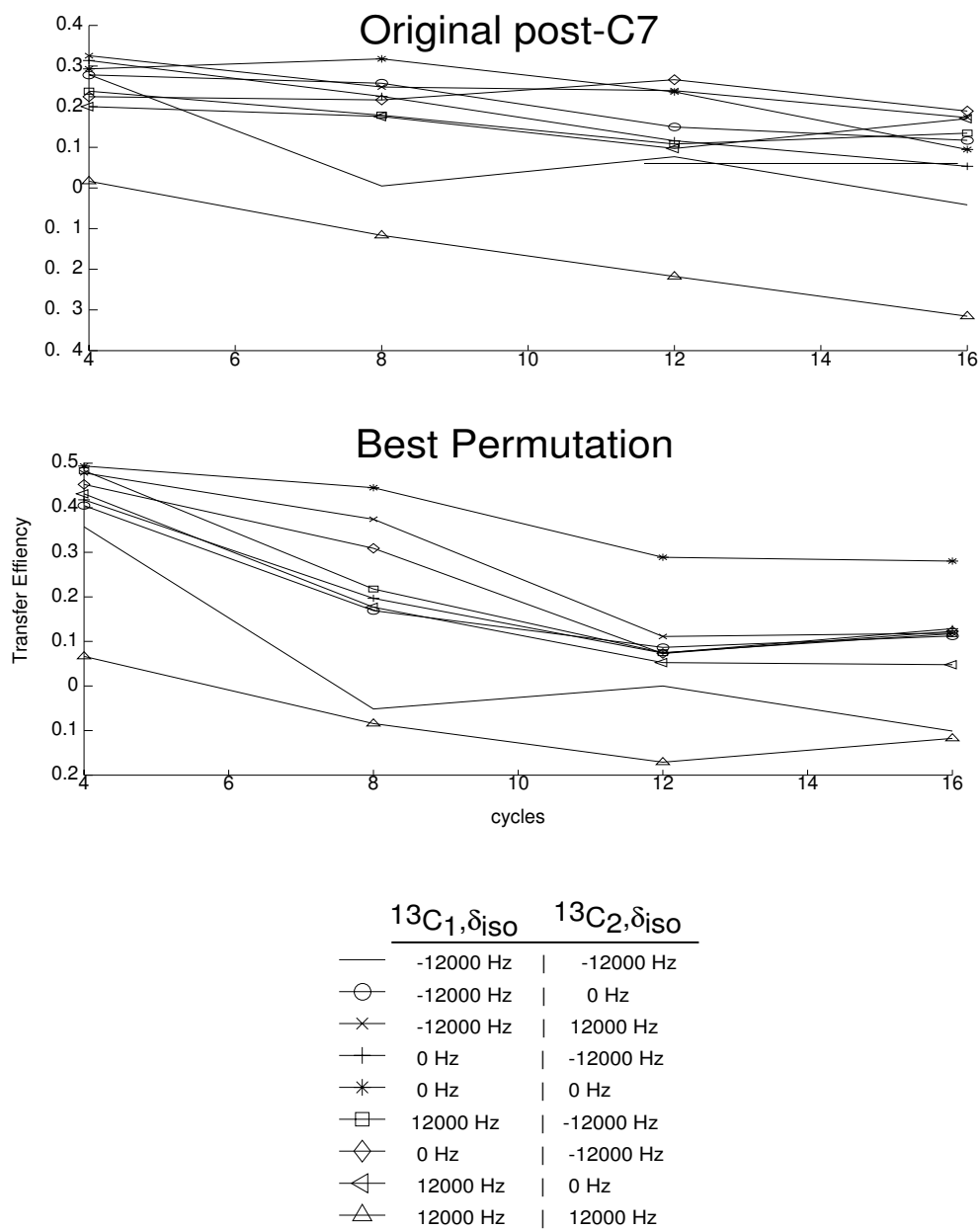


Figure 6.36: Transfer efficiencies using the post- $C7$ and the best permuted cycles across over different cycles for the SS_2 spin system.

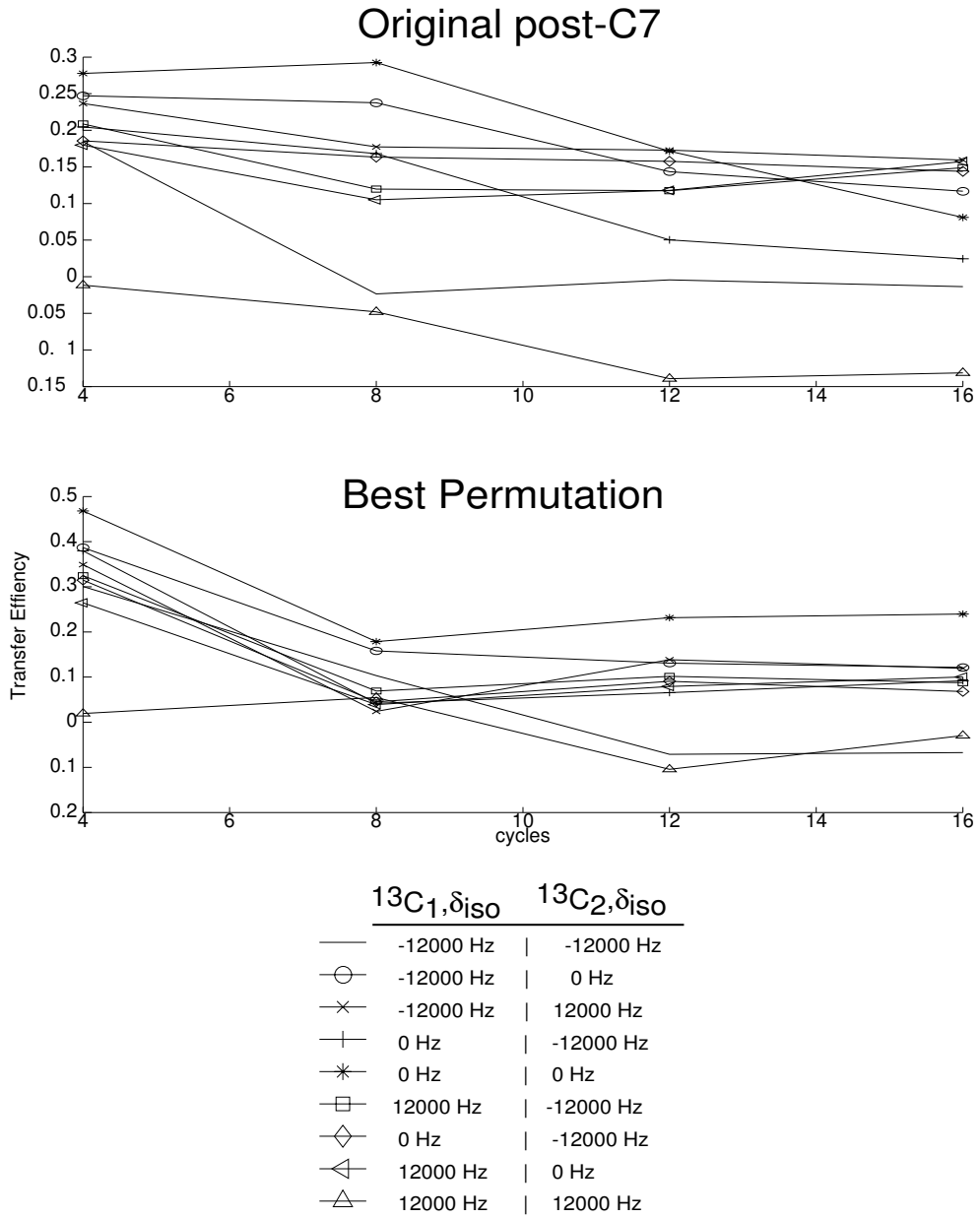


Figure 6.37: Transfer efficiencies using the post-C7 and the best permuted cycles across over different cycles for the SS_3 spin system.

we have shown here to generate markedly improved sequences using only the well known symmetry principles used for the shorter sequences.

Chapter 7

Future Expansions

The permutation technique described in chapter 6 is limited only by computer power and memory. For the study of the post-C7 the total simulation time for all of the permutations in all of the spin systems took about 3 weeks running on 4 different processors. If we had not optimized each step as we have done in chapters 4 and 2 this problem would have taken months to accomplish.

One may ask themselves why every permutation had to be calculated? Why not perform a minimization (or maximization in this case)? There are two fundamental problems associated with using normal minimization methods. The first is the vast dimensionality of the system. Using gradient or simulate annealing methods would almost certainly find local minima and not global minima unless it can sample most of the space. The second problem is that there is little information about the functional form of this ‘minimization’ function. We cannot say it is continuous or single valued making gradient searching techniques inaccurate and not robust. We are still not even sure if the function at point a parameter point A has any method of getting us to $A + 1$ without a look-up

table approach. If it is a look-up table, then running all the permutations seems to be the only way, unless we leave gradient/distance based minimization techniques. It is for these reasons that these techniques failed to produce any reasonable answers, and why the permutation approach seems to be the best alternative at the time.

There are techniques for both search very large dimensional spaces as well as the ‘look-up’ table problem and both are easily tackled using Evolutionary type Algorithms (EA)[137, 138], or Neural Networks (NN), or even both. I will discuss both in turn a give the basic structure for implementing both of these structures and the problems that should be able to be tackled using them.

7.1 Evolutionary Algorithms (EA)

The basics of an EA begin with a ‘gene.’ A gene in the biological sense propagates forward by creating children. These children contain some mixture of both parents due to *crossover/breeding* and *mutations*. A gene is then only likely to survive if it has a suitable mixture of the good qualities from the parents. It will die off if the child has inherited most of the bad ones. The relevance of ‘good’ and ‘bad’ from an algorithmic point of view is given as follows: a ‘good’ gene is one which has a fitness better or close to the parents value, a ‘bad’ gene has a worse fitness then the parents. The fitness can simply be an evaluation of the function giving a χ^2 or distance value or in our case the M_R value. We wish to find the best gene/phenotype we can.

There two different classes of EA using the blending mechanisms. If the blending mechanism includes both crossover and mutation it is called a Genetic Algorithm (GA)[139, 140, 141], if it only uses mutation (i.e. the children have no ‘parents’, just mutated versions

of itself) then this is call Evolutionary Programming (EP)[142, 143]. GAs use both a type of forced evolution from the parents and a pure evolution from mutation, where as EPs use only the pure evolution. A sub class of GAs is the Differential Evolution (DE)[144] where a child can have more then one parent. However the blending is performed, a strategy is typically devised for moving one step in a generation. These are usually called p-c Evolutionary strategies[145, 146], $ES_{(p,c)}$. Here p is the number of parents, and c is the number of children to be generated. These ESs are best shown pictorially as shown in Figure 7.1 along with a sub class of ESs, the ‘plus’ and ‘comma’ methods.

Which ever strategy is used we need to make our problem and the parameter space fit into a gene. There are many different ways of implementing such a gene using the RSS structure based on the number of assumption we can/wish to make. Using the methodology we discussed in chapter 6 our gene is of the length of the total number of cycles in we wish to optimize. A ‘base-pair’ is then one of the 4 possible symmetry cycles (o, O, w, W). Two arbitrary parent genes and a resulting child gene is shown in Figure 7.2. Initially we would generate a random set of these genes for the parents and evaluate the fitness for each of them (here it would be M_R) . In fact it would be better to pick genes that span the extremes of the parameter space. This means including genes with all of one type of base-pair (i.e. ($o,o,o,o,o\dots$), (W,W,W,W,W,W,\dots), (w,w,w,w,w,w,\dots), (O,O,O,O,O,\dots)). If one does not span the entire range initially, it is likely that the minimization will remain in the local range initially generated, or that it will take many iterations to get out of the local area.

The next step is where a GA or DE or EP algorithmic decision comes into play as well as the rates of mutation and cross over points. A diagram showing the plus and comma type of $ES_{(2,1)}$ for an EP algorithm is shown in Figure 7.3. A diagram showing the

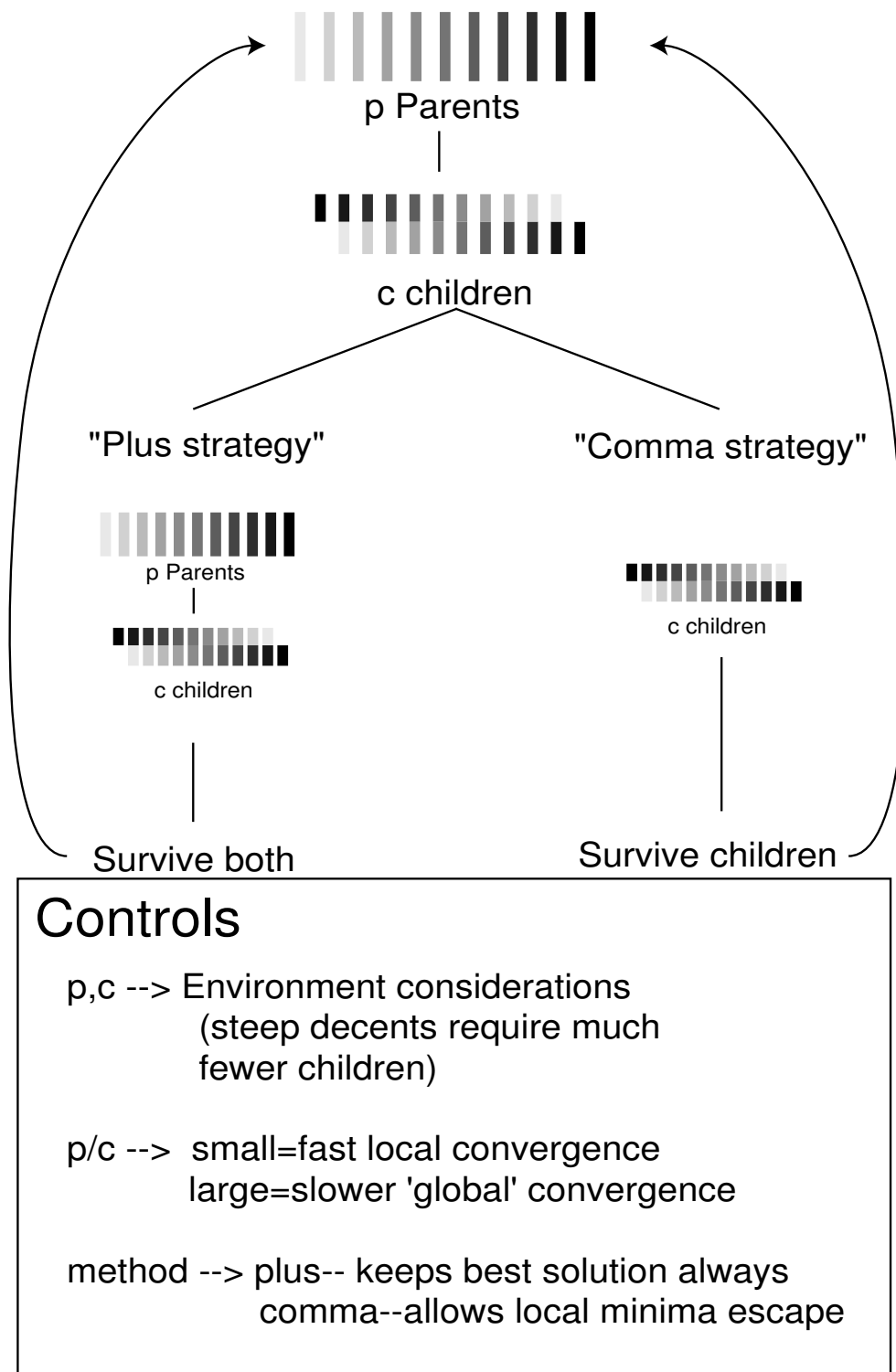


Figure 7.1: The standard evolutionary strategy methods and controls.

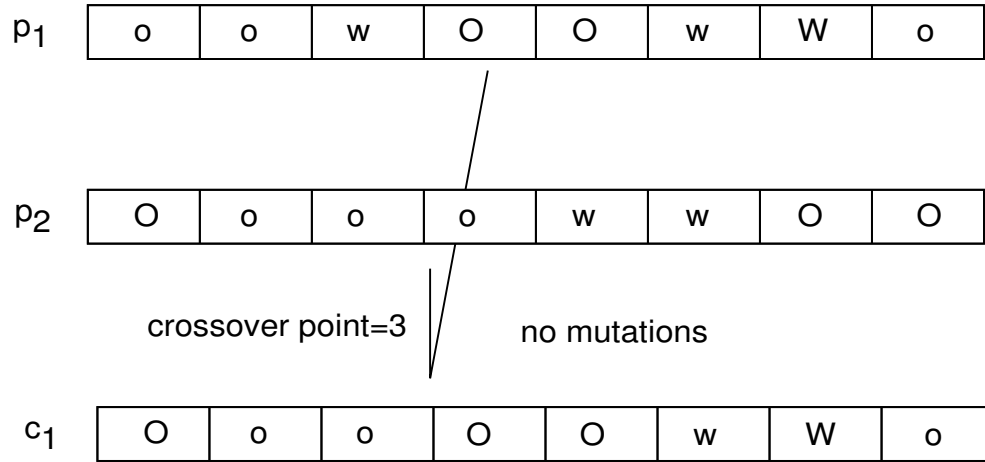


Figure 7.2: An arbitrary permutation cycle parent genes and resulting child.

plus and comma type of $ES_{(3,2)}$ for a GA algorithm is shown in Figure 7.4, and a diagram showing the plus and comma type of $ES_{(3,1)}$ for a DE algorithm is shown in Figure 7.5. DE methods are particular suited for continuous parameter space problems where the ‘add’ function in Figure 7.5 makes some sort of sense. Both EP and GA methods are better suited for the ‘4-switch’, (w,W,o,O) , type of values we want to search for.

In the permutations examples, we have limited ourselves to the basic symmetry 4-switch, we should be able to use the DE types for more complex searches. For instance we can arbitrarily change the phase of each RSS cycle and find a minimum. This phase changing should provide even better cycles than the permutation method as hinted by the super-cycle of Ref. [120]. This leads us to another branch of searches where we may begin to find different ‘post’ methods for an internal compensation of sequences. This type of internal search has been performed before, however, using the gradient techniques and for a much less general problem[147].

