CHAPTER 7 “YARM”

7.1 Summary

This chapter introduces a computer program named Yet Another Relaxation Matrix, or YARM for short. The purpose of YARM is to simplify the analysis of NOESY crosspeak intensity data by creating a common framework around which data analysis programs can be built. Two general uses of YARM will be presented in this chapter; NMR model verification and NMR model refinement.

NMR model verification: Using the relaxation matrix approach, YARM can calculate a set of simulated NOESY crosspeak intensities for a proposed model. These simulated volumes can then be compared to experimentally measured volumes in a quantitative manner. This gives a statistical measure of the "correctness" of the proposed model by directly comparing it to the NMR derived data.

NMR model refinement: YARM provides a mechanism refining a proposed model. Using a least-squares approach, the NMR derived model can be adjusted until the the simulated NOE volumes from a given model match the experimentally measured volumes.

7.2 Introduction and Background

Determination of NMR derived structures relies heavily on the interpretation of the NOE to determine distances between protons. These interproton distance constraints are then used to to calculate a molecular coordinate set that best meets the requirements of the constraints. Of fundamental importance in the process of structure determination is the method by which one calculates the distance constraint from the measured NOE, a number of approaches have been utilized.

Historically, interpretation of experimentally measured NOE intensities has naturally progressed from very simple methods to more complex. One of the earliest studies employing the NOE in a biomolecular structure determination was by Wagner and Wüthrich in the assignment of bovine pancreatic trypsin inhibitor (1982) in which the NOE connectivities were used only to determine sequence information. Kumar et al. (1981) introduced the concept of using the NOE crosspeak intensity as a method of quantifying distances. They proposed the initial concepts of what would be called the "Isolated Spin Pair Approximation" (Reid, 1987; Patel, et al., 1987; Clore and Gronenborn, 1985; Clore and Gronenborn, 1989), which assumes that any two nuclei are relatively isolated from all other nuclei in NOESY experiments at short mixing times. This allows for a simple method of distance determination by "relative intensity comparison" of the NOE between a spin pair of known distance to that of a spin pair of unknown distance.

$$\frac{\sigma_{ref}}{\sigma_{ij}} = \frac{r_{ij}^6}{r_{ref}^6} \qquad 7.1$$

For nucleic acids this is often accomplished by measuring the cross-relaxation rate of the H5-H6 crosspeak in cytosine or (for RNA only) uridine, with a fixed distance interproton distance of 2.46Å. Often the semi-quantitative method of the ISPA is simplified by defining NOE intensities as strong, medium or weak and giving large bounds to the restraint distances.

Spin isolation is rarely observed in biological systems and the ISPA method of distance determination can give incorrect distance values, with errors of up to 1.3 Å (Wemmer, 1991; Reid, 1989; Nerdal, 1989; Schmitz & James, 1995). To account for "spin-diffusion" effects, Keepers and James (1984) realized that dipolar relaxation

between nuclei must be accounted for simultaneously for all spin pairs. They use the rate (or relaxation) matrix method for analyzing NOE data in the same manner that chemical exchange processes had been done previously.

The concept of using the relaxation matrix in the analysis of NOE intensities has given rise a number of computer programs that exploit this approach for NMR structure determination: IRMA (Boelens, *et al*., 1988), MARDIGRAS (Borgias & James, 1990) and MORASS (Post, *et al*., 1990). All of these methods, however, have made the assumption that the rotational diffusion of the molecule in question can be adequately described using an isotropic rotation model. The isotropic definition of the spectral density function and originally proposed by Bloembergen, Purcell and Pound (1948) is utilized (see Chapter 5, section 5.3.1).

The problem in this assumption has been pointed out a number of times in the literature (Birchall & Lane, 1990; Schmitz & James, 1995). Nucleic acids may be especially affected by this, as was noted by Withka *et al*. (1990) in which they state "The asymmetry of the duplex DNA complicates the straightforward analysis of NOE data in term of conformational analysis." That is, the rotational diffusion of DNA is asymmetric. The consequence of this is that an internuclear spin-pair vector, parallel to the long axis of a DNA, experiences different fluctuating magnetic fields than a vector perpendicular to the long axis. This will cause differential relaxation effects that would affect the NOE intensity differently.

The computer program YARM was initially created to incorporate the anisotropic rotation definition of the spectral density function into the relaxation matrix calculations.

7.3 YARM

The theoretical basis for these calculations is presented in chapter 5 of this thesis and should be consulted, however, a quick overview of the relaxation matrix method will be given. An example will be shown for evaluating the model of a DNA using a few of the more commonly used structural analysis YARM scripts, along with a description of how each script works. A quick overview will also be given for the structural refinement component of YARM.

Finally, for the programmers (and other interested in how the calculations are performed) the two C++ object definition files and the structural refinement program are included for your perusal. The object definition files are the heart and soul of the mathematical calculations, with all the fodder striped away and should be consulted for a complete understanding of the YARM calculations. It should be noted that the entire source code for YARM is too large to include in this thesis. If interested, see the web page at (http://bass.chem.yale.edu/~lapham/yarm/) where the full code can be downloaded.

7.3.1 Overview of simulating NOE initensities

As mentioned earlier, the full theoretical treatment of using the relaxation matrix to perform NOE intensity simulations is presented in chapter 5. This section is simply a broad overview of the process.

The NOE crosspeak volume matrix, $\mathbf{V}(t_{mix})$, is related to the relaxation matrix, **R**, by the following equation,

$$\mathbf{V}(t_{mix}) = \mathbf{V}(0)\exp[\mathbf{R}t_{mix}] \qquad 7.3$$

Thus, if the relaxation matrix can be accurately constructed, the NOE volumes will be accurately calculated. This is, however, the difficult part of the process. The elements in the relaxation matrix are composed of functions that relate the properties of molecular structure, rotational motion and intramolecular motion to the cross-relaxation rates. The first of these properties is the "molecular structure" of the molecule, or the X, Y and Z Cartesian coordinates of the time-averaged position of each atom. The second is the "rotational motion" of the molecule, also commonly referred to as the correlation time. The third is the "intramolecular motion" of the molecule, a description of the .dynamical movements between atoms on the same molecule.

In YARM, we call these three properties the "model" of the molecule. Thus, a "model" of a biomolecule is not just a description of the coordinate structure, it would also require a description of the tumbling rate and the intramolecular dynamics. Figure 7.1 below demonstrates the overall process of calculating NOE volumes from this "model" and where in the calculations each part of the model is used. For instance, the two motional components of the model are used in converting the relaxation matrix **R** to the distance matrix **r**.

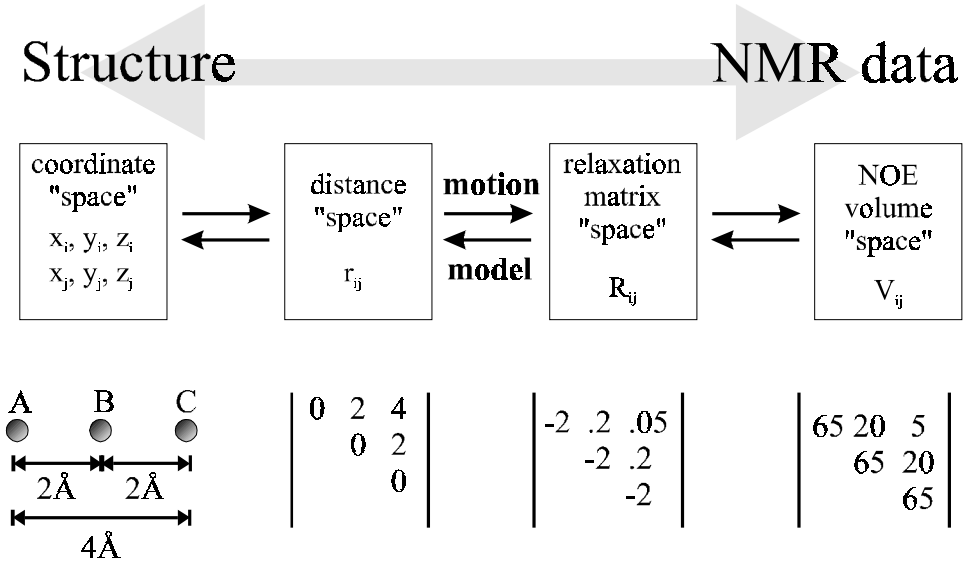

Figure 7. 1 YARM data flowchart

Theoretically, if the three components of the model were perfectly well known, the conversion from coordinate space to NOE volume space would be exactly correct and reversible. This, of course, rarely occurs with experimentally derived data. Often, only a subset of the possible NOE crosspeak volumes are assigned, or are resolved enough for accurate quantitation. Because of this, divining the structure of a molecule based on the NOE intensities is not as straightforward as performing the reverse calculation, shown above.