Evolutionary Programming

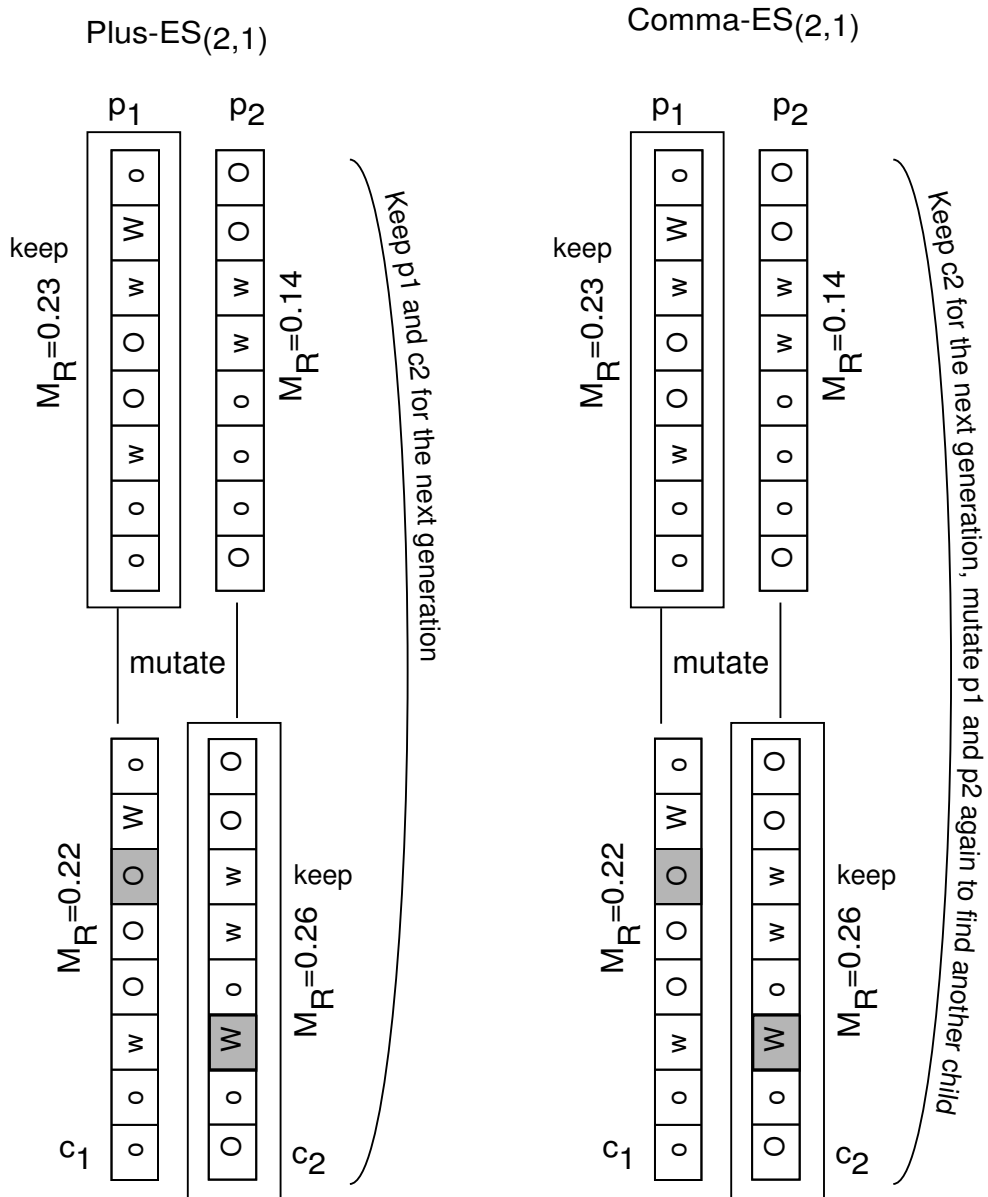
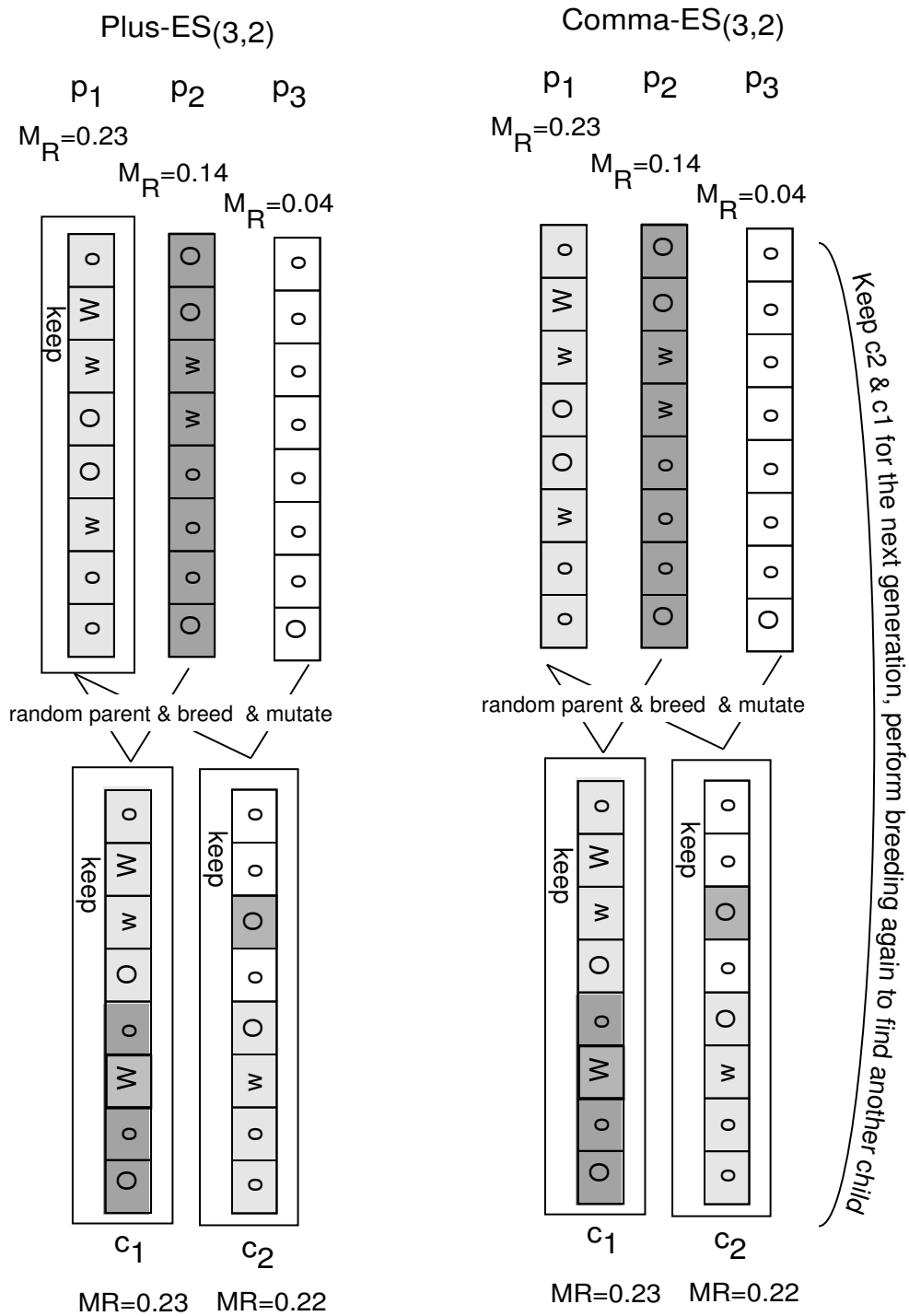
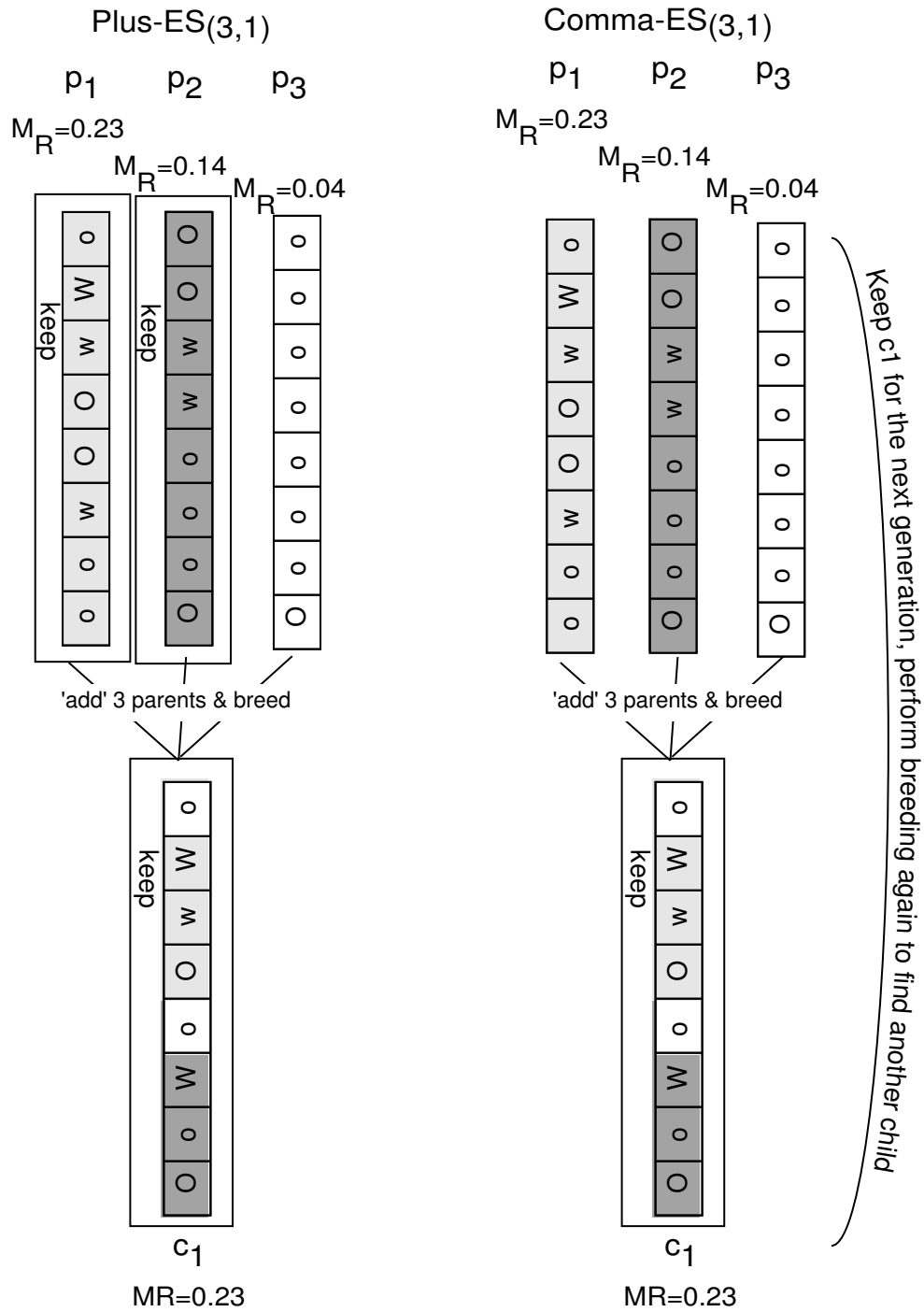


Figure 7.3: Evolution Programming (EP) generation step for an $ES_{(2,1)}$ strategy.

Genetic Algorithm

Figure 7.4: Genetic Algorithm (GA) generation step for an $ES_{(3,2)}$ strategy.

Differential Evolution

Figure 7.5: Differential Evolution (DE) generation step for an $ES_{(3,1)}$ strategy.

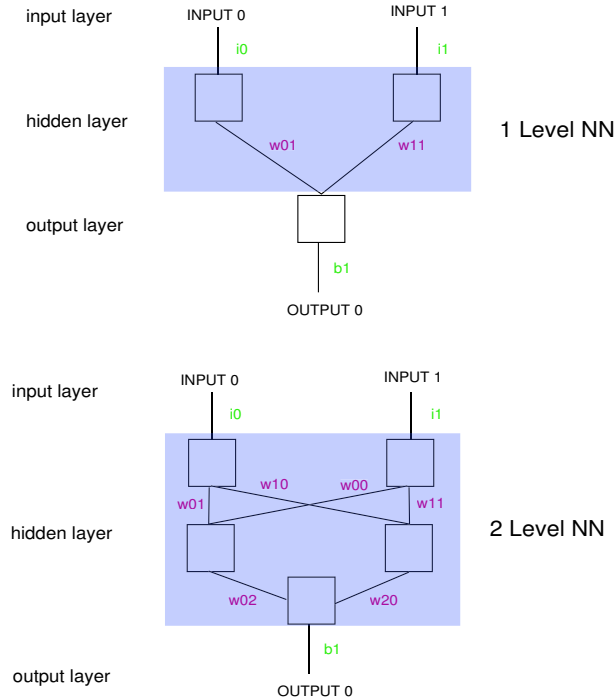


Figure 7.6: Basic 1 and 2 layer feed–forward neural networks.

7.2 Neural Networks

There is an astounding amount of literature on neural networks, much of it organized towards programmers[148, 149, 150]. To go through all but the most basic of neural networks here, would be much too much, so I will only attempt to scratch the surface of their capability. A neural network is simply a number of nodes connected by weights to other nodes based on neurons in a brain. It is designed to recreate a function when the function is unknown. Like a brain, in order to make predictions it must be trained. The training process is the hardest part of designing a NN and is crucial to the prediction power of a network[151]. For a network to begin to model a function, it must be trained with known inputs and outputs, then it should be able to give the correct outputs based on arbitrary inputs for a particular model. Figure 7.6 represents a subclass of NNs used for predictive purposes, a one and two level feed–forward (FF) NN (there are others, such as self organizing

networks). The ‘feed-forward’ implies that the input layer, I , effects the next layer(s) (called the hidden layers) and these effect the output layers. Each node, i , in a FFNN uses the weights, w_{ij} of the previous layer passed through a ‘relevance’ or threshold (R) function which determines its value, N_i .

$$N_i = R \left[\sum_{\text{connected nodes}} w_{ij} I_j + b_i \right] \quad (7.1)$$

The relevance function is used to model a typical neuron electronic switch where if the electric potential is high enough, it will pass on the signal, if it is not, the signal will halt there. The relevance function is usually a sigmoid function ($1/(1 + \exp(-A))$) or a simple step function. The sigmoid allows for a small range of valid signals to pass, where as the step function does not. There is also a bias value b_i applied to each value.

To train a network one typically picks a fixed number of hidden layers, and then manipulates the weights w_{ij} and the connectivity of the nodes. This is where the majority of the NN literature is based as in essence we wish to choose only the relevant data[152] and determine the connectivity and weights as a minimization process. One can even use evolutionary techniques to perform the minimization[153]. The simplest technique is back propagation. This minimization process does not try to determine any relevance or node removal; it simply uses the differences in the desired values and the current output values to adjust the weights and biases. Given a training data set, we can specify a ‘learning-rate’ (a value from 0..1) that determines the amount to adjust the weights given the distances. If we choose a learning rate of 1 then the network will quickly adjust the weights to match that one data set, if it is 0 then no adjustments occur with the differences. A simple back propagation fully connected FFNN C++ class is given in Appendix A.1.5.

So what could a NN provide in our optimization of the generic RSS (and potentially

other) sequences? As you can see the amount of data generated from the permutation study is quite large. We now have a huge training data set. In essence one could train a NN given the permutation order as the output, and the tensor coefficients as the input. As we generate more and more data we could even output a phases of an RSS sub-cycle from the tensor coefficients. After the training, we should be able to see two things. The first, and most obvious, is attempting to input our maximum $T_{2,\pm 2}$ condition and see what the network produces as an answer. The second, and I think more interesting, is the information that can be obtained from the relevant weights and connectivity's. From these particular values, it could be possible to find the most relevant pathways, which we could then infer possible symmetry classes to the general analytical problem. If nothing else, a properly trained NN can at least give us the good answers from a sequence abstraction without having to go through each permutation or phase shift, only a much smaller subset of them.

7.3 Final Remarks

The majority of this thesis is geared to the creation of fast numerical techniques and algorithms to simulate NMR situations as fast as one can. We can now tackle problems that were before next to impossible before this assembly. The genetic and neural networks applications are some of the more interesting paths to follow as their implementation is now some what possible given the processes shown here. There results, however, are unknown. For all I know, these newer algorithmic techniques may not give much new insight to the basic problems and control of NMR. However, simply looking at the statistical distributions of one RSS sequence has demonstrated that the solution NMR seeks is usually the far outlier (our evolutionary searches), while most of the data fits into a nice distribution (the neural

networks). Optimal control of a given NMR system may soon be reduced to a trained neural network producing first order results (pulse sequences), while the evolutionary techniques find the optimum.

Bibliography

- [1] E. M. Purcell, H. C. Torrey, and R. V. Pound, *Phys. Rev.* **69** (1-2), 37 (1946).
- [2] F. Bloch, W. W. Hansen, and P. M., *Phys. Rev.* **70** (7-8), 474 (1946).
- [3] A. Abragam, *The Principles of Nuclear Magnetism: The International Series of Monographs on Physics* (Clarendon Press, Oxford, 1961).
- [4] M. H. Levitt, *Spin Dynamics: Basics of Nuclear Magnetic Resonance* (John Wiley & Sons, ltd., Chichester, 2000).
- [5] R. R. Ernst, G. Bodenhausen, and A. Wokaun, *Principles of Nuclear Magnetic Resonance in One and Two Dimensions* (Clarendon Press, Oxford, 1989).
- [6] C. P. Slichter, *Principles of Magnetic Resonance* (Springer, Heidelberg, 1978).
- [7] A. Turing, in *Proceedings of the London Mathematical Society, Series 2* (Oxford University Press, Oxford, 1936), Vol. 42.
- [8] R. W. Sebesta, *Concepts of Programming Languages 5/E* (Addison Wesley Higher Education, Boston, MA, 2001).
- [9] B. Stroustrup, *The design and evolution of C++* (Addison-Wesley, Boston, MA, 1994).