Often in the world of biomolecular structure determination, assumptions must be made about one or more of the model components. For instance, the rotational correlation time of a molecule is a difficult quantity to measure experimentally, and often it must be estimated. A firm understanding of intramolecular dynamics can be just as elusive; it is often difficult to distinguish between a rigid structure and a two conformation state structure in fast exchange on the NMR time scale. While it may be

difficult to exactly determine these quantities, they cannot be ignored without compromising the integrity of the analysis.

It is my opinion that the biomolecular NMR spectroscopist who proposes a molecular "structure" that "best fits their NMR data" must back the statement up with a statistical analysis. The motional components of the model must be proposed, as they are just as important calculations as the atomic coordinates. This can be as simple as stating "we assume an isotropic rotation model with correlation time of 5 ns and a rigid structure". Even if there is no conclusive data to support this, it must be stated to allow for discussion of the structural model. Finally, a statistical analysis of the actual NOE data measured for the molecule can then be presented. Thus, rather than a qualitative "this structural model fits the data" there can be a quantitative report on how *well* it fits the data.

7.3.2 Statistical analysis of volume sets

A number of statistical methods for comparing NOE volumes have been developed. The YARM subroutine "Stats" returns a list of each of these functions in the order shown below:

```
($rms, $r, $q, $q6) = &Stats( \%vol1, \%vol2);
```

Where the definitions of the statistical functions are:

$$RMS = \left[\frac{\sum_{ij} \left\{ VolExp_{ij} - VolSim_{ij} \right\}^2}{\left\{ \sum_{ij} VolExp_{ij}^2 + \sum_{ij} VolSim_{ij}^2 \right\}} \right]^{\frac{1}{2}} \qquad 7.4$$

$$R-factor(R)=\frac{\sum_{ij}\left|VolExp_{ij}-VolSim_{ij}\right|}{\sum_{ij}VolExp_{ij}} \qquad 7.5$$

$$Q-factor(Q)=\frac{\sum_{ij}\left|VolExp_{ij}-VolSim_{ij}\right|}{\sum_{ij}VolExp_{ij}+\sum_{ij}VolSim_{ij}} \qquad 7.6$$

$$Q^{1/6}-factor(Q^{1/6})=\frac{\sum_{ij}\left|VolExp_{ij}^{1/6}-VolSim_{ij}^{1/6}\right|}{\sum_{ij}VolExp_{ij}^{1/6}+\sum_{ij}VolSim_{ij}^{1/6}} \qquad 7.7$$

7.3.3 Model Validation

An example of how the structural analysis works is presented. The Dickerson dodecomer DNA, 5'-CGCGAATTCGCG-3' is a symmetric self-complementary dimer which has been studied extensively by NMR and X-ray crystallography techniques. NMR NOESY data were collected on the DNA (see chapter 6) and the resolved NOE crosspeak volumes were measured quantitatively, a total of 225 volumes in all.

We begin the analysis by arbitrarily proposing the following two models for the DNA (of course, the structural biologists would want to use the structures derived from their XPLOR calculations, and the like).

Property	MODEL #1	MODEL #2
Atom coordinate positions:	A-form DNA	B-form DNA
Molecular tumbling:	isotropic, 5ns	anisotropic, 2 and 6 ns
Intramolecular dynamics:	rigid	S^2=0.9

The "correctness" of the two models can be examined quantitatively by comparison of the back-calculated NOE intensities to the actual experimental data using the YARM scripts, model1.pl and model2.pl (see section 7.3.3). The script model1.pl

simulates the NOE intensities using the first proposed model, and model2.pl uses the second proposed model. The following is the output from these programs:

```
bass (lapham): [~/yarm_thesis]> ./model1.pl dick_a.pdb
YARM v0.9 February 22, 1998
Simulating NOE volumes using isotropic-rigid...
Pairwise statistical analysis:
           RMS = 0.5128
      R-factor = 0.7508
      Q-factor = 0.3754
Q^(1/6)-factor = 0.1947

bass (lapham): [~/yarm_thesis]> ./model2.pl dick_b.pdb
YARM v0.9 February 22, 1998
Simulating NOE volumes using anisotropic S=0.9...
Principal axis vector components Ax=0.01 Ay=-0.03 Az=1.00
Pairwise statistical analysis:
           RMS = 0.2888
      R-factor = 0.4162
      Q-factor = 0.2081
Q^(1/6)-factor = 0.0882
```

Clearly the second model is a better fit to our experimental data, but we can be more quantitative than that. The second model fits the experimental data with a rms of 0.29, an R-factor of 0.42, a Q-factor of 0.21 and a $Q^{1/6}$-factor of 0.088.