-
- [10] B. Stroustrup, *The C++ Programming Language. third ed.* (Addison-Wesley, Boston, MA, 1997).
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Boston, MA, 1995).
- [12] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen., *LAPACK Users Guide, third ed.* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999).
- [13] T. L. Veldhuizen and M. E. Jernigan, in *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97), Lecture Notes in Computer Science* (Springer-Verlag, New York, 1997).
- [14] T. Veldhuizen, C++ Report **7(5)**, 26 (1995).
- [15] E. Unruh, 1994, aNSI X3J16-94-0075/ISO WG21-462.
- [16] U. W. Eisenecker and K. Czarnecki, *Generative Programming - Towards a New Paradigm of Software Engineering* (Addison Wesley, Boston, MA, 2001).
- [17] C. Pescio, C++ Report **9(7)**, (1997).
- [18] N. C. Myers, C++ Report **7(5)**, 42 (1995).
- [19] T. Veldhuizen, C++ Report **7(4)**, 36 (1995).
- [20] J. G. Siek, Master's thesis, University of Notre Dame, 1994.
- [21] W. Clint, automatically Tuned Linear Algebra Software (ATLAS).

- [22] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design* (Morgan Kaufmann Publishers Inc., San Francisco, Ca, 1998).
- [23] F. Bloch, Phys. Rev. **70** (7-8), 460 (1946).
- [24] M. Mehring and V. A. Weberruss, *Object-Oriented Magnetic Resonance* (Academic Press, London, UK, 2001).
- [25] A. Vlassenbroek, J. Jeener, and P. Broekaert, J. Mag. Reson. A **118**, 234 (1996).
- [26] J. Jeener, A. Vlassenbroek, and P. Broekaert, J. Chem. Phys. **103**(9), 1309 (1995).
- [27] G. Deville, M. Bernier, and J. M. Delrieux, Phys. Rev. B **19**(11), 5666 (1979).
- [28] T. Enss, S. Ahn, and W. S. Warren, Chem. Phys. Lett. **305**, 101 (1999).
- [29] W. S. Warren, S. Lee, W. Richter, and S. Vathyan, Chem. Phys. Lett. **247**, 207 (1995).
- [30] R. N. Zare, *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics* (John Wiley & Sons, Inc., Chichester, 1988).
- [31] S. Wi and L. Frydman, J. Chem. Phys. **112**(7), 3248 (2000).
- [32] W. Warren, W. Richter, A. Andreotti, and B. Farmer, Science **262** (5142), 2005 (1993).
- [33] W. Richter and W. Warren, Conc. Mag. Reson. **12**(6), 396 (2000).
- [34] S. Lee, W. Richter, S. Vathyam, and W. S. Warren, J. Chem. Phys. **105**(3), 874 (1996).
- [35] W. S. Warren, S. Y. Huang, S. Ahn, and Y. Y. Lin, J. Chem. Phys. **116**(5), 2075 (2002).

-
- [36] Q. H. He, W. Richter, S. Vathyam, and W. S. Warren, *J. Chem. Phys.* **98**(9), 6779 (1993).
- [37] R. R. Rizzi, S. Ahn, D. C. Alsop, S. Garrett-Roe, M. Mescher, W. Richter, M. D. Schnall, J. S. Leigh, and W. S. Warren, *Mag. Reson. Med.* **18**, 627 (2000).
- [38] W. Richter, M. Richtera, W. S. Warren, H. Merkle, P. Andersen, G. Adriany, and K. Ugurbil, *Mag. Reson. Img.* **18**, 489 (2000).
- [39] W. Richter, S. Lee, W. Warren, and Q. He, *Science* **267** (5198), 654 (1995).
- [40] J. H. V. Vleck, *Electric and Magnetic Susceptibilities* (Oxford University Press, Great Britan, 1932).
- [41] P. Deuffhard, *Numerische Mathematik* **41**, 399 (1983).
- [42] J. R. Cash and A. H. Karp, *ACM Transactions on Mathematical Software* **16**, 201 (1990).
- [43] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis* (Springer-Verlag, New York, 1980).
- [44] P. Deuffhard, *SIAM Rev.* **27**, 505 (1985).
- [45] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing* (Cambridge University Press, Cambridge, 1997).
- [46] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice–Hall, Englewood Cliffs, NJ, 1971).

- [47] J. H. Shirley, Phys. Rev. B. **138**, 979 (1965).
- [48] A. Schmidt and S. Vega, J. Chem. Phys. **96** (4), 2655 (1992).
- [49] R. Challoner and M. CA, J. Mag. Reson. **98** (1), 123 (1992).
- [50] O. Weintraub and S. Vega, J. Mag. Reson. Ser. A. **105** (3), 245 (1993).
- [51] T. Levante, B. H. Baldus, M. and Meier, and R. Ernst, Mol. Phys. **86** (5), 1195 (1995).
- [52] J. W. Logan, J. T. Urban, J. D. Walls, K. H. Lim, A. Jerschow, and A. Pines, Solid State NMR **22**, 97 (2002).
- [53] J. Walls, K. Lim, J. Logan, J. Urban, A. Jerschow, and A. Pines, J. Chem. Phys **117**, 518 (2002).
- [54] J. Walls, K. Lim, and A. Pines, J. Chem. Phys. **116**, 79 (2002).
- [55] M. H. Levitt, D. P. Raleigh, F. Cruzet, and R. G. Griffin, J. Chem. Phys. **92**(11), 6347 (1990).
- [56] D. P. Raleigh, M. H. Levitt, and R. G. Griffin, Chem. Phys. Lett. **146**, 71 (1988).
- [57] M. G. Colombo, B. H. Meier, and R. R. Ernst, Chem. Phys. Lett. **146**, 189 (1988).
- [58] Y. Zur and M. H. Levitt, J. Chem. Phys. **78**(9), 5293 (1983).
- [59] M. Eden, Y. K. Lee, and M. H. Levitt, J. Magn. Reson. A. **120**, 56 (1996).
- [60] M. Hohwy, H. Bildse, and N. C. Nielsen, J. Magn. Reson. **136**, 6 (1999).
- [61] T. Charpentier, C. Fermon, and J. Virlet, J. Magn. Reson. **132**, 181 (1998).
- [62] M. H. Levitt and M. Eden, Mol. Phys. **95**(5), 879 (1998).

- [63] H. Geen and r. Freeman, *J. Mag. Reson.* **93(1)**, 93 (1991).
- [64] P. Bilski, N. A. Sergeev, and J. Wasicki, *Solid State Nuc. Mag. Reson.* **22(1)**, 1 (2002).
- [65] A. Baram, *J. Phys. Chem.* **88(9)**, 1695 (1984).
- [66] M. Mortimer, G. Oates, and T. B. Smith, *Chem. Phys. Lett.* **115(3)**, 299 (1985).
- [67] A. Kumar and P. K. Madhu, *Conc. Mag. Reson.* **8(2)**, 139 (1996).
- [68] P. Hazendonk, A. D. Bain, H. Grondey, P. H. M. Harrison, and R. S. Dumont, *J. Mag. Reson.* **146**, 33 (2000).
- [69] M. Eden and M. H. Levitt, *J. Mag. Reson.* **132**, 220 (1998).
- [70] D. W. Alderman, M. S. Solum, and D. M. Grant, *J. Chem. Phys.* **84**, 3717 (1986).
- [71] M. J. Mombourquette and J. A. Weil, *J. Mag. Reson.* **99**, 37 (1992).
- [72] L. Andreozzi, M. Giordano, and D. Leporini, *J. Mag. Reson. A* **104**, 166 (1993).
- [73] D. Wang and G. R. Hanson, *J. Mag. Reson. A* **117**, 1 (1995).
- [74] S. J. Varner, R. L. Vold, and G. L. Hoatson, *J. Mag. Reson. A* **123**, 72 (1996).
- [75] M. Bak and N. C. Nielsen, *J. Mag. Reson.* **125**, 132 (1997).
- [76] S. K. Zaremba, *Ann. Mat. Pura. Appl.* **4:73**, 293 (1966).
- [77] J. M. Koons, E. Hughes, H. M. Cho, and P. D. Ellis, *J. Mag. Reson. A* **114**, 12 (1995).
- [78] L. Gonzalez-Tovany and V. Beltran-Lopez, *J. Mag. Reson.* **89**, 227 (1990).
- [79] C. V. B., H. H. Suzukawa, and M. Wolfsberg, *J. Chem. Phys.* **59(8)**, 3992 (1973).

- [80] H. Conroy, *J. Chem. Phys.* **47(2)**, 5307 (1967).
- [81] V. I. Lebedev, *Zh. Vychisl. Mat. Fiz.* **16**, 293 (1976).
- [82] V. I. Lebedev, *Zh. Vychisl. Mat. Fiz.* **15**, 48 (1975).
- [83] J. Dongarra, P. Kacsuk, and N. P. (eds.), *Recent advances in parallel virtual machine and message passing interface: 7th European PVM/MPI users group meeting* (Springer, Berlin, 2000).
- [84] P. Hodgkinson and L. Emsley, *Prog. Nucl. Magn. Reson. Spectrosc.* **36**, 201 (2000).
- [85] M. Bak, J. T. Rasmussen, and N. C. Nielsen, *J. Magn. Reson.* **147**, 296 (2000).
- [86] S. Smith, T. Levante, B. Meier, and R. Ernst, *J. Mag. Reson.* **106a**, 75 (1994).
- [87] Y. Y. Lin, N. Lisitza, S. D. Ahn, and W. S. Warren, *Science* **290 (5489)**, 118 (2000).
- [88] C. A. Meriles, D. Sakellariou, H. Heise, A. J. Moule, and A. Pines, *Science* **293**, 82 (2001).
- [89] H. Heise, D. Sakellariou, C. A. Meriles, A. Moule, and A. Pines, *J. Mag. Reson.* **156**, 146 (2002).
- [90] T. M. Brill, S. Ryu, R. Gaylor, J. Jundt, D. D. Griffin, Y. Q. Song, P. N. Sen, and M. D. Hurlimann, *Science* **297**, 369 (2002).
- [91] R. McDermott, A. H. Trabesinger, M. Muck, E. L. Hahn, A. Pines, and J. Clarke, *Science* **295**, 2247 (2002).
- [92] J. D. Walls, M. Marjanska, D. Sakellariou, F. Castiglione, and A. Pines, *Chem. Phys. Lett.* **357**, 241 (2002).

-
- [93] R. H. Havlin, G. Park, and A. Pines, *J. Mag. Reson.* **157**, 163 (2002).
- [94] M. Frigo and S. G. Johnson, Technical report, Massachusetts Institute of Technology (unpublished).
- [95] E. Lusk, Technical report, University of Tennessee (unpublished).
- [96] F. James, Technical report, Computing and Networks Division CERN Geneva, Switzerland (unpublished).
- [97] G. Bader and P. Deuffhard, *Numerische Mathematik* **41**, 373 (1983).
- [98] M. K. Stehling, R. Turner, and P. Mansfield, *Science* **254 (5028)**, 43 (1991).
- [99] M. Hohwy, H. J. Jakobsen, M. Eden, M. H. Levitt, and N. C. Nielsen, *J. Chem. Phys.* **108**, 2686 (1998).
- [100] M. P. Augustine and K. W. Zilm, *J. Mag. Reson. Ser. A.* **123**, 145 (1996).
- [101] M. K.T., B. Sun, G. Chinga, J. Zwanziger, T. Terao, and A. Pines, *J. Mag. Reson.* **86 (3)**, 470 (1990).
- [102] R. H. Havlin, T. Mazur, W. B. Blanton, and A. Pines, (2002), in preparation.
- [103] U. Haeberlen, *High Resolution NMR in Solids: Selective Averaging* (Academic Press, New York, 1976).
- [104] M. Mehring, *Principles of High Resolution NMR in Solids* (Springer, Berlin, 1983).
- [105] M. Eden and M. H. Levitta, *J. Chem. Phys.* **111(4)**, 1511 (1999).
- [106] P. Tekely, P. Palmas, and D. Canet, *J. Mag. Reson. A* **107(2)**, 129 (1994).

- [107] M. Ernst, S. Bush, A. Kolbert, and A. Pines, *J. Chem. Phys.* **105** (9), 3387 (1996).
- [108] A. E. Bennett, C. M. Rienstra, M. Auger, K. V. Lakshmi, and R. G. Griffin, *J. Chem. Phys.* **103** (16), 6951 (1995).
- [109] A. Bielecki, A. C. Kolbert, and M. H. Levitt, *Chem. Phys. Lett.* **155**(4-5), 341 (1989).
- [110] M. Carravetta, M. Eden, X. Zhao, A. Brinkmann, and M. H. Levitt, *Chem. Phys. Lett.* **321**, 205 (2000).
- [111] X. Zhao, M. Eden, and M. Levitt, *Chem. Phys. Lett.* **234**, 353 (2001).
- [112] A. Brinkmann, M. Eden, and M. H. Levitt, *J. Chem. Phys.* **112**(19), 8539 (2000).
- [113] J. Walls, W. B. Blanton, R. H. Havlin, and A. Pines, *Chem. Phys. Lett.* **363** (3-4), 372 (2002).
- [114] R. Tycko and G. Dabbagh, *Chem. Phys. Lett.* **173**, 461 (1990).
- [115] Y. K. Lee, N. D. Kurur, M. Helmle, O. G. Johannessen, N. C. Nielsen, and M. H. Levitt, *Chem. Phys. Lett.* **242**(3), 304 (1995).
- [116] W. Sommer, J. Gottwald, D. Demco, and H. Spiess, *J. Magn. Reson. A* **113**(1), 131 (1995).
- [117] C. M. Rienstra, M. E. Hatcher, L. J. Mueller, B. Q. Sun, S. W. Fesik, and R. G. Griffin, *J. Am. Chem. Soc.* **120**(41), 10602 (1998).
- [118] M. Hohwy, C. M. Rienstra, and R. G. Griffin, *J. Chem. Phys.* **117**(10), 4973 (2002).
- [119] M. Hohwy, C. M. Rienstra, C. P. Jaroniec, and R. G. Griffin, *J. Chem. Phys.* **110**(16), 7983 (1999).

- [120] A. Brinkmann and M. H. Levitt, *J. Chem. Phys.* **115**(1), 357 (2001).
- [121] A. Brinkmann, J. S. auf der Gnne, and M. H. Levitt, *J. Mag. Reson.* **156**(1), 79 (2002).
- [122] M. Hohwy, C. P. Jaroniec, B. Reif, C. M. Rienstra, and G. R. G., *J. Am. Chem. Soc.* **122**(13), 3218 (2000).
- [123] B. Reif, M. Hohwy, C. P. Jaroniec, C. M. Rienstra, and R. G. Griffin, *J. Mag. Reson.* **145**, 132 (2000).
- [124] M. H. Levitt, K. A. C., A. Bielecki, and D. J. Ruben, *Solid State Nucl. Magn. Reson.* **2**(4), 151 (1993).
- [125] Y. Yu and B. M. Fung, *J. Mag. Reson.* **130**, 317 (1998).
- [126] A. J. Shaka, J. Keeler, T. Frenkiel, and R. Freeman, *J. Mag. Reson.* **52**(2), 335 (1983).
- [127] A. J. Shaka, J. Keeler, and R. Freeman, *J. Mag. Reson.* **53**, 313 (1983).
- [128] A. J. Shaka and J. Keeler, *Prog. NMR Spectrosc.* **19**, 47 (1987).
- [129] M. H. Levitt, R. Freeman, and T. Frenkiel, *Adv. Mag. Reson.* **11**, 47 (1983).
- [130] M. H. Levitt, R. Freeman, and T. Frenkiel, *J. Mag. Reson.* **50**(1), 157 (1982).
- [131] M. H. Levitt, R. Freeman, and T. Frenkiel, *J. Mag. Reson.* **47**(2), 328 (1982).
- [132] M. H. Levitt and R. Freeman, *J. Mag. Reson.* **43**(3), 502 (1981).
- [133] W. S. Warren, J. B. Murdoch, and A. Pines, *J. Mag. Reson.* **60**(2), 236 (1984).
- [134] J. Murdoch, W. S. Warren, D. P. Weitekamp, and A. Pines, *J. Mag. Reson.* **60**(2), 205 (1984).

- [135] A. Bennett, Ph.D. thesis, Massachusetts Institute of Technology, Massachusetts Institute of Technology, 1995.
- [136] A. Bennett, C. Rienstra, J. Griffiths, W. Zhen, P. Lansbury, and R. Griffin, *J. Chem. Phys.* **108(22)**, 9463 (1998).
- [137] D. B. Fogel, in *Evolutionary Computation. The Fossil Record. Selected Readings on the History of Evolutionary Computation* (IEEE Press, Philadelphia, 1998), Chap. 16: Classifier Systems, this is a reprint of (Holland and Reitman, 1978), with an added introduction by Fogel.
- [138] W. M. Spears, K. A. D. Jong, T. Bäck, D. B. Fogel, and H. de Garis, in *Proceedings of the European Conference on Machine Learning (ECML-93)*, Vol. 667 of *LNAI*, edited by P. B. Brazdil (Springer Verlag, Vienna, Austria, 1993), pp. 442–459.
- [139] M. D. Vose, *Evolutionary Computation* **3**, 453 (1996).
- [140] M. D. Vose, in *Foundations of Genetic Algorithms 2*, edited by L. D. Whitley (Morgan Kaufmann, San Mateo, CA, 1993), pp. 63–73.
- [141] M. D. Vose, *The simple genetic algorithm: foundations and theory* (MIT Press, Cambridge, MA, 1999).
- [142] in *Evolutionary Programming – Proceedings of the Third International Conference*, edited by A. V. Sebald and L. J. Fogel (World Scientific Publishing, River Edge, NJ, 1994).
- [143] in *Proceedings of the 1995 IEEE International Conference on Evolutionary Computation*, edited by ??? (IEEE Press, Piscataway, 1995), Vol. 1.

-
- [144] R. Storn and K. Price, Technical report, International Computer Science Institute, UC Berkeley (unpublished).
- [145] D. B. Fogel, in *Evolutionary Algorithms*, edited by L. D. Davis, K. De Jong, M. D. Vose, and L. D. Whitley (Springer, New York, 1999), pp. 89–109.
- [146] D. Deugo and F. Oppacher, in *Artificial Neural Nets and Genetic Algorithms*, edited by R. F. Albrecht, N. C. Steele, and C. R. Reeves (Springer Verlag, Wien, 1993), pp. 400–407.
- [147] D. Sakellariou, A. Lesage, P. Hodgkinson, and L. Emsley, *Chem. Phys. Lett.* **319**, 253 (200).
- [148] C. M. Bishop, *Neural Networks for Pattern Recognition* (Clarendon Press, Oxford, 1995).
- [149] A. Blum, *Neural networks in C++* (Wiley & Sons, New York, 1994).
- [150] T. Masters, *Practical Neural Network Recipes in C++* (Academic Press, Boston, 1996).
- [151] E. Barnard, *IEEE Transactions on Neural Networks* **3(2)**, 232 (1992).
- [152] A. L. Blum and P. Langley, *Arti. Intel.* **97**, 245 (1997).
- [153] X. Yao, *International Journal of Intelligent Systems* **8**, 539 (1993).

Appendix A

Auxillary code

The code presented here are all dependant on the *BlochLib* library and tool kit.

As a result you will probably need it to compile this code. You can get it here

<http://waugh.cchem.berkeley.edu/blochlib/> (and if it is not there I hope to maintain a copy here <http://theadddedones.com/> and perhaps <http://sourceforge.net/>). The code examples here are relatively short and should be easily typed in by hand.

A.1 General C++ code and examples

A.1.1 C++ Template code used to generate prime number at compilation

```
//-----
// Program by Erwin Unruh
// Compile with: g++ -c prime.cc | & grep conversion
// The 'grep' command picks out only the errors we want to see
// namely those with the prime numbers

// Class to create "output" at compile time (error messages)

//gives error on D=int
template <int i, int prim> struct D {};
//no error on D=int
template <int i> struct D<i,0> { D(int); };

// Class to compute prime condition
template <int p, int i> struct is_prime {
    enum { prim = ((p%i) && is_prime < (i>2 ? p : 0), i-1>::prim) };
};
//specific instances to stop
template<> struct is_prime<0,1> { enum { prim = 1}; };
template<> struct is_prime<0,0> { enum { prim = 1}; };
```



```

// Class to iterate through all values: 2..i
template <int i> struct Prime_print {
    Prime_print<i-1> a; //cascade from i to 2
    enum { prim = is_prime<i,i-1>::prim };

    //will produce an error if 'prim'==1
    //(if we have a prime number)
    void f() { a.f(); D<i,prim> d = prim; }
};
//specific instance to stop at i=2
template<> struct Prime_print<2> {
    enum { prim = 1};
    void f() { D<2,prim> d = prim; }
};

void foo() {
    Prime_print<25> a;
    a.f();
}

/*****expected output from 'Prime_print<25> a; a.f();'

prime.cc:30: conversion from 'Prime_print<2>::{anonymous enum}'
to non-scalar type 'D<2,1>' requested
prime.cc:25: conversion from 'Prime_print<3>::{anonymous enum}'
to non-scalar type 'D<3,1>' requested
prime.cc:25: conversion from 'Prime_print<5>::{anonymous enum}'
to non-scalar type 'D<5,1>' requested
prime.cc:25: conversion from 'Prime_print<7>::{anonymous enum}'
to non-scalar type 'D<7,1>' requested
prime.cc:25: conversion from 'Prime_print<11>::{anonymous enum}'
to non-scalar type 'D<11,1>' requested
prime.cc:25: conversion from 'Prime_print<13>::{anonymous enum}'
to non-scalar type 'D<13,1>' requested
prime.cc:25: conversion from 'Prime_print<17>::{anonymous enum}'
to non-scalar type 'D<17,1>' requested
prime.cc:25: conversion from 'Prime_print<19>::{anonymous enum}'
to non-scalar type 'D<19,1>' requested
prime.cc:25: conversion from 'Prime_print<23>::{anonymous enum}'
to non-scalar type 'D<23,1>' requested

*/

```

A.1.2 C++ Template meta-program to unroll a fixed length vector at compilation time

```

// This meta program applies to a fixed length vector
// where the template arguments for this vector
// would be T=the data types, and N the vector length
// we will call this vector a 'coord<T,N>' to distinguish
// between the general vector case.
//
// this is only a code pieces, it will not work unless

```

```

// one has defined a valid coord, and the coordExpr classes

//Here is the = operator that passes
// the expression To the 'coordAssign' meta program
template<class T, int N>
template<class Expr_T>
coord<T, N> &coord<T, N>::operator=(const coordExpr<Expr_T> &rhs)
{
    coordAssign<N,0>::assign(*this, rhs, ApAssign);
    return *this;
}

// This is a 'ApAssign' class for
// a data type 'T'
template<class T>
class ApAssign{
public:
    ApAssign(){}
    static inline void apply(T &a, T &b)
    { a=b; }
};

// a 'quick' meta program (one the compiler performs)
// to unroll loops completely...this is the 'entry' point,
// below a specific instance (N=0, I=0) is expressed
// to stop the template cascade
template<int N, int I>
class coordAssign {
public:

//this is tells us when to stop the cascade
    enum { loopFlag = (I < N-1) ? 1 : 0 };

    template<class CoordType, class Expr, class Op>
    static inline void assign(CoordType& vec, Expr expr, Op u)
    {
        //assign the two elements
        u.apply(vec[I], expr(I));
        //move on the the next instance (I+1)
        coordAssign<N * loopFlag,
            (I+1) * loopFlag >::assign(vec, expr, u);
    }
};

// the class to 'kill' or stop the above one..
// we get here we stop the template unrolling
template<>
class coordAssign<0,0> {
public:
    template<class VecType, class Expr, class Op>
    static inline void assign(VecType& vec, Expr expr, Op u)

```

```

    { }
};

```

A.1.3 C++ code for performing a matrix multiplication with L2 cache blocking and partial loop unrolling.

```

template<class T>
void mulmatUnroll(matrix<T> &c, matrix<T> &a, matrix<T> &b)
{
    int i,j,k, leftover;
    static int Unrolls=5;

    //figure out how many do not fit in the unrolling
    leftover=a.rows() % (Unrolls);

    for(k=0;k<b.rows();++k){
        for(j=0;j<b.cols();++j){

            i=0;
            //do the elements that do not fit in the unrollment
            for(;i<leftover;++i)
                { c(i,j)+=a(i,k) * b(k,j); }

            //do the rest
            for(;i<a.rows();i+=Unrolls){

                //avoid calculating the indexes twice
                int i1=i+1, i2=i+2, i3=i+3, i4=i+4;

                //avoid reading the b(k,j) more then once
                typename matrix<T>::numtype tmpBkj=b(k,j);

                //read the a(i,k)'s first into the registers
                typename matrix<T>::numtype tmpAij=a(i,k);
                typename matrix<T>::numtype tmpA1j=a(i1,k);
                typename matrix<T>::numtype tmpA2j=a(i2,k);
                typename matrix<T>::numtype tmpA3j=a(i3,k);
                typename matrix<T>::numtype tmpA4j=a(i4,k);

                c(i,j)+=tmpAij * tmpBkj;
                c(i1,j)+=tmpA1j * tmpBkj;
                c(i2,j)+=tmpA2j * tmpBkj;
                c(i3,j)+=tmpA3j * tmpBkj;
                c(i4,j)+=tmpA4j * tmpBkj;
            }
        }
    }
}

/* L2 blocking ****/
int L2rowMAX=140;
int L2colMAX=140;

```

```

//makes the sub matrix elements into the
//proper place from the original
template<class T>
void makeSubMatrixFrom(
    matrix<T> &out ,
    matrix<T> &Orig ,
    int beR, //begining row index
    int enR, //ending row index
    int beC, //begining column index
    int enC) //ending column index
{
    out.resize(enR-beR, enC-beC);
    for(int i=beR, ctR=0;i<enR;++i, ++ctR){
        for(int j=beC, ctC=0;j<enC;++j, ++ctC){
            out(ctR, ctC)=Orig(i, j);
        }
    }
}

//puts the sub matrix elements into the
//proper place in the original
template<class T>
void putSubMatrixTo(
    matrix<T> &in ,
    matrix<T> &Orig ,
    int beR, //begining row index
    int enR, //ending row index
    int beC, //begining column index
    int enC) //ending column index
{
    for(int i=beR, ctR=0;i<enR;++i, ++ctR){
        for(int j=beC, ctC=0;j<enC;++j, ++ctC){
            Orig(i, j)+=in(ctR, ctC);
        }
    }
}

template<class T>
void L2BlockMatMul(matrix<T> &C, matrix<T> &A, matrix<T>&B)
{
    //resize our return matrix to the proper size
    C.resize(A.rows(), B.cols());
    C=0;
    //no need to do this if matrix is less then L2 size
    if(A.rows()<L2rowMAX && B.cols() < L2colMAX)
        { mulmatLUnrool(C,A,B); return; }

    //the number of divisions along rows and cols
    int rDiv=(int) ceil(double(A.rows())/double(L2rowMAX));
    int cDiv=(int) ceil(double(B.cols())/double(L2colMAX));
    int BDiv=(int) ceil(double(B.rows())/double(L2colMAX));
    int i, j, k;

```

```

//now do C(i,j)=Sum_k (a(i,k)*b(k,j))

for(i=0;i<rDiv;++i){
//the current beginning Row index for out matrix
int beCr=i*L2rowMAX;
//the current ending Row index for out matrix
int enCr=(i+1)*L2rowMAX;
if(enCr>A.rows()) enCr=A.rows();

for(j=0;j<cDiv;++j){
//the current beginning Column index for out matrix
int beCc=j*L2colMAX;
//the current ending Row index for out matrix
int enCc=(j+1)*L2colMAX;
if(enCc>B.cols()) enCc=B.cols();

//sub output matrix for out matrix
matrix<T> Cij(enCr-beCr, enCc-beCc);
//zero out the matrix
Cij=0;

//now loop through the B Row divisions
for(k=0;k<BDiv;++k){
//this value is beginning for the columns
// of A and the rows of B
int beAB=k*L2colMAX;

//this value is for end the columns
// of A and the rows of B
int enAB=(k+1)*L2colMAX;
if(enAB>B.cols()) enAB=B.rows();

//sub A and B matrices
matrix<T> Aik;
makeSubMatrixFrom(Aik, A, beCr, enCr, beAB, enAB);
matrix<T> Bkj;
makeSubMatrixFrom(Bkj, B, beAB, enAB, beCc, enCc);

//perform the multiply on the subs
// noting that the elements in Cij will be
// added to (not overwritten)
mulmatLUnroll(Cij, Aik, Bkj);
}
//put the sub C matrix back into the original
putSubMatrixTo(Cij, C, beCr, enCr, beCc, enCc);
}
}
}

```

A.1.4 An MPI master/slave implementation framework

```
#include "blochlib.h"
```

```

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//define out function we wish to run in parallel
void MyFunction(int kk){
    cout<<endl<<"I was called on:_"<<MPIworld.rank()
        <<"_with value:_"<<kk<<endl;
    sleep(MPIworld.rank()-1);
}

int main(int argc, char* argv[])
{
//Start up the Master controller
    MPIworld.start(argc, argv);

//dump out info about what and where we are
    std::cout<<MPIworld.name()<<"::"<<MPIworld.rank()
        <<" /"<<MPIworld.size()<<std::endl<<endl;

//this int gets sent when the Master has sent
//everything (the kill switch)
    int done=-1;
    int cur=0; //the current value

//if we are the master, we need to initialize some things
    if(MPIworld.master()){

//the elements in here will be sent to the slave procs
        int Max=10; //only want to send 10 things
        int CT=0, rr=-1;

//we must perform an initial send to all the proc
//from 1..size, if size>Max we need to send no more
        for(int qq=1;qq<MPIworld.size();++qq){
            MPIworld.put(CT, qq); ++CT;
            if(CT>Max) break;
        }
        int get;

//now we get an Integer from ANY processor that is NOT
//the master...and keep putting values until we run out
        while(CT<Max){
//get an int ('get'=the proc is came from)
            get=MPIworld.getAny(rr);
            MPIworld.put(CT, get); //put the next value
            ++CT; //advance
        }

//put the 'We-Are-Done' flag to all the procs once we finish
        for(int qq=1;qq<MPIworld.size();++qq)

```

```

        MPIworld.put(done, qq);

    }else{ //slave procs

//keep looping until we the master tells us to quit
    while(1){
        MPIworld.get(cur,0);
        if(cur==done) break; //id we get the kill switch get out
        MyFunction(cur); //run out function with the gotten value
        MPIworld.put(cur,0); //send back a request for more
    }
}

//exit MPI and leave the prog
MPIworld.end();
return 0;
}

```

A.1.5 C++ class for a 1 hidden layer Fully connected back-propagation Neural Network

```

/*
A simple 1 hidden layer Back propgation
fully connected Feed Foward neural Net
*/

#include "blochlib.h"

using namespace BlochLib;

template<class Num_T>
class sigmoid{
public:
    Num_T operator()(int i, Num_T &in){ return sigmoid(in); }
    inline static Num_T sigm(Num_T num) // The sigmoid function.
    { return (1./(1.+exp(-num))); }
};

template<class Num_T> //Num_T is the output/input data type
class BackPropNN {

private:
// Weights for the neurons input—hidden
    rmatrixs IHweights_;
// Weights for the neurons hidden—>output
    rmatrixs HOweights_;

    Vector<float> IHbias_; //the in—hidden biases
    Vector<float> HObias_; //the hidden—out biases
    Vector<Num_T> hlayer_; //the hidden layer 'values'
    Vector<Num_T> outTry_; //the attempted outputs

    Vector<Num_T> outError_; //the ouput—>hidden errors

```

```

    Vector<Num_T> hiddenError_; //the hidden—>input errors

    float lrate_;

public:
    BackPropNN();
    BackPropNN(int numin, int numH, int numout);
    ~BackPropNN() {};

    //resize the ins and outs
    void resize(int numin, int numH, int numout);

    //reset the weights to random
    void reset();

    inline float learningRate()
    { return lrate_; } // getthe learing rate

    void learningRate(float lr)
    { lrate_=lr; } //set the learing rate

    void fowardPass(Vector<Num_T> &in);
    void backPass(Vector<Num_T> &input, Vector<Num_T> &target);

    Vector<Num_T> run(Vector<Num_T> &input);
    Vector<Num_T> train(Vector<Num_T> &input, Vector<Num_T> &target);

    rmatrixs IHweights(){ return IHweights_; }
    rmatrixs HOweights(){ return HOweights_; }

    //dumps a matlab file that
    // plots the neurons with lines between
    // them based on the weight
    void print(std::string fname);

    float error(Vector<Num_T> &target);

};

template<class Num_T>
BackPropNN<Num_T>::BackPropNN(int numin, int numout, int numH=0)
{
    lrate_=0.5;
    resize(numin, numout, numH);
}

template<class Num_T>
void BackPropNN<Num_T>::resize(int numin, int numout, int numH=0)
{
    RunTimeAssert(numin>=1);

```



```

RunTimeAssert(numout>=1);
if(numH==0) numH=numin;
RunTimeAssert(numH>=1);

//the weight size is (numin+1)x(numin+1)
// the '+1' for the bias entries
IHweights_.resize(numin, numH);
HWeights_.resize(numH,numout);
IHbias_.resize(numH,0);
HObias_.resize(numout,0);
hlayer_.resize(numH,0);
outTry_.resize(numout,0);
outError_.resize(numout,0);
hiddenError_.resize(numH,0);
reset();
}

template<class Num_T>
void BackPropNN<Num_T>::reset()
{
Random<UniformRandom<float>> myR(-1, 1);
HWeights_.apply(myR);
IHweights_.apply(myR);
IHbias_.apply(myR);
HObias_.apply(myR);
hlayer_.fill(0.0);
outTry_.fill(0.0);
}

//this does the foward propogation...
template<class Num_T>
void BackPropNN<Num_T>::fowardPass(Vector<Num_T> &in)
{
register int i,j;
register Num_T tmp=0;

//input --> hidden
for(i=0;i<IHweights_.cols();++i){
for(j=0;j<in.size();++j){
tmp+=in(j)*IHweights_(j,i);
}
hlayer_[i]=sigmoid<Num_T>::sigm(tmp+IHbias_(i));
tmp=0;
}
//hidden --> output
for(i=0;i<outTry_.size();++i){
for(j=0;j<HWeights_.rows();++j){
tmp+=hlayer_(j)*HWeights_(j,i);
}
outTry_[i]=sigmoid<Num_T>::sigm(tmp+HObias_(i));
tmp=0;
}
}
}