Additionally, the YARM scripts saved a correlation plot of the simulated data versus the experimental data in a file, so the accuracy of the fit can be viewed graphically, as shown below in figure 7.2.

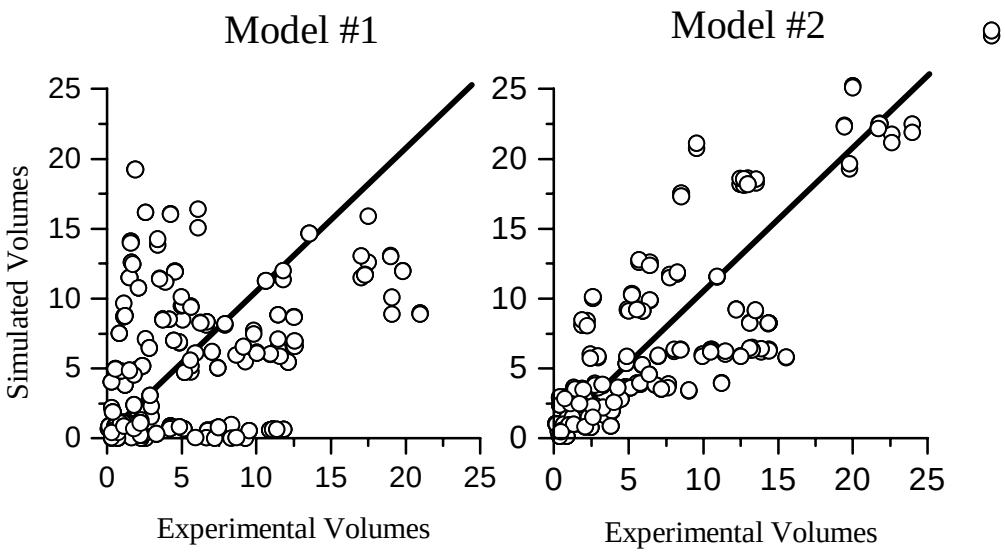

Figure 7. 2 YARM Correlation plots

It is clear that the first model is a better fit to the data, statistically, than the second model. The correlation plots are simply a visual way of coming to the same conclusion, the second model is better correlated to the experimental data.

Incidentally, neither of these models simulate the experimental NOEs very well, the best fit to the data comes from yet another proposed model for this DNA; see chapter 6 for a full discussion.

The model1.pl and model2.pl scripts are presented at the end of this section. Notice that the scripts are written in the Perl scripting language. YARM is actually nothing more then a series of perl subroutines. The first YARM subroutine called in the model1.pl script is:

```
%xyz = &Pdb_Read_All( $pdb_file );
```

The Pdb_Read_All YARM subroutine reads in all the atom names and positions from a PDB formatted structure file. The atom names and coordinates are then stored in the variable %xyz for use in other YARM subroutines. The actual calculation of the simulated NOE volumes comes from the line:

```
%vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tc );
```

The YARM subroutine Sim_Vol simulates volumes! It needs to know the spectrometer frequency ($sfreq), mixing time ($tmix), normalized volume ($vol0), atom names and coordinate (\%xyz), which atom pairs to return (\%rij) and the correlation time ($tc). It then returns to the variable %vol_sim the results of the calculation.

The model2.pl script uses the Sim_Vol subroutine in a slightly different manner:

```
%vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tl, $ts, $Ax, $Ay, $Az, \%S );
```

Notice that the first five arguments are the same as those in model1.pl, but now two correlation times ($tl and $ts), the vector components of the principal axis of rotation ($Ax, $Ay, $Az), and an Order Parameter (\%S) have been included. That is because the "model" for this script uses anisotropic rotation and includes an order parameter of 0.9.

This is just a short description of the scripts. The web page has many more example scripts and more interesting uses of the program. Additionally, each of the subroutines is explained in much greater detail.