```

```

template<class Num_T>
float BackPropNN<Num_T>::error (Vector<Num_T> &target)
{
    return norm(target-outTry_);
}

//this does the backwards propogation...
template<class Num_T>
void BackPropNN<Num_T>::backPass(
    Vector<Num_T> &input ,
    Vector<Num_T> &target)
{
    register int i,j;
    register Num_T tmp=0;

    //error for ouputs
    outError_=target-outTry_;

    //error for hidden
    for (i=0;i<HOweights_.cols();++i){
        for (j=0;j<outTry_.size();++j){
            tmp+=outError_[j]*HOweights_(i,j);
        }
        hiddenError_(i)=float (hlayer_(i)*(1.0-hlayer_(i))*tmp);
        tmp=0;
    }

    //adjust hidden—>output weights
    Num_T len=0;
    len=sum(sqr(hlayer_)); //the mean length of the hidden
    if (len<=0.1) len=0.1; //do not reduce too much...
    for (i=0;i<HOweights_.rows();++i){
        for (j=0;j<outTry_.size();++j){
            HOweights_(i,j)+=float (lrate_*outError_(j)*hlayer_(i)/len);
        }
    }

    //adjust hidden bias levels
    for (i=0;i<HObias_.size();++i){
        HObias_(i)+=float (lrate_*outError_(i)/len);
    }

    //adjust weights from input to hidden
    len=sum(sqr(input));
    if (len<=0.1) len=0.1; //do not reduce too much...
    for (i=0;i<input.size();++i){
        for (j=0;j<IHweights_.cols();++j){
            IHweights_(i,j)+=float (lrate_*hiddenError_(j)*input(i)/len);
        }
    }

    //adjust input bias levels
    for (i=0;i<IHweights_.cols();++i){

```

```

    IHbias_(i)+=float(lrate_*hiddenError_(i)/len);
  }
}

template<class Num_T>
Vector<Num_T> BackPropNN<Num_T>::
  train(Vector<Num_T> &in , Vector<Num_T> &out)
{
  fowardPass(in);
  backPass(in , out);
  return outTry_;
}

template<class Num_T>
Vector<Num_T> BackPropNN<Num_T>::
  run(Vector<Num_T> &in)
{
  fowardPass(in);
  return outTry_;
}

//this dumps the info to a matab
// script so that it can be easily plotted
template<class Num_T>
void BackPropNN<Num_T>::print(std::string fname)
{
  std::ofstream oo(fname.c_str());
  if(oo.fail()){
    std::cerr<<std::endl<<"BackPropNN. print"<<std::endl;
    std::cerr<<" cannot_open_output_file "<<std::endl;
    return;
  }
}

/*we wish the picture to look like
  O   O
 / \ / \
O   O   O
 \ / \ /
  O   O

*/

//the 'dot' for a Neuron
oo<<" figure(153);\n"
  <<" clf_reset;\n";

//we want each node to be spearated by
// 5 in the on the x 'axis' and 10 on the yaxis
// we need to scale the xaxis based on the maxNode
oo<<" inNodes="<<IHweights_.rows()<<" ;\n"
  <<" hNodes="<<IHweights_.cols()<<" ;\n"
  <<" outNodes="<<outTry_.size()<<" ;\n"

```

```

<<"maxNodes=max(inNodes ,max(hNodes , outNodes ));\n"
<<"ySep=10;\n"
<<"xSep=2*ySep;\n"
<<"ybSep=2;\n"
<<"inSep=(xSep/(inNodes+2));\n"
<<"hSep=(xSep/(hNodes+2));\n"
<<"outSep=(xSep/(outNodes+1));\n"
<<"xc=[-1 ,_-1 ,_1 ,_1];\n"
<<"yc=[-1 ,_1 ,_1 ,_-1];\n"
<<"hold_on\n";

//print out the weights and biases
oo<<"IHweights=[";
for (int i=0;i<IHweights_.rows();++i){
  oo<<" ";
  for (int j=0;j<IHweights_.cols();++j){
    oo<<IHweights_(i ,j)<<" ";
  }
  oo<<" ]\n";
}
oo<<" ];\n";

oo<<"HWeights=[";
for (int i=0;i<HWeights_.rows();++i){
  oo<<" ";
  for (int j=0;j<HWeights_.cols();++j){
    oo<<HWeights_(i ,j)<<" ";
  }
  oo<<" ]\n";
}
oo<<" ];\n"
<<"IHbias=["<<IHbias_<<" ];\n"
<<"HObias=["<<HObias_<<" ];\n"

//find the max of all of them
<<"maxW=max(max(abs(IHweights)));\n"
<<"maxW=max(maxW,max(max(abs(HWeights))));\n"
<<"maxW=max(maxW,max(max(abs(IHbias))));\n"
<<"maxW=max(maxW,max(max(abs(HObias))));\n"
<<"maxWidth=5;\n"
<<"altColor=[0 ,0 ,_0.8];\n"
<<"posColor=[0.8 ,0 ,0];\n"

//print a line for each one of them...
<<"%_INput—>HIDDEN_lines_\n"
<<"for _i=1:hNodes_\n"
<<"  _for _j=1:inNodes_\n"
<<"    _color=posColor;\n"
<<"    _if _IHweights(j ,i)<0,_color=altColor; ,_end;\n"
<<"    _li=line ([j*inSep_+i*hSep] , [2*ySep_+ySep] ,_)
<<"    _'Color' ,_color ,_'LineWidth' ,"
```

```

<<" .....maxWidth*abs(IHweights(j,i))/maxW);\n"
<<" ..end\n"
<<" end\n"

<<"%_Hidden—>out_lines_\n"
<<" for _i=1:hNodes_\n"
<<" ..for _j=1:outNodes_\n"
<<" .....color=posColor;_\n"
<<" .....if _HWeights(i,j)<0, _color=altColor; ,_end;_\n"
<<" .....li=line([i*hSep+_j*outSep],[ySep_0],_)"
<<" ..... 'Color', _color, _'LineWidth',_"
<<" .....maxWidth*abs(HWeights(i,j))/maxW);\n"
<<" ..end\n"
<<" end\n"

<<"%_input_Bias—>Hidden_lines_\n"
<<" j=inNodes+1;\n"
<<" for _i=1:hNodes_\n"
<<" .....color=posColor;_\n"
<<" .....if _IHbias(i)<0, _color=altColor; ,_end;_\n"
<<" .....li=line([j*inSep+_i*hSep],[2*ySep-ybSep_ySep],_)"
<<" ..... 'Color', _color, _'LineWidth',_"
<<" .....maxWidth*abs(IHbias(i))/maxW);\n"
<<" end\n"

<<"%_Hidden_Bias—>output_lines_\n"
<<" j=hNodes+1;\n"
<<" for _i=1:outNodes_\n"
<<" .....color=posColor;_\n"
<<" .....if _HObias(i)<0, _color=altColor; ,_end;_\n"
<<" .....li=line([j*hSep+_i*outSep],[ySep-ybSep_0],_)"
<<" ..... 'Color', _color, _'LineWidth',_"
<<" .....maxWidth*abs(HObias(i))/maxW);\n"
<<" end\n";

oo<<" for _i=1:inNodes\n"
<<" ... fill(xc/inNodes+i*inSep, _yc/inNodes+2*ySep, _'r');\n"
<<" end\n"
<<"%bias_I—>H_node\n"
<<" fill(xc/inNodes+(inNodes+1)*inSep,_"
<<" .....yc/inNodes+2*ySep-ybSep, 'g');\n"
<<" \n"
<<" for _i=1:hNodes\n"
<<" ... fill(xc/hNodes+i*hSep, _yc/hNodes+ySep, _'k');\n"
<<" end\n"
<<"%bias_H—>O_node\n"
<<" fill(xc/hNodes+(hNodes+1)*hSep, _yc/hNodes+ySep-ybSep, _'g');\n"
<<" \n"
<<" for _i=1:outNodes\n"
<<" ... fill(xc/outNodes+i*outSep, _yc/outNodes, _'b');\n"
<<" end\n"
<<" daspect([1_1_1]);\n"
<<" axis_tight;\n"

```

```

    <<"hold_off\n"
    <<"\n";
}

```

A.2 NMR algorithms

A.2.1 Mathematica Package to generate Wigner Rotation matrices and Spin operators.

This small and simple Mathematica package (a .m file) allows the creation of the basic Cartesian spin operators and Wigner rotation matrices of a given spin space of spin I . To use the package, simple call `MakeSpace[Spin]` where `Spin` is the total spin (i.e. 1/2, 1, 3/2, etc). It will make `Iz`, `Ix`, `Iy`, `Ipp` and `Imm` as global matrices. To generate the Wigner rotation matrix call `Wigner[Spin]` where `Spin` is the same as the `MakeSpace` value.

```

(* spinten.m*)
(*In this package we try to creat all the nessesary bits
for generating everything we could possibly want to
do with spin tensors and rotations*)

Unprotect[Ix, Iy, Iz, Ipp, Imm, rank, created, WignerExpIy, D12, d12]
Clear[Ix, Iy, Iz, Ipp, Imm, rank, created, WignerExpIy, D12, d12]

BeginPackage["spinten`"]

Unprotect[MakeSpace, MakeIz, MakeIplus, MakeIx, MakeIy, MakeImin,
Wigner, Direct, MultWig, MakeSpinSys, MakeExpIz]

Clear[MakeSpace, MakeIz, MakeIplus, MakeIx, MakeIy, MakeImin,
Wigner, Direct, MultWig, MakeSpinSys, MakeExpIz]

(* Usages*)

Wigner::usage=
  "Wigner[L,m,mp,{alpha,beta,gamma}] generates a wigner
  \n rotation element
  \n <mp,L|Exp[-Iz alpha] Exp[-Iy beta] *
  \n Exp[-Iz gamma]|m,L>.
  \n Other possibles include:
  \n Wigner[L] --> For an entire matrix
  \n Wigner[L,{alpha,beta,gamma}] --> matrix using default
  'alpha,beta,gamma'
  \n Wigner[L,mp,m] --> using default
  \n 'alpha,beta,gamma' symbols";

Wigner::errmb="m's in 'Wigner' is bigger than L.. Bad,bad, person";
Wigner::errms="m's in 'Wigner' is smaller than L.. Bad,bad, person";

```

```

MultWig:: usage=
  "MultWig[{L1, L2}, {J, M3p, M3}] Will give the Dj(m3p, m3) wigner
  \n elements from two Other Wigner Matrices !!";

MultWig:: errm="You J or M3p or M3 is too big for L1+L2";

MakeSpace:: usage=
  "MakeSpace[L] this function generates all the matrices for spin=L
  \n systems. The output simply creates definitions for Iz, Ix,
  \n and Iy Which can then be called up as Ix, Iy, Iz.
  \n If you have defined them previously this will redefine them";

MakeSpace:: err = "you have entered in a value for L that is not
  \n and interger or half an interger";

MakeIz:: usage = "MakeIz[L] Generates Iz in space of rank L";
MakeIx:: usage="MakeIx[L] Generates Ix in space of rank L";
MakeIy:: usage="MakeIy[L] Generates Iy in space of rank L";
MakeIplus:: usage = "MakeIplus[L] Generates I+ in space of rank L";
MakeImin:: usage = "MakeImin[L] Generates I- in space of rank L";
MakeExpIz:: usage="MakeExpIz[a, L] A faster way of doing exp[a Iz]";

Direct:: usage =
  "Direct[m, p] the creates a direct product of
  \n two matrices {m and p}";

MakeSpinSys:: usage=
  "MakeSpinSys[{L1, L2, L3}] this function generates all the
  \n matrices for spin=L systems.
  \n The output simply creates a LIST for Iz, Ix,
  \n and Iy Which can then be called up as Ix, Iy, Iz.
  \n If you have defined them previously this will redefine them";

Begin["Private"]

(* here we define the 'base/default' pauli matrices
  (they are for spin 1/2) *)

Global 'Iz=1/2{{1,0},{0,-1}}
Global 'Ix=1/2{{0,1},{1,0}}
Global 'Iy=1/2{{0,-I},{I,0}}
Global 'Ipp={{0,1},{0,0}}
Global 'Imm={{0,0},{1,0}}
Global 'numspin=1;
Global 'rank=1/2
Global 'd12=MatrixExp[-I Global '[Beta] \
  Global 'Iy]//ExpToTrig//Simplify
Global 'D12=(MatrixExp[-I Global '[Alpha] Global 'Iz].
  Global 'd12.
  MatrixExp[-I Global '[Gamma] Global 'Iz])//Simplify

```

```
(* this flag tells me that i have already created made the matrix
   Exp[-I beta Iy]. This can be a large large,
   especially symbolically and need only be done
   once (of course unless i change my L)*)
```

```
Global 'createdwig=0;
```

```
(*here is a function that does a direct
   product between two matrices*)
```

```
Direct [m_, p_] :=
Module [{dimM=Dimensions[m][[1]], dimP=Dimensions[p][[1]], froo},
If [dimM==0||dimP==0,
      Print ["Bad_person ,_you_gave_me_a_'NULL'_for_a_matrix" ]];

  froo=Table[0, {i,1,dimP*dimM},{j, 1, dimP*dimM}];
  Table[Table[
    froo [[ i+(1*dimP), j+(k*dimP)]] = m[[ l+1,k+1]] * p[[ i, j]], \
    {i, 1, dimP},{j, \
      1, dimP},{l, 0, dimM-1},{k, 0, dimM-1}]]];
```

```
(*here is a function that creates the Iz, Iplus,
   and Imin matrix for a Rank L spin space by doing
   using these simple identities
```

```
Iz |L,m> = m |L,m>
I+/- |L,m> = Sqrt[L(L+1)-m(m+/-1)] |L, m+/-1>
*)
```

```
MakeIz [L_] :=
Table[
  Table[
    If [mp==m, m, 0],
    {mp, L, -L, -1}, {m, L, -L, -1}]]
```

```
MakeIplus [L_] :=
Table[
  Table[
    If [mp==(m+1), Sqrt [L(L+1)-m(m+1)], 0],
    {mp, L, -L, -1}, {m, L, -L, -1}]]
```

```
MakeImin [L_] :=
Table[
  Table[
    If [mp==m-1, Sqrt [L(L+1)-m(m-1)], 0],
    {mp, L, -L, -1}, {m, L, -L, -1}]]
```

```
MakeIx [L_] := 1/2 (MakeIplus [L]+MakeImin [L])
```

```
MakeIy [L_] := -I 1/2 (MakeIplus [L]-MakeImin [L])
```

```
MakeSpace [L_] := Module [{},
If [Mod[L, 0.5]!=0, Message [MakeSpace::err];,
If [Global 'rank!=L,
```



```

Global 'rank=L;
Global 'Iz=MakeIz[L];
Global 'Ipp=MakeIplus[L];
Global 'Imm=MakeImin[L];
Global 'Ix=1/2(Global 'Ipp+Global 'Imm);
Global 'Iy=-I 1/2(Global 'Ipp-Global 'Imm);
Global 'createdwig=0;]]

MakeExpIz[a_ , L_] :=
  Table[
    Table[
      If[mp==m, Exp[m a], 0],
      {mp, L, -L, -1}, {m, L, -L, -1}]]

Wigner[L_ , mp_ , m_ , {alpha_ , beta_ , gamma_}] :=
  Module[{l=1/2},
    If[mp>L, Message[Wigner::errmb],
      If[mp<-L, Message[Wigner::errms],
        If[m>L, Message[Wigner::errmb],
          If[m<-L, Message[Wigner::errms]]]]];
    tmp=Global 'd12;
    While[l<L-1/2,
      tmp=MultWig[{tmp, Global 'd12}, l+1/2];
      l=1+1/2;
    ];
    If[L==0, 1,
      If[L==1/2, Global 'D12,
        Exp[-I mp Global '\[Alpha]]*Exp[-I m Global '\[Gamma]]*
        MultWig[{tmp, Global 'd12}, {L, mp, m}]]]
  ]

Wigner[L_ , mp_ , m_] := Wigner[L, mp, m, {alpha, beta, gamma}]

Wigner[L_ , {alpha_ , beta_ , gamma_}] :=
  Module[{l, tmp},
    l=1/2;
    If[mp>L, Message[Wigner::errmb],
      If[mp<-L, Message[Wigner::errms],
        If[m>L, Message[Wigner::errmb],
          If[m<-L, Message[Wigner::errms]]]]];
    tmp=Global 'd12;
    While[l<L-1/2,
      tmp=MultWig[{Global 'd12, tmp}, l+1/2];
      l=1+1/2;
    ];
    MakeExpIz[-I Global '\[Alpha], L].
    If[L==0, 1,
      If[L==1/2, Global 'd12,
        MultWig[{Global 'd12, tmp}, L]].
    MakeExpIz[-I Global '\[Gamma], L]
  ]

```

```

Wigner[L_]:=Wigner[L, {alpha, beta, gamma}]

MultWig[{L1_?MatrixQ, L2_?MatrixQ}, {J_, m3p_, m3_}, {a_, b_, g_}]:=
Module[{
  l1=(Dimensions[L1][[1]]-1)/2,
  l2=(Dimensions[L2][[1]]-1)/2,

  If[J>l1+l2, Message[MultWig::errm],
    (Sum[
      Sum[If[m3-m1>l2 || m3-m1<-l2 || m3p-m1p>l2 || m3p-m1p<-l2, 0,
        ClebschGordan[{l1, m1}, {l2, m3-m1}, {J, m3}]*
        ClebschGordan[{l1, m1p}, {l2, m3p-m1p}, {J, m3p}]*
        L1[[l1-m1p+1]][[l1-m1+1]]*
        L2[[l2-(m3p-m1p)+1]][[l2-(m3-m1)+1]]],
      {m1, l1, -l1, -1}],
    {m1p, l1, -l1, -1}]]//Simplify]
]

MultWig[{L1_?MatrixQ, L2_?MatrixQ}, {J_, m3p_, m3_}]:=
MultWig[{L1, L2}, {J, m3p, m3}, {a, b, g}]

MultWig[{L1_, L2_}, J_]:=
Table[MultWig[{L1, L2}, {J, i, j}], {i, J, -J, -1}, {j, J, -J, -1}]

MultWig[{L1_, L2_}, J_, {a_, b_, g_}]:=
Table[MultWig[{L1, L2}, {J, i, j}, {a, b, g}], \
  {i, J, -J, -1}, {j, J, -J, -1}]

MakeSpinSys[spinsizes_]:=
Module[{i},
  If[Length[spinsizes]==1, MakeSpace[spinsizes[[1]]],
  If[Length[spinsizes]==0, MakeSpace[spinsizes],
  Global`Ix=Table[MakeIx[spinsizes[[i]]], \
    {i, 1, Length[spinsizes]}];
  Global`Iy=Table[MakeIy[spinsizes[[i]]], \
    {i, 1, Length[spinsizes]}];
  Global`Iz=Table[MakeIz[spinsizes[[i]]], \
    {i, 1, Length[spinsizes]}];
  Global`Ipp=Table[MakeIplus[spinsizes[[i]]], \
    {i, 1, Length[spinsizes]}];
  Global`Imm=Table[MakeImin[spinsizes[[i]]], \
    {i, 1, Length[spinsizes]}];
  Global`numspin=Length[spinsizes];
  ]];]

(*T=Function[{L, m, spinsys},
  Module[{tmpix},
    MakeSpinSys[spinsys];
  *)

End[]

Protect[Wigner, MakeSpace, MakeIz, MakeIplus, MakeImin,

```

```
MakeIx, MakeIy, Direct, MultWig, MakeSpinSys, MakeExpIz]
```

```
EndPackage []
```

A.2.2 Rational Reduction C++ Class

This includes both the C++ header file, the C++ source file and an example usage file.