Model1.pl example YARM script

```
#!/usr/local/bin/perl

# MODEL #1

require "/usr/local/yarm/yarm_lib.pl";

##################################################
# Define variables
##################################################
$pdb_file = shift;
$corr_file = "$pdb_file.corr";

$sfreq = 600;
$vol0 = 100;
$tmix = 0.2;

$tl = 2.5;
$ts = 6;
$tc = 5;
$S = 0.9;

$f95_vol_file = "d12_30s.vols";
$f95_peak_file = "d12_30s.peaks";

# Set to 0 for no debugging messages
# Set to 1 for a few debugging messages
# set to 2 for TONS of debugging messages (lots!)
$debug = 0;

##################################################
# Call YARM subroutines
##################################################
print "$yarm_version\n";

# Get non-exchangeable nucleic acid protons from a PDB file
%xyz = &Pdb_Read_All( $pdb_file );
%xyz = &Get_Atom_Type( \%xyz, \%nonX_NA );
%xyz = &Pseudo_Methyl(\%xyz);

# Build the order parameter hash
foreach $atom ( keys %xyz ) { $S{$atom} = $S; }


# Measure distances of all atom pairs between
# 0 to 10 angstroms
%rij  = &Rij_Hash( \%xyz, 0, 20 );

# ANISOTROPIC ROTATION
# Calculate the principal axis of rotation vector
# print "Simulating NOE volumes using anisotropic S=$S...\n";
# ( $Ax, $Ay, $Az ) = Principal_Axis( \%xyz );
# printf ("Principal axis vector components Ax=%4.2f Ay=%4.2f
Az=%4.2f\n",
#    $Ax, $Ay, $Az );
# %vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij,
$tl, $ts, $Ax, $Ay, $Az, \%S );

# ISOTROPIC ROTATION
print "Simulating NOE volumes using isotropic-rigid...\n";
%vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tc
);

# Read in experimental volumes
%vol_exp = &F95_Read_Merge( $f95_vol_file, $f95_peak_file );
# Convert the experimental volumes to include segids A and B
# We have to do this b/c this DNA is a symmetric molecule
%vol_exp = &Make_Symm_Molecule( A, B, \%vol_exp );

# Normalize the experimental volumes to the simulated volumes
%vol_exp = &Norm_Hash( \%vol_exp, \%vol_sim );

# Calculate statistics between the experimental and simulated
volume sets
( $rms, $r, $q, $q6 ) = &Stats( \%vol_exp, \%vol_sim );

&Print_Correlation( \%vol_exp, \%vol_exp, \%vol_sim,
$corr_file );

##################################################
# Print a nice report
##################################################
# Print the final "report" rms value
print   "Pairwise statistical analysis:\n";
printf ("           RMS = %5.4f\n", $rms);
printf ("      R-factor = %5.4f\n", $r);
printf ("      Q-factor = %5.4f\n", $q);
printf ("Q^(1/6)-factor = %5.4f\n", $q6);
```

Model2.pl example YARM script

```
#!/usr/local/bin/perl

# MODEL #2

require "/usr/local/yarm/yarm_lib.pl";

##################################################
# Define variables
##################################################
$pdb_file = "dick_b.pdb";
$corr_file = "$pdb_file.corr";

$sfreq = 600;
$vol0 = 100;
$tmix = 0.2;

$tl = 2.5;
$ts = 6;
$tc = 5;
$S = 0.9;

$f95_vol_file = "d12_30s.vols";
$f95_peak_file = "d12_30s.peaks";

# Set to 0 for no debugging messages
# Set to 1 for a few debugging messages
# set to 2 for TONS of debugging messages (lots!)
$debug = 0;

##################################################
# Call YARM subroutines
##################################################
print "$yarm_version\n";

# Get non-exchangeable nucleic acid protons from a PDB file
%xyz = &Pdb_Read_All( $pdb_file );
%xyz = &Get_Atom_Type( \%xyz, \%nonX_NA );
%xyz = &Pseudo_Methyl(\%xyz);

# Build the order parameter hash
foreach $atom ( keys %xyz ) { $S{$atom} = $S; }


# Measure distances of all atom pairs between
# 0 to 10 angstroms
%rij  = &Rij_Hash( \%xyz, 0, 10 );

# ANISOTROPIC ROTATION
# Calculate the principal axis of rotation vector
print "Simulating NOE volumes using anisotropic S=$S...\n";
( $Ax, $Ay, $Az ) = Principal_Axis( \%xyz );
printf ("Principal axis vector components Ax=%4.2f Ay=%4.2f
Az=%4.2f\n",
    $Ax, $Ay, $Az );
%vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tl,
$ts, $Ax, $Ay, $Az, \%S );

# ISOTROPIC ROTATION
# print "Simulating NOE volumes using isotropic-rigid...\n";
# %vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tc
);

# Read in experimental volumes
%vol_exp = &F95_Read_Merge( $f95_vol_file, $f95_peak_file );
# Convert the experimental volumes to include segids A and B
# We have to do this b/c this DNA is a symmetric molecule
%vol_exp = &Make_Symm_Molecule( A, B, \%vol_exp );

# Normalize the experimental volumes to the simulated volumes
%vol_exp = &Norm_Hash( \%vol_exp, \%vol_sim );

# Calculate statistics between the experimental and simulated
volume sets
( $rms, $r, $q, $q6 ) = &Stats( \%vol_exp, \%vol_sim );

&Print_Correlation( \%vol_exp, \%vol_exp, \%vol_sim,
$corr_file );

##################################################
# Print a nice report
##################################################
# Print the final "report" rms value
print   "Pairwise statistical analysis:\n";
printf ("           RMS = %5.4f\n", $rms);
printf ("      R-factor = %5.4f\n", $r);
printf ("      Q-factor = %5.4f\n", $q);
printf ("Q^(1/6)-factor = %5.4f\n", $q6);
```