The header file

```
#ifndef _Prop_Reduce_h_
#define _Prop_Reduce_h_ 1

#include "blochlib.h"

/** This class should be used as follows... */

PropReduce myred(base, fact, log);
myred.reduce();

for(int i=0;i<base;++i){
  < generate the ind and fow props... >
}
for(int i=0;i<myred.maxBackReduce();++i){
  < generate the back props... >
}

Vector<matrix> myred.generateProps(ind, fow, bac);
***/

/** The Rational Reduction of Propogators ***/

using namespace BlochLib;

class PropReduce{
private:
  Vector<Vector<int>>> BackRed;
  Vector<int> BackName;

  Vector<Vector<int>>> FowardRed;
  Vector<int> FowardName;

  Vector<Vector<int>>> SpecialRed;
  Vector<int> SpecialName;

  Vector<Vector<int>>> dat;

  int Mults, UseMe, baseTag, speTag;
```

```

    bool iteration(Vector<Vector<int>> &dat,
                  Vector<Vector<int>> &propRed,
                  Vector<Vector<int>> &subN,
                  Vector<int> &name);

public:

    int base, factor;
    std::ostream *logf;

//constructors
    PropReduce(){}
    PropReduce(int bas, int factor, std::ostream *oo=0);

    void setParams(int bas, int fact, std::ostream *oo=0);

    void fowardReduce();
    void backReduce();
    void specialReduce();

    void reduce();
    inline int bestMultiplications(){ return Mults; }

    inline int maxBackReduce() const { return UseMe+1; }
    inline int maxFowardReduce() const { return FowardRed.size(); }

//these functions will create the propogators
// from 3 input matrix lists..the first are the
// individual propogators ("0","1","2"... )
// the second the 'Foward' props ("0*1","0*1*2" ...)
// the third the, 'Back' props ("7*8", "6*7*8"... )
// the forth is the place to fill...
    void generateProps(Vector<matrix> &indiv,
                      Vector<matrix> &Foward,
                      Vector<matrix> &Back,
                      Vector<matrix> &FillMe);

};

```

```
#endif
```

The source file

```

#include "blochlib.h"
#include "propreduce.h"

using namespace BlochLib;
using namespace std;

PropReduce::PropReduce(int bas, int fac, ostream *oo)
{ setParams(bas, fac, oo); UseMe=0; Mults=0;}

```

```

void PropReduce::setParams(int bas , int fac , ostream *oo)
{
  //find the greatest common divisor...
  int u = abs(bas);
  int v = abs(fac);
  int q,t;
  while(v){
    q = int(floor( double(u)/double(v) ));
    t = u - v*q;
    u = v;
    v = t;
  }

  base=bas/u;
  factor=fac/u;
  logf=oo;

  dat.resize( base , Vector<int>(factor) );
  FowardName.resize( base-1);
  FowardRed.resize( base-1);

  BackName.resize( base-1);
  BackRed.resize( base-1);

  int ct=0, ct2=0;
  for( int i=0;i<base*factor;++i){
    dat[ ct ][ ct2]=i%base;
    ++ct2;
    if(ct2>=factor){ ++ct; ct2=0; }
  }

  baseTag=100*BlochLib::max( base , factor );
  for( int i=1;i<base;++i){
    FowardName[ i-1]=i*baseTag;
    FowardRed[ i-1].resize( i+1);
    for( int j=0;j<=i;++j) FowardRed[ i-1][j]=j;

    if(logf)
      *logf<<"Foward_reduction :_"<<FowardName[ i-1]<<"="
      <<FowardRed[ i-1]<<std::endl;

    BackName[ i-1]=-i*baseTag;
    BackRed[ i-1].resize( i+1);
    for( int j=base-i-1, k=0;j<base;++j,++k){ BackRed[ i-1][k]=j; }

    if(logf)
      *logf<<"Back_reduction :_"<<BackName[ i-1]<<"="
      <<BackRed[ i-1]<<std::endl;
  }

  //this is a special one which simply trims the the 0 and base-1

```

```

// factor and can be calcd by U(0)'*U(t_r)*U(base-1)'
SpecialRed.resize(1, Vector<int>(base-2));
speTag=20000*BlochLib::max(base, factor);
SpecialName.resize(1, speTag);
for(int i=1; i<base-1; ++i){
    SpecialRed[0][i-1]=i;
}
if(logf)
    *logf<<" Special_reduction:_"<<SpecialRed[0]<<"="
    <<SpecialName[0]<<std::endl<<std::endl;
}

bool PropReduce::iteration(Vector<Vector<int>> &dat,
    Vector<Vector<int>> &propRed,
    Vector<Vector<int>> &subN,
    Vector<int> &name)
{
    //loops to find the matches
    bool gotanyTot=false;

    for(int i=0; i<dat.size(); ++i){
        bool gotany=false;
        Vector<int> curU;
        for(int M=0; M<dat[i].size(); ++M){
            bool got=false;
            int p=0;
            for(p=subN.size()-1; p>=0; --p){
                if(subN[p].size()+M<=dat[i].size()){
                    if(subN[p]==dat[i](Range(M, M+subN[p].size()-1))){
                        got=true;
                        break;
                    }
                }
            }
        }

        if(got){
            for(int k=0; k<M; ++k){
                curU.push_back(dat[i][k]);
            }
            curU.push_back(name[p]);
            for(int k=subN[p].size()+M; k<dat[i].size(); ++k){
                curU.push_back(dat[i][k]);
            }
            propRed[i]=curU;
            gotany=true;
            break;
        }
    }
    if(!gotany){
        for(int k=0; k<dat[i].size(); ++k){
            curU.push_back(dat[i][k]);
        }
    }
}

```

```

    propRed [ i ]=(curU);
  }else{
    gotanyTot=true;
  }
}
}
return gotanyTot;
}

/** foward reductions... */
void PropReduce::fowardReduce()
{
  Vector<Vector<int>> propRed(base, Vector<int>(0));
  while(iteration(dat, propRed, FowardRed, FowardName)){
    dat=propRed;
  }

  int multi=0;
  for(int i=0;i<dat.size();++i){
    if(logf) *logf<<"Sequence_"<<i<<" :_"<<dat [ i]<<endl;
    multi+=dat [ i ]. size ();
  }
  if(logf)
    *logf<<"_After_Foward_Reduction... Number_of_multiplications :_"
    <<multi<<endl<<endl;
}

/**Back Reductions */
// the back reductions we do note get for free
// (like the forward ones whcih we have to calc
// from the exp(H) operation), so the number
// of back reductions used depends on the total multiplication
// saveings...so we need to go through the entire loops of
// back reductions...
void PropReduce::backReduce()
{
  Vector<Vector<int>> propRed(base, Vector<int>(0));
  Vector<Vector<int>> holdDat(dat.size());
  for(int i=0;i<dat.size();++i) holdDat [ i ]=dat [ i ];

  Mults=1000000;
  int multi=0;
  UseMe=0;
  Vector<Vector<int>> curBack;
  Vector<int> curName;
  for(int k=0;k<BackRed.size();++k){
    if(logf) *logf<<"_Number_of_'Back_Reductions ':_"<<k<<endl;
    curBack=BackRed(Range(0, k));
    curName=BackName(Range(0, k));

    for(int i=0;i<dat.size();++i) dat [ i ]=holdDat [ i ];

```

```

while(iteration(dat, propRed, curBack, curName)){
    dat=propRed;

    multi=curBack.size();
    for(int j=0;j<dat.size();++j){
        multi+=dat[j].size();
    }

    if(Mults>multi){
        UseMe=k;
        Mults=multi;
    }
}
for(int j=0;j<dat.size();++j){
    if(logf)
        *logf<<" Sequence_"<<j<<" :_"<<dat[j]<<std::endl;
}
if(logf)
    *logf<<" _After_Back_Reduction ... Number_of_multiplications:_ "
    <<multi<<std::endl<<std::endl;
}

//need to 'regen' the best one for displaying
if(logf) *logf<<" _Number_of_'Back_Reductions' :_"<<UseMe<<std::endl;
curBack=BackRed(Range(0,UseMe));
curName=BackName(Range(0,UseMe));

for(int i=0;i<dat.size();++i) dat[i]=holdDat[i];

Vector<int> BackNeedToGen;
while(iteration(dat, propRed, curBack, curName)){
    dat=propRed;
}
}

/** Special Reductions */
void PropReduce::specialReduce()
{
    Vector<Vector<int>> propRed(base, Vector<int>(0));
    while(iteration(dat, propRed, SpecialRed, SpecialName)){
        dat=propRed;
    }

    Vector<Vector<int>> curBack=BackRed(Range(0, UseMe));
    int multi=curBack.size();
    for(int i=0;i<dat.size();++i){
        if(logf) *logf<<" Sequence_"<<i<<" :_"<<dat[i]<<std::endl;
        multi+=dat[i].size();
    }
    int ttt=Mults-multi; //savings for 'specials'
    Mults-=ttt;
    if(logf)
        *logf<<" _After_Special_Reduction ... Number_of_multiplications:_ "

```



```

    <<Mults<<std::endl<<std::endl;
}

void PropReduce::reduce()
{
    fowardReduce();
    backReduce();
    specialReduce();
    if(logf){
        *logf<<endl<<"_The_Best_Reduction_is_for_using_"
        <<UseMe+1<<"_Back_Reductions"<<std::endl;
        *logf<<"_For_a_grand_total_of_"<<Mults
        <<"_multiplications"<<std::endl;
        *logf<<"_The_total_Sequence...."<<std::endl;
    }
    for(int j=0;j<dat.size();++j){
        if(logf) *logf<<"Sequence_"<<j<<" :_"<<dat[j]<<std::endl;
    }
}

//these functions will create the propogators
// from 3 input matrix lists..the first are the
// individual propogators ("0","1","2"... )
// the second the 'Foward' props ("0*1","0*1*2" ...)
// the third the, 'Back' props ("7*8", "6*7*8"... )
// the forth is the place to fill...
void PropReduce::
    generateProps(Vector<matrix> &indiv ,
                 Vector<matrix> &Foward ,
                 Vector<matrix> &Back ,
                 Vector<matrix> &FillMe)
{
    if(indiv.size() != base){
        std::cerr<<" PropReduce:: generateProps ()"<<endl;
        std::cerr<<"_Individual_Matrices_must_have_length_'base'"<<endl;
        exit(1);
    }

    if(Foward.size() != base-1){
        std::cerr<<" PropReduce:: generateProps ()"<<endl;
        std::cerr<<"_Foward_Matrices_must_have_length_'base-1'"<<endl;
        exit(1);
    }

    if(FillMe.size() != base){
        std::cerr<<" PropReduce:: generateProps ()"<<endl;
        std::cerr<<"_FillMe_Matrices_must_have_length_'base'"<<endl;
        exit(1);
    }

    if(Back.size() != UseMe+1){
        std::cerr<<" PropReduce:: generateProps ()"<<endl;

```



```

ofstream oo(fname.c_str());

PropReduce myReduce(base, factor, &oo);
myReduce.reduce();

}

```

A.2.3 Optimized static Hamiltonian FID propagation

```

Vector<complex> StaticFID(matrix &H, matrix &rho,
                        matrix &detect, int npts, double dt)
{
  Vector<complex> fid(npts, 0);
  complex z(0.0, -dt*2.0*Pi); //i*2 pi dt
  matrix evect; //eigenvectors of H
  dmatrix eval; //eigenvalues of H

  //diagonalize the Hamiltonian
  diag(H, eval, evect);
  const double cutoff = 1.0e-10;

  // put rho into eigenbase of H
  matrix sig0=adjprop(evect, rho); //adjoint(evect)*rho*(evect);

  // Put detection op. to eigenbase of H
  matrix Do=adjprop(evect, detect); //adjoint(evect)*de*(evect);

  int hs = hamil.rows();
  int ls = hs*hs;
  eval=exp(z*eval); //exp[-i 2 pi t Omega]

  //storage for the coefficients
  complex *A=new complex[ls];
  //storage for the eigenvalue differences
  complex *B=new complex[ls];
  int i, j, pos=0;

  //calculate them, ommiting anything that is '0'
  for(i=0; i<hs; i++){
    for(j=0; j<hs; j++){
      //the shorter matrix trace and matrix multiplication
      //into an N^2 loop rather than an N^3 loop
      A[pos] = Do(i, j)*sig0(j, i);
      //calculate the eigenvalue terms from
      // the multiplication
      B[pos] = eval(i)*conj(eval(j));
      //do not care about the value if the coief
      // is below our cutoff value
      if(square_norm(A[pos])>cutoff){ pos++;}
    }
  }
  //move npts*dt in time
  for(int k=0; k<npts(); k++){

```

```

z = 0; //temporary signal

//this is our reduced
//matrix and trace multiplication
for(int p=0; p<pos; p++){
  //add all the coieff*frequencies
  z += A[p];
  A[p] *= B[p];
}
//assign temp signal to fid
fid(k)=z ;
}
delete [] A;
delete [] B;
return fid;
}

```

A.2.4 γ -COMPUTE C++ Class

```

/*compute.cc
 * this little class develops stroposcopically observed
 * spectra using the 'COMPUTE' method given in

@Article{Eden96,
  author="Mattias Eden and Young K. Lee and Malcolm H. Levitt",
  title="Efficient Simulation of Periodic Problems in NMR.
  Application to Decoupling and Rotational Resonance",
  journal="J. Magn. Reson. A.",
  volume="120",
  pages="56-71",
  year=1996
}

@Article{Hohwy99,
  author="Hohwy, M. and Bildse, H. and Nielsen, N. C.",
  title="Efficient Spectral Simulations in {NMR} of Rotating
  Solids. The  $\gamma$ -COMPUTE Algorithm",
  journal="J. Magn. Reson.",
  volume="136",
  pages="6-14",
  year=1999
}

* it calcualtes a single propogator for some modulation
* period, T. It uses all the little compute_step used to calculate
* propogator to reconstruct the entire frequency range
* and thus a fid from the frequencies
*
* it also calculates propogators via a direct method
* i.e.  $U(t)=\text{Prod}(\exp(-i dt H(t)))$ 

the 'function_t' class MUST have a function called
'hmatrix Hamiltonian(double TIME1, double TIME2, double WR)'

```

where *'TIME1=the begining of a delta T step*
'TIME2=the END of a delta T step
'WR'= the Rotor Speed

The Hamiltonian function Must perform the correct rotation under WR, it is also up to the user to set the correct ROTOR ANGLE BEFORE this is called

It is desgined to be part of the BLOCHLIB tool kit thus the 'BEGIN_BL_NAMESPACE' macro and the 'odd' includes.