7.3.4 Model refinement

In addition to model verification, YARM contains a structural refinement component. From a given rotational and intramolecular dynamic model, YARM will find the set of Cartesian coordinates that best fit the NOE data.

We ask the simple question: Does the comparison of the simulated and experimental NOEs suggest that an atom pair move closer together, or farther apart? If an atom pair A and B have an experimentally determined NOE volume of 20 and a simulated NOE volume of 10, the two atoms in the model should be moved closer together. The distance they should move will be roughly proportional to the difference in the 1/6 root of the volumes. This is known as the residual function, r:

$$residual_{ij} = VolSim_{ij}^{1/6} - VolExp_{ij}^{1/6} \qquad 7.8$$

The goal of this structure refinement process is to minimizing this function for all atom pairs. The direction each atom should be moved in order to minimize all atom pair interactions is determined by taking the vector sum of all residuals for each individual atom. This overall movement vector is known as the gradient, and is shown below as the thicker line (the vector sum of the thinner lines).

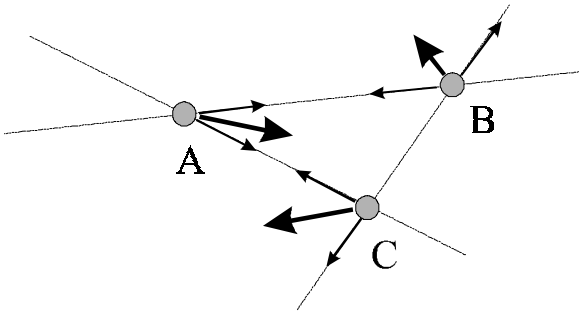

Figure 7. 3 The gradient vector

Movement along the gradient will result in a minimizing of the function defined in equation 7.8.

An example YARM script for model refinement is presented on the next page. The subroutine call that actually performs the calculations is:

```
%xyz2 = &Structure_Refine( \%xyz, \%vol_exp, $num_iter, $sfreq, $tmix, $vol0, $tc );
```

In which the &Structure_Refine subroutine returns a new coordinate hash (in this case, named %xyz2) that can be used as any other coordinate hash in YARM.