**/*

```

#ifndef _compute_h_
#define _compute_h_ 1
#include "container/matrix/matrix.h" //Blochlib file
#include "container/Vector/Vector.h" //Blochlib file

BEGIN_BL_NAMESPACE

template<class function_t>
class compute {
private:
  //storage for U_k
  static Vector<matrix > U_k;

  //a pointer to the hamiltonian function class
  function_t *mf;

  //1 if ro==1/2(det+adjoint(det)), 0=false, 2=not calculated YET
  //calculated via 'isroSYM' below
  int rosym;
  int isroSYM(const matrix &ro,const matrix &det)
  {
    if(rosym==2){
      if(ro==0.5*(det+adjoint(det))) return 1;
      else return 0;
    }else{
      return rosym;
    }
  }
}

//given a period of '1/wr'
// and a desired sweep width 'sw'
// the number 'n' (compute_step) divisions of
// the rotor cycle is floor(sw/wr+0.5)
// as it must be an integer
// thus the sweep width may need to modified
// to accomidate 'n'
//
// This also calculates the number of
// 'gamma' powder angles we can calculate

```

```

// given 'n', should we desire and gamma angles
// computed at all, we alter the 'gammaloop'
// factor to > 1 to perform the reordering
// gamma_step needs to be a mutiple of compute_step
void CalcComputeStep()
{
  if(wr_==0.0) return;
  compute_step=int(floor(sw_/wr_+0.5));
  if(gamma_step>=compute_step){
    gammaloop=gamma_step/compute_step;
  }
  if(gammaloop<1) gammaloop=1;
  gamma_step=gammaloop*compute_step;
  // compute_time=1./(double(compute_step)*wr_);
  sw_=double(compute_step*wr_);
}

public:
//the TOTAL one period propogator
matrix Uf;

//number of rotor divisions
int compute_step;

//total number of gamma angles
// to calculate
int gamma_step;

//total number of reorderings of propogators to calculate
// more gamma_steps (gamma_step=compute_step*gammaloop)
int gammaloop;

//sweep width and rotor speed in Hz
double sw_, wr_;

//start time and end time and step time for
// the period
double tmin;
double tmax;
double tau;

compute();
compute(function_t &);
compute(function_t &in, int compute_stepIn)
compute(function_t &, double wr, double sw, double tmin,
         double tmax);
compute(function_t &, int compute_stepIn, double tmin,
         double tmax);

~compute(){ mf=NULL; }

//functions for internal variables
inline double wr() const

```

```

{ return wr_; }
inline void setWr(double in)
{ wr_=in; CalcComputeStep(); }

inline double sweepWidth() const
{ return sw_; }
inline void setSweepWidth(double in)
{ sw_=in; CalcComputeStep(); }

inline int gammaStep() const
{ return gamma_step; }
inline void setGammaStep(int in)
{
  RunTimeAssert(in >=1);
  gamma_step=in;
  CalcComputeStep();
}

//calculates the U_k propogators
// given no additional reordering
// to compute the gamma angles
void calcUFID(){ calcUFID(1); }
//calculates the U_k propogators
// given the current gamma angle index desired
void calcUFID(int gammaon);

//computes the FID given initial and detection
// matrices and the number of propogator
// points desired
Vector<complex> FID(matrix &ro , matrix &det , int npts);

};

//thje static list of U_k matrices
template<class function_t>
Vector<typename compute<function_t >::matrix>
compute<function_t >::U_k(1 , matrix ());

//default constructor
template<class function_t>
compute<function_t >::compute()
{
  mf=NULL;
  compute_step=0;pmax=0;
  tmin=0.; tmax=0.; tau=0.;
  rosym=2;
  gammaloop=1;
  gamma_step=10;
  wr_=0;
  sw_=0;
}

```

```

//constctor assigns function pointer
template<class function_t>
compute<function_t >::compute(function_t &in)
{
    mf=&in;
    tmin=0.;compute_step=0;
    tmax=0.; tau=0.;pmax=0;
    rosym=2;
    gammaloop=1;
    gamma_step=10;
    wr_=0;
    sw_=0;
}

//constctor assigns function pointer
// and compute_step
template<class function_t>
compute<function_t >::compute(function_t &in , int compute_stepIn)
{
    mf=&in;
    compute_step=compute_stepIn;
    U_k.resize(compute_step+1, mf->Fe());
    Uf=mf->Fe();
    pmax=compute_step+1;
    tmin=0.;tmax=0.; tau=0.;
    rosym=2;
}

template<class function_t>
compute<function_t , MatrixType.T >::compute(
    function_t &in , //function
    double wr, //rotor speed
    double sw, //sweep width
    double tminin , //start time of a period
    double tmaxin) //end time of a period
{
    mf=&in;
    wr_=wr;
    sw_=sw;
    gammaloop=1;
    gamma_step=10;
    CalcComputeStep();
    U_k.resize(compute_step+1, mf->Fe());
    pmax=compute_step+1;
    Uf=mf->Fe();
    tmin=tminin;
    tmax=tmaxin;
    tau=(tmax-tmin)/compute_step;
    if(tau<=0){
        std::cerr<<std::endl<<std::endl
            <<"Error: compute::compute()"<<std::endl;
        std::cerr<<"_your_time_for_the_propgator_is_negative"<<std::endl;
        std::cerr<<"... an evil stench fills the room..."<<std::endl;
    }
}

```



```

    BLEXCEPTION( __FILE__ , __LINE__ )
  }
  rosym=2;
}

template<class function_t>
compute<function_t >::compute(function_t &in , //the function
    int compute_stepin , //initial compute steps
    double tminin , //begining time of the period
    double tmaxin) //the end time of the period
{
  mf=&in;
  compute_step=compute_stepin;
  U_k.resize( compute_step+1, mf->Fe());
  gammaloop=1;
  gamma_step=10;
  Uf=mf->Fe();
  tmin=tminin;
  tmax=tmaxin;
  tau=(tmax-tmin)/compute_step;
  if(tau<=0){
    std::cerr<<std::endl
      <<std::endl<<" Error: _compute:: compute()"<<std::endl;
    std::cerr<<" _your_time_for_the_propogator_is_negative"<<std::endl;
    std::cerr<<" ... an_evil_stench_fills_the_room..."<<std::endl;
    BLEXCEPTION( __FILE__ , __LINE__ )
  }
  rosym=2;
}

//calculate the U_k propogators
template<class function_t>
void compute<function_t >::calcUFID(int gammaon)
{
  //the effecitve 'gamma' angle is
  //performed by 'shifting' time....
  double tadd=PI2*double(gammaon)/
    double(gammaloop*compute_step)/wr_;
  double t1=tmin+tadd;
  double t2=tmin+tadd+tau;
  static matrix hh;

  //loop through the compute step divisions
  // using the 'Hamiltonian(t1, t2, wr) function
  // required in the function_t
  for(int i=0;i<compute_step;i++){
    hh=Mexp(mf->Hamiltonian(t1, t2, wr_), -complex(0,1)*tau*PI2);
    if(i==0){ U_k[0].identity(hh.rows());}
    U_k[i+1]=hh*U_k[i];
    t1+=tau;
    t2+=tau;
  }
}

```

```

}
//total period propogator is the last step
Uf=(U_k[compute_step]);
}

//fid calculation
// needs 1) to loop through all permutations
// of the gammaloop, to shift time properly
// 2) use the propogators to calculate the FID
template<class function_t>
Vector<complex>
  compute<function_t >::FID(matrix &ro , matrix &det , int npts)
{
  Vector<complex> fid(npts , 0);
  for(int q=0;q<gammaloop;q++){
    calcUFID(q);
    fid+=calcFID( ro , det , npts);
  }
  return fid;
}

template<class function_t>
Vector<complex>
  compute<function_t >::calcFID(matrix &ro , matrix &det , int npts)
{
  //zero out a new FID vector
  Vector<complex> fid(npts,0);
  //is ro and det symmetric?
  rosym=isroSYM(ro , det);
  matrix evect; //eigenvectors
  dmatrix eval; //eigenvalues
  //calculate the effective Hamiltonian
  // from the total period propogator
  diag(Uf, eval , evect);
  int N=Uin.rows();
  int i=0, j=0, p=0, r=0, s=0;

  //vector of log(eigenvalues) in H_eff
  Vector<complex> ev(N, 0);
  //the matrix of frequencis differences
  // w_rs
  matrix wrs(N, N);
  double tau2PI=tau;

  //calculate the transition matrix
  complex tott=complex(0. ,double(compute_step)*tau2PI);
  for(i=0;i<N;i++){
    ev[i]=log(eval(i , i));
  }

  for(i=0;i<N;i++){

```

```

    for (j=0;j<N;j++){
        wrs(i,j)=chop((ev[i]-ev[j])/tott, 1e-10);
    }
}

//the gamma-compute algorithm use the symetry relation
// between the gamma powder angles DIVIDED into 2Pi/compute_step
// sections...the detection operator
// so for each gamma angle we would think that we would need
// (compute_step) propogators for each rotor cycle division
// (which we set to be equal to (compute_step) also) to
// corispond to each different gamma angle... well not so,
// becuase we have some nice time symetry between the gamma
// angle and the time eveolved. So we only need to calculate the
// propogators for gamma=0...from this one in select combinations
// we can generate all the (compute_step) propogators from
// the gamma=0 ones we still need to divide our rotor cycle up
// however, also into (compute_step) propogators
// for the remaining notes i will use this labaling convention
//
// pQs_k --> the transformed detection operator for the kth
//           rotor division for a gamma angle of
//           'p'*2Pi/(compute_step)
//
// pRoT --> the transformed density matrix for the for a gamma
//           angle of 'p'*2Pi/(compute_step)
// NOTE:: the rotor division info is contained in the Qs
//
// pU_k --> the unitary trasformation for the ith rotor division
//           for a gamma angle of 'p'*2Pi/(compute_step)...
//           the operators 0U_k were calculated in the function
//           'calcUFID(int)'
//
// the 'pth' one of all of these is realated back to the '0th'
// operator by some series of internal multiplications
//

complex tmp1;
static Vector<matrix> Qs;
Qs.resize(compute_step);
static Vector<matrix> RoT;
RoT.resize(compute_step);

//calculating
// the kth density matrix
// (0RoT)^d=(evect)^d*(0U_k)^d*ro*(0U_k)*evect
// the kth detection op
// (0Qs_k)^d=(evect)^d*(0U_k)^d*ro*(0U_k)*evect
//
//IF ro=1/2(det+adjoint(det)) then
// the ith density matrix is (0RoT)^d=0Qs_k+(0Qs_k)^d
//

```

```

// the '^d' is a adjoint operation

for (i=0;i<compute_step;i++){
  Qs[i]=adjprop(evect ,adjprop(U_k[i+1], det));
  if(rosym==0) RoT[i]=adjprop(evect , adjprop(U_k[i+1], ro));
  else RoT[i]=Qs[i]+adjoint(Qs[i]);
}

//The signal is then a nice sum over the transition matrix
// and the pQs's and pRos Of course this is where we
// manipulate the '0th' operators to create the 'pth'
// and combine them all into a 'f' matrix which
// contains the amplitudes
//
// pF_k(r,s)= means the 'pth' gamma angle for the
//           'kth' rotor division element (r,s)
//
// pF_k(r,s)= exp[i m wrs(r,s) tott] * Ro[p%compute_step](s,r)
//           * Qs_[k+p%compute_step](r,s) exp[-i wrs(r,s) j tott]
//
// here m=int((k+p)/compute_step) - int(p/compute_step)

//of course we have many 'p' sections (or gamma anglers)
// that contribute to the amplitude factors , and becuae
// they are strictly amplitudes for separate gamma anlges , we can
// easily sum them into a total amplitude
//
// Fave_k(r,s)=1/compute_step * Sum_(p=0)^(p=n-1) { pF_k(r,s) }
//           = 1/(compute_step)*Sum(...) { [(p%compute_step)R](s,r)
//           exp(i [p-int(p/compute_step)*
//           compute_step] wrs(r,s) tau)}
//           *0Qs_(k+p%n)(r,s)] exp(-i
//           [j+p-int((j+p)/compute_step))*
//           compute_step] wrs(r,s) tau)

static Vector<matrix> Fave;
Fave.resize(compute_step , mf->Fz());

//amplitude calculating
int ind1 , ind2;
for (i=0;i<compute_step;i++){
  for (p=0;p<compute_step;p++){
    //proper 'p' for Q selection index
    ind1=(i+p)%compute_step;
    if((i+p)>=compute_step){
      ind2=p-compute_step;
    }else{
      ind2=p;
    }
    tmp1=complex(0. , -double(ind2)*tau2PI);
    matrix tmm(N,N);
    for (r=0;r<N;r++){

```

```

    for (s=0;s<N;s++){
        Fave[p](r,s)+=Qs[ind1](r,s)*
            RoT[i](s,r)*
            exp(tmp1*wrs(r,s));
    }
}
}
}
complex tmp=complex(0.,(tau2PI)); //i*tau

//a little computation time save...
// calculate the exp(i*tau*wrs) once
wrs=exp(tmp*wrs);
matrix tmpmm(wrs); //copy w_rs

//the FID at intervals of i*tau is then given by
//
// s(i*tau)=Sum_(r,s) { Fave_i(r,s)*exp[i wrs(r,s) i* tau] }

//here is the j=0 point...saves us a exp calculation

for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        fid[0]+=Fave[0](i,j);
    }
}

for (i=1;i<npts;i++){
//to select the proper 'p' for Fave_p
p=i%compute_step;
for (r=0;r<N;++r){
    for (s=0;s<N;++s){
        fid[i]+=Fave[p](r,s)*tmpmm(r,s);
        //advance 'time' exp(dt*wij)
        tmpmm(r,s)*=wrs(r,s);
    }
}
}
//need to normalize the fid as we have
// added together many 'sub fids'
// but the total should still be 1
int ff=rosym==1?2:1;
fid*=double(1./double(compute_step/ff));
return fid;
}
/**/ END compute CLASS ***/
END_NAMESPACE
#endif

```

A.3 BlochLib Configurations and Sources

A.3.1 *Solid* configuration files

1D static and spinning experiments shown in Figure 5.6

```
# a simple MAS and Static FID collection
spins{
  #the global options}
  numspin 2
  T 1H 0
  T 1H 1
  #csa <iso> <del> <eta> <spin>}
  C 5000 4200 0 0
  C -5000 6012 0.5 1
  #j coupling <iso> <spin1> <spin2>}
  J 400 0 1
}

parameters{
#use a file found with the $BlochLib$ distribution}
  powder{
    #powder file used for the static FID
    aveType ZCW_3.3722
    #powder file used for the spinning FID
    #aveType rep2000
  }
#number of 1D fid points
  npts1D=512
#sweep width}
  sw=40000
}

pulses{
#set the spinning
  wr=0 #set for NON-spinning FID
  #wr=2000 #set for SPINNING FID
#set the rotor}
  rotor=0 #set for NON-spinning fids
  #rotor=acos(1/sqrt(3))*rad2deg #set for SPINNING FID
#set the detection matrix
  detect(Ip)
#set the initial matrix
  ro(Ix)
#no pulses necessary for ro=Ix
#collect the fid
  fid()
  savefidtext(simpStat) #save as a text file
}
```

post-C7 input file for the point-to-point FID in Figure 5.7a

```
#performs a point-to-point C7 (a 1D FID)
```

```
spins{
#the global options
  numspin 2
  T 1H 0
  T 1H 1
  D 1500 0 1
}

parameters{
  powder{
    aveType zcw
    thetaStep 233
    phiStep 144
  }
#the integrator step size
  maxtstep=1e-6
#number of 1D fid points
  npts1D=512
  roeq= Iz
  detect=Iz
}

pulses{
#our post-C7 sub pulse section
  sub1{
    #post C7 pulse amplitude
    amp=7*wr
    amplitude(amp)
    #phase steppers
    stph=0
    phst=360/7
    #pulse times
    t90=1/amp/4
    t270=3/amp/4
    t360=1/amp

    #post C7 loop
    loop(k=1:7)
      1H: pulse(t90 , stph)
      1H: pulse(t360 , stph+180)
      1H: pulse(t270 , stph)
      stph=stph+phst
    end
  }

#a single fid is considered point to point}
  ptop()
#set the spinning
  wr=5000
  rotor=rad2deg*acos(1/sqrt(3))
#can use 'reuse' as the variables
# are set once in our subsection
  reuse(sub1)
```

```

#collect the fid}
fid()
savefidtext(simpC7) #save tas a text file
}

```

post-C7 input file for the 2D FID in Figure 5.7b

```

# performs a 'real' experiment
# for the post-C7 (a series of 2D fids are collected)

spins{
  #the global options
  numspin 2
  T 1H 0
  T 1H 1
  D 1500 0 1
}

parameters{
  powder{
    aveType zcw
    thetaStep 233
    phiStep 144
  }
  #number of 1D fid points
  npts1D=512
}

pulses{

#our post-C7 sub pulse section
sub1{
  #post C7 pulse amplitude
  amp=7*wr
  amplitude(amp)
  #phase steppers
  stph=0
  phst=360/7
  #pulse times
  t90=1/amp/4
  t270=3/amp/4
  t360=1/amp

#post C7 loop}
loop(k=1:7)
  1H: pulse(t90 , stph)
  1H: pulse(t360 , stph+180)
  1H: pulse(t270 , stph)
  stph=stph+phst
end
}

```



```

#number of 2D points
fidpt=128
#collection a matrix of data
2D()
#set the spinning
wr=5000
#the basic rotor angle
rotor=rad2deg*acos(1/sqrt(3))
#set the detection matrix
detect(Ip)
#reset the ro back to the eq
ro(Iz)

#90 time amplitudes
amp=150000
t90=1/amp/4

#loop over the rotor steps
loop(m=0:fidpt-1)

#may use 'reuse' all variables are static in sub1
# must be repeat m times to advance the density matrix
# for each fid (the first fid gets no c7)
reuse(sub1, m)

#pulse the IZ down to the xy plane for detection
1H:pulse(t90, 270, amp)

#collect the fid at the 'mth' position
fid(m)
#reset the ro back to the eq
ro(Iz)
end
savefidmatlab(2dc7) #save the matlab file
}
}

```

A.3.2 Magnetic Field Calculator input file

The input coil type 'Dcircle' is a user registered function, and not part of the normal please view the source code in the distribution for details. par

```

MyCoil{
subcoil1{
type helmholtz
loops 25
amps -4
numpts 4000
R 2
length 3
axis z
}

subcoil2{

```

```

    type Dcircle
    loops 1
    amps 2
    numpts 2000
    R 2
    theta1 0
    theta2 180
    axis z
    center 0, -.6, 5
}

subcoil3{
    type Dcircle
    loops 1
    amps 2
    numpts 2000
    R 2
    theta1 0
    theta2 180
    axis z
    center 0, -.6, -5
}
}

grid{
    min -1,-1,-1
    max 1,1,1
    dim 10,10,10
}

params{
    #which magnetic field section to use
    section MyCoil

    #output text file name
    textout shape.biot
    #output matlab file name
    matout field.mat
}

```

A.3.3 Quantum Mechanical Single Pulse Simulations

A.3.4 Example Classical Simulation of the Bulk Susceptibility

This simulation is a replication of the simulation performed by M. Augustine in Figure 2 of Ref. [100]. It demonstrates the slight offset effect imposed by the magnetization of one spin on another. Both the C++ source using the *BlochLib* framework and the

configuration file is given. Results from this simulation can be found in Figure 5.11.

C++ source

```

#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

/*
This simulates the effect of the Bulk Suseptibility on
a HETCOR experiement... hopefully we shall see several echos
in the indirect dimension

a HETOCR is a 2D experiement

spin1:: 90--t--90-----
spin2:: -----90-FID

*/
timer stopwatch;
void printTime(int nrounds=1){
    std::cout <<std::endl<< "Time_taken:_\"
    << (stopwatch()/nrounds) << "_seconds";
}

void Info(std::string mess)
{
    cout<<mess<<endl;
    cout.flush();
}

int main(int argc, char* argv [])
{
    std::string fn;

    //the parameter file
    query_parameter(argc,argv,1, "Enter_file_to_parse:_\", fn);
    Parameters pset(fn);

    //get the basic parameters
    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");
    double inTemp=pset.getParamD("temperature");
    string spintype1=pset.getParamS("spintype1");
    string spintype2=pset.getParamS("spintype2");
    string detsp=pset.getParamS("detect");

    double moles=pset.getParamD("moles");

    std::string fout=pset.getParamS("fidout");

```

```

coord<int> dims(pset.getParamCoordI("dim"));
coord<> mins(pset.getParamCoordD("min"));
coord<> maxs(pset.getParamCoordD("max"));

std::string dataou=pset.getParamS("trajectories", "", false);

// Grid Set up
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrid;

Info("Creating_grid....");
Grid<UniformGrid> gg(mins, maxs, dims);

Info("Creating_initial_shape....");
TheShape tester;
Info("Creating_total_shape-grid....");
TheGrid jj(gg, tester);

//List BlochParameters
typedef ListBlochParams<
    TheGrid,
    BPOptions::Density | BPOptions::HighField,
    double > MyPars;
int nsp=jj.size();
Info("Creating_entire_spin_parameter_list_for_"
    +itost(nsp)+
    "_spins....");
MyPars mypars(nsp, "1H", jj);
nsp=mypars.size();

//The pulse list for a real pulse on protons..
Info("Creating_real_pulse_lists...");

//get the info from the pset
coord<> pang1=pset.getParamCoordD("pulse1");
coord<> pang2=pset.getParamCoordD("pulse2");
double delaystep=pset.getParamD("delay");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype1, pang1[2]*PI2, pang1[1]*DEG2RAD);
Pulse PP2(spintype1, pang2[2]*PI2, pang2[1]*DEG2RAD);
PP2+=Pulse(spintype2, pang2[2]*PI2, pang2[1]*DEG2RAD);

//get the Bo
double inBo=pset.getParamD("Bo");

Info("Setting_spin_parameter_offsets....");
for(int j=0;j<nsp;j++){
    if(j%2==0){ mypars(j)=spintype1; }
    else{ mypars(j)=spintype2;}

    mypars(j).moles(moles/nsp);

```

```

    mypars(j).Bo(inBo);
    mypars.temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(cout);
PP1.print(cout);
PP2.print(cout);

//Extra interactions
typedef Interactions<
    Offset<>,
    Relax<>,
    BulkSus > MyInteractions;
Info("Setting_Interactions ....");

//the offsets
//get the first offset
double offset1=pset.getParamD("offset1")*PI2;
double offset2=pset.getParamD("offset2")*PI2;
Offset<> myOffs(mypars, offset1);

//Relaxation
double t2s1=pset.getParamD("T2_1");
double t1s1=pset.getParamD("T1_1");
double t2s2=pset.getParamD("T2_2");
double t1s2=pset.getParamD("T1_2");
Relax<> myRels(mypars,
    (!t2s1)?0.0:1.0/t2s1,
    (!t1s1)?0.0:1.0/t1s1);

    for(int i=0;i<nsp;++i){
//set the offsets and relaxation vals
if(i%2==0){
myOffs.offset(i)=offset1;
myRels.T1(i)=(!t1s1)?0.0:1.0/t1s1;
myRels.T2(i)=(!t2s1)?0.0:1.0/t2s1;
}else{
myOffs.offset(i)=offset2;
myRels.T1(i)=(!t1s2)?0.0:1.0/t1s2;
myRels.T2(i)=(!t2s2)?0.0:1.0/t2s2;
}
}
}

//Bulk suseptibility
double D=pset.getParamD("D");
BulkSus myBs(D);

//total interaction obect
MyInteractions MyInts(myOffs, myRels, myBs);

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions > PulseBloch;

```

```

    typedef Bloch< MyPars, NoPulse, MyInteractions > NoPulseBloch;

//second dimension points
    int npts2D=pset.getParamI("npts2D");

//our data matrix
    matrix FIDs(npts2D, nsteps);

//get the time for the 2 90 pulse
    double tpulse1=PP1.timeForAngle(pang1[0]*Pi/180., spintype1);
    double tpulse2=PP2.timeForAngle(pang2[0]*Pi/180., spintype1);

//the time trains this one will always be the same
    Info(" Initializing _Time_train_for_first_Pulse .... ");
    TimeTrain<UniformTimeEngine >
        P1(UniformTimeEngine(0., tpulse1, 10,10));

//loop over all our D values
    for(int kk=0;kk<npts2D;++kk)
    {
        double curDelay=double(kk)*delaystep;
        cout<<"On_delay:_"<<curDelay<<"_"<<kk<<"/"<<npts2D
            <<"\r\n"; cout.flush();

//the time trains for the dealy
        TimeTrain<UniformTimeEngine >
            D1(UniformTimeEngine(tpulse1, tpulse1+curDelay, 10,5));

//the time trains for the dealy
        TimeTrain<UniformTimeEngine >
            P2(UniformTimeEngine(
                tpulse1+curDelay,
                tpulse2+tpulse1+curDelay,
                10,
                10));

        TimeTrain<UniformTimeEngine >
            F1(UniformTimeEngine(
                tpulse2+tpulse1+curDelay,
                tpulse2+tpulse1+curDelay+tf,
                nsteps,
                5));

//This is the 'Bloch' to perform a pulse
        PulseBloch myparspulse(mypars, PP1, MyInts);

//This is the Bloch solver to Collect the FID
//(.i.e. has no pusles...FASTER)
        NoPulseBloch me;
        me=(myparspulse);

//out initial condition
        Vector<coord<>> tm=me.currentMag();

```

```

        stopwatch.reset ();
        BlochSolver<PulseBloch > drivP(myparspulse , tm, "out" );
drivP.setProgressBar(SolverOps::Off);

//integrate the Pulse
drivP.setWritePolicy(SolverOps::Hold);
if(!drivP.solve(P1)){
    Info(" _ERROR!!.. could_not_integrate_pulse_P1.... ");
    return -1;
}

//the fids initial condition is just the previous
// integrations last point
    BlochSolver<NoPulseBloch > driv(me, drivP.lastPoint());
driv.setProgressBar(SolverOps::Off);

//integrate the Delay
driv.setWritePolicy(SolverOps::Hold);
if(!driv.solve(D1)){
    Info(" _ERROR!!.. could_not_integrate_delay_D1.... ");
    return -1;
}

//integrate second the Pulse
drivP.setWritePolicy(SolverOps::Hold);
//set the new pulse set
myparspulse.setPulses(PP2);
drivP.setInitialCondition(driv.lastPoint());
if(!drivP.solve(P2)){
    Info(" _ERROR!!.. could_not_integrate_pulse_P2.... ");
    return -1;
}

//set the detection spin
driv.setDetect(detsp);
//set various data collection policies
driv.setInitialCondition(drivP.lastPoint());
driv.setCollectionPolicy(SolverOps::MagAndFID);
driv.setWritePolicy(SolverOps::Hold);

//integrate the FID
    if(driv.solve(F1)){
        FIDs.putRow(kk, driv.FID());
    }
}

matstream matout(fout , ios::binary | ios::out);
matout.put("vdat" , FIDs);
matout.close();
printTime();
}

```

Input Config File

```
#parameter file for looping through
# several BulkSus parameters

dim 1,1,2
min -0.5,-0.5,-0.5
max 0.5, 0.5, 0.5

#fid pieces
npts 512
tf 8

#the pulse bits
#angle, phase, amplitude
pulse1 90,90,80000
pulse2 90,-90,80000

#the t2 delay
delay 0.000125
npts2D 64

#basic spin parameters
spintype1 1H
spintype2 31P
detect 31P

Bo 4.7
temperature 300
moles .104

#offsets for each spin
offset1 -722
offset2 -4.9

#relaxation params for each spin
T2_1 0.002
T1_1 0
T2_2 0.5
T1_2 0

#for the Bulk Suseptibility
D 1

#file output names for the data
fidout data
```


A.3.5 Example Classical Simulation of the Modulated Demagnetizing Field

This simulation is a replication of the simulation performed by Y.Y Lin in *Science* [87]. It demonstrates the non-linear properties of including both Radiation Damping and the Modulated Demagnetizing field resulting in a resurrection of a completely crushed magnetization. Both the C++ source using the *BlochLib* framework and the configuration file is given. Results of this simulation can be seen in Figure 5.12.

C++ source

```
#include "blochlib.h"

/*
  this is an attempt to imitate the result from YY Lin in
  6 OCTOBER 2000 VOL 290 SCIENCE
  The simulated effective pulse sequence

  RF ---90x---FID
  Grad-----Gzt-----

  where the gradient complete crushes the magnetization
  with some small eps error from the idea
  */

using namespace BlochLib;
using namespace std;

timer stopwatch;
void printTime(int nrounds=1){
    std::cout <<std::endl<< "Time_taken:_ "
                << (stopwatch()/nrounds)
                << "_seconds\n";
}

void Info(std::string mess)
{ std::cout<<mess; std::cout.flush();}

//some typedefs to make typing easier
typedef XYZcylinder TheShape;
typedef XYZshape<TheShape> TheGridS;
typedef GradientGrid<TheGridS > TheGrid;
typedef ListBlochParams< TheGrid ,
                        BPOptions:: Particle | BPOptions:: HighField ,
                        double > MyPars;
```

```

//Extra ineractions
typedef Interactions<Offset<MyPars>,
    Relax<>,
    RadDamp,
    ModulatedDemagField > MyInteractions;

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse , MyInteractions > PulseBloch;
typedef Bloch< MyPars, NoPulse , MyInteractions > NoPulseBloch;

int main(int argc ,char* argv [])
{

//Get all the various parameters
std::string fn;
query_parameter(argc ,argv ,1 , "Enter_file_to_parse:_", fn);
Parameters pset(fn);
double pangl=pset.getParamD("pulseangle1");
double amp=pset.getParamD("pulseamp");

int nsteps=pset.getParamI("npts");
double tf=pset.getParamD("tf");

std::string fout=pset.getParamS("fidout");
std::string magout=pset.getParamS("magout");

int cv=pset.getParamI("lyps", "", false);
std::string lypfile=pset.getParamS("lypout", "", false, "lyps");

std::string dataou=pset.getParamS("trajectories", "", false);

//gradient pars
double gradtime1=pset.getParamD("gradtime1");

/*****/
//Grids
coord<int> dims(pset.getParamCoordI("dim"));
coord<> mins(pset.getParamCoordD("gmin"));
coord<> maxs(pset.getParamCoordD("gmax"));

coord<> smins(pset.getParamCoordD("smin"));
coord<> smaxs(pset.getParamCoordD("smax"));

Info("Creating_grid... \n");
Grid<UniformGrid> gg(mins, maxs, dims);
Info("Creating_initial_shape... \n");
TheShape tester(smins, smaxs);
Info("Creating_total_shape-grid... \n");
TheGridS grids( gg, tester);

//dump the grid to a file

```

```

std::ofstream goo("grid");
goo<<grids<<std::endl;

//create the gradient grids..
char ideal=pset.getParamC("ideal");
coord<> grad=pset.getParamCoordD("grad");

Info("Creating_Gradient_map_grids ... \n");
TheGrid jj(grids);

jj.G(grad);
/*****/

/*****/
//set up Parameter lists
int nsp=jj.size();
Info("Creating_entire_spin_parameter_list_for_"
      +itost(nsp)+
      "_spins ... \n");
MyPars mypars(jj.size(), "1H", jj);
nsp=mypars.size();

double inBo=pset.getParamD("Bo");
double inTemp=pset.getParamD("temperature");
std::string spintype=pset.getParamS("spintype");
double moles=pset.getParamD("moles");
std::string detsp=spintype;

Info("setting_spin_parameter_offsets ... \n");
for (int j=0;j<nsp;j++){
  mypars(j)=spintype;
  mypars(j).Bo(inBo);
  mypars(j).temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(std::cout);
/*****/
//The pulse list for a real pulse on protons..
Info("Creating_real_pulse_lists ... \n");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype, amp, 0.);

PP1.print(std::cout);
double tpulse=PP1.timeForAngle(pangl*Pi/180., spintype);

/*****/
//time train
double tct=0;
Info("Initializing_Time_train_for_first_Pulse ... \n");
TimeTrain<UniformTimeEngine > P1(0., tpulse, 10,100);

```

```

tct+=tpulse;
Info(" Initializing _Time_train_for_First_Gradient_Pulse....\n");
TimeTrain<UniformTimeEngine > G1(tct , tct+gradtime1 , 50 ,100);
tct+=gradtime1;
Info(" Initializing _Time_train_for_FID....\n");
TimeTrain<UniformTimeEngine > F1(tct , tf+tct , nsteps ,5);
if (ideal=='y'){ F1.setBeginTime(0); F1.setEndTime(tf); }

/*****
/*****
//interactions
double t2s=pset.getParamD("T2");
double t1s=pset.getParamD("T1");
double offset=pset.getParamD("offset")*PI2;

//demag field 'time constant'
//because we are in the 'particle' rep
// we need to calculate the real Mo separately
double mo=mypars[0].gamma()*hbar*
        tanh(hbar*PI*(inBo*mypars[0].gamma()/PI2)/kb/inTemp)
        *No*moles*1e6/2.0;
double demag=1.0/(mo*permVac*mypars[0].gamma());

double tr=pset.getParamD("raddamp");

Info(" setting _Interactions....\n");

Offset<MyPars> myOffs(mypars , offset);
Relax<> myRels(mypars , (!t2s)?0.0:1.0/t2s , (!t1s)?0.0:1.0/t1s);
RadDamp RdRun(tr);
ModulatedDemagField DipDip(demag , jj.G());
std::cout<<" Total_Magnetization:_ "<<mo<<std::endl;
std::cout<<DipDip<<" _Td:_ "<<DipDip.td()
        <<" _axis:_ "<<DipDip.direction()<<std::endl;

MyInteractions MyInts(myOffs , myRels , RdRun , DipDip);
demag=pset.getParamD(" demagOff" , "" , false , 0.0);
if (demag!=0) DipDip.off();

/*****
/*****
//This is the 'Bloch' to perform a pulse
Info(" Initializing _total_parameter_list_with_a_pulse....\n");
PulseBloch myparspulse(mypars , PP1 , MyInts);

//This is the Bloch solver to Collect the FID
//(i.e. has no pulses...FASTER)
Info(" Initializing _total_parameter_list_for_FID_collection....\n");
NoPulseBloch me;
me=myparspulse;

Vector<coord<>> tm=me.currentMag();
std::cout<<"TOTAL_mag_initial_condition:_ "<<sum(tm)<<std::endl;

```

```

//the 'error' in the helix
double emp=pset.getParamD("eps", "", false, 1e-3);

//set the circular initialcondition..a single helix
if(ideal=='y'){
  MyPars::iterator myit(mypars);
  double lmax=smaxs.z()-smins.z();
  coord<> tp;
  while(myit){
    tp=myit.Point();
    tm[myit.curpos()].x()=sin(tp.z()/lmax*PI2)+emp;
    tm[myit.curpos()].y()=cos(tp.z()/lmax*PI2);
    tm[myit.curpos()].z()=0.0;
    ++myit;
  }
}
stopwatch.reset();

//the two main solvers
BlochSolver<PulseBloch > drivP(myparspulse, tm);
BlochSolver<NoPulseBloch > drivD(me, tm);

//integrate pulse and gradient pulse
//only if NOT ideal experiment
if(ideal=='n'){
  //output trajectory data if wanted
  if(dataou!=""){
    drivP.setWritePolicy(SolverOps::Continuous);
    drivP.setRawOut(dataou, std::ios::out);
  }else{
    drivP.setWritePolicy(SolverOps::Hold);
  }
  drivP.setCollectionPolicy(SolverOps::FinalPoint);

  //integrate the first pulse
  myOffs.off(); //turn off gradient
  Info("\nIntegrating first Pulse....\n");

  if(!drivP.solve(P1)){
    Info("\nERROR!!... could not integrate pulse P1....\n");
    return -1;
  }

  //integrate the gradient pulse

  Info("\nIntegrating the Gradient Pulse....\n");
  drivD.setInitialCondition(drivP.lastPoint());
  //output trajectory data if wanted
  if(dataou!=""){
    drivD.setWritePolicy(SolverOps::Continuous);
    drivD.setRawOut(dataou, std::ios::app|std::ios::out);
  }
}

```

```

    }else{
      drivD.setWritePolicy(SolverOps::Hold);
    }

    if(gradtime1>0){
      myOffs.on(); //turn on gradient
      if(!drivD.solve(G1)){
        Info(" _ERROR!!.. could not integrate G1.... \n");
        return -1;
      }
    }
  }

//integrate FID
if(cv){
  me.calcVariational();
  drivD.setVariationalInitCond(me.curVariational());
  drivD.setLyapunovPolicy(SolverOps::LypContinous);
  drivD.setLypDataFile(lypfile);
}

myOffs.off();
Info("\nIntegrating for FID ..... \n");

//output trajectory data if wanted
drivD.setCollectionPolicy(SolverOps::MagAndFID);
if(dataou!=""){
  drivD.setWritePolicy(SolverOps::Continuous);
  if(ideal=='y') drivD.setRawOut(dataou, std::ios::out);
  else drivD.setRawOut(dataou, std::ios::app|std::ios::out);
}else{
  drivD.setWritePolicy(SolverOps::Hold);
}

//solve the FID and write it to a file
if(drivD.solve(F1)){
  drivD.writeSpectrum(fout);
  drivD.writeMag(magout);
}
printTime();

//ring a bell when we are done
std::cout<<"\a"<<std::endl;
}

```

Input Config File

```

#parameter file for 1 pulse - 1 Grad Z sequences
#grid units in cm
dim 1,1, 100

```

```
gmin -0.02,-0.02,-0.004693
gmax 0.02, 0.02, 0.004693

#cylinder shape min and max
smin 0,0,-0.004693
smax .003, 6.28, .004693

#fid pieces
npts 512
tf 2

#the pulse bits
pulseangle1 90
pulseamp 80000

#basic spin parameters
Bo 14.1
temperature 300
offset 0
T2 0
T1 0
spintype 1H

#error in ideal gradient pulse
# along the x-axis
eps 1e-3

#turn on(0) or off(1) the demagnetizing field
demagOff 0

#95% water (2 protons a pop)
moles 0.1045

#the extra interactions parts
raddamp 0.01

## #gradient things
#choose 'real gradient'(n) or ideal initial condition(y)
#if ideal magnetization will be spread evenly
#around a circle in the xy plane
ideal y
#non-ideal bits (grad units in Gauss/cm)
grad 0,0,1
gradtime1 0.005

#output data file names
fidout data
magout mag
trajectories traj
```