Model refinement using YARM

```
#!/usr/local/bin/perl
# yarm Yet Another Relaxation Matrix program

# Read in the library file:
require "/usr/local/yarm/yarm_lib.pl";

################################################
# Define variables
################################################
# Starting structure
$pdb_file = "dick_a.pdb";

$peak_file = "d12_30s.peaks";
$vol_file = "d12_30s.vols";

# NMR relaxation paramters
$tl = 3;
$ts = 6;
$tc = 4;
$sfreq = 600;
$tmix = 0.2;
$vol0 = 100;

## Refinement variables ##
# Number of refinement iterations to perform
$num_iter = 3;
# Min and max for XPLOR distance restraint file
$min = 0.1; $max = 0.1;
# Name of XPLOR distance restraint file
$xplor_file = "noe.dat";

################################################
# Begin YARM structure refinement process
################################################
print "$yarm_version\n";

# Get non-exchangeable nucleic acid protons from a PDB file
# STARTING STRUCTURE FOR REFINEMENT
%xyz = &Pdb_Read_All( $pdb_file );
%xyz = &Get_Atom_Type( \%xyz, \%nonX_NA );
%xyz = &Pseudo_Methyl(\%xyz);
```

```
# Read in experimental volumes
%vol_exp = &F95_Read_Merge( $vol_file, $peak_file );
%vol_exp = &Make_Symm_Molecule( A, B, \%vol_exp );

# B-form volumes (just for normalization)
%rij = Rij_Hash( \%xyz );

# Must simulate volumes in order to normalize the experimental
values
# ANISOTROPIC SIMULATION
( $Ax, $Ay, $Az ) = &Principal_Axis( \%xyz );
print "Simulating NOE volumes using anisotropic-rigid
model\n";
print "Ax=$Ax Ay=$Ay Az=$Az\n";
%vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tl,
$ts, $Ax, $Ay, $Az );

# ISOTROPIC SIMULATION
# print "Simulating NOE volumes using isotropic-rigid
model\n";
# %vol_sim = &Sim_Vol( $sfreq, $tmix, $vol0, \%xyz, \%rij, $tc
);

# Normalize the experimental volumes to the simulated
%vol_exp = &Norm_Hash( \%vol_exp, \%vol_sim );

# Calculate new coordinate set %xyz2
# %xyz2 = &Structure_Refine( \%xyz, \%vol_exp, $num_iter,
$sfreq, $tmix, $vol0, $tl, $ts, $Ax, $Ay, $Az );
%xyz2 = &Structure_Refine( \%xyz, \%vol_exp, $num_iter,
$sfreq, $tmix, $vol0, $tc );

%rij2 = Rij_Hash( \%xyz2 );

%rij2 = Fix_Rij( \%rij2 );

&Xplor_Write2( \%vol_exp, \%rij2, $min, $max, $xplor_file );
```

7.3.5 Other software packages

As the name of this program implies, there are a number of computer programs available that calculate the relaxation matrix. This begs the question, why write another? I am glad you asked, because I would like to tell you why. This software package was written with the express intention that people can use it to LEARN about calculating the relaxation matrix. It would appear to this author that often the details of HOW calculations are performed are hidden from the end users. Hopefully, it will be clear what parts of the calculation are robust and what parts involve a certain level of assumptions. Great pain have been put forth to separate the code into its constituent parts, for example, if you want to know the mathematics behind calculating a cross relaxation rate member of the relaxation matrix, simply look in the **nmr_relax.c** file under the subroutine "rij2rate_iso". In fact, this is the file in which most of the real calculations are performed. This code is completely removed from the code that manipulates the input and output files, etc.

A quick overview of two other programs available for calculating the relaxation matrix are presented here. It should be stated that it is not with the intent of supplanting the existing rate matrix calculation software that YARM was written. Rather, it is the intention of the author that they are used for the development of new ideas, which require the "relaxation matrix" framework around which to work.

MORASS, Multispin Overhauser Relaxation AnalysiS (Post, et al., 1990; Meadows, et al., 1994) was used initially to understand how one codes these types of programs. The authors kindly release their FORTRAN code with the program at no charge (anonymous ftp dggp12.chem.purdue.edu). This program suffers from the usual

problem that all FORTRAN programs suffer from, obscure code. While the authors do make the code publically available, it is nearly impossible to follow the data flow and actually know HOW the calculations are performed. The program also suffers from the "problem" of being a complete software package, it is difficult to integrate it into other calculations or even to modify it.

Another program, Matrix Analysis of Relaxation for DIscerning the Geometry of an Aqueous Structure or MARDIGRAS (Borgias and James, 1990), can be purchased from the regents of the University of California. Professor Thomas James was kind enough to supply the code for this program for the purposes of recompiling it for the LINUX operating system. This is mentioned here because it should be stated that none of the code was examined or used in the creating of the programs written in this section. MARDIGRAS is, once again, not available for free and the FORTRAN source code is not available. Thus, it must be used as a "black box" in which you must trust is performing the calculations correctly. As with MORASS, it is difficult to incorporate into other calculations and impossible to modify.

7.3.6 Source code: nmr_relax.c and nmr_relax.h

These two files define the object "NmrParams". It is in this object definition that all the NMR relaxation calculations take place.

Nmr_relax.h C++ header file

```
// NMR_RELAX.H
// class definitions for nmr relaxation calculations
// Jon Lapham <lapham@tecate.chem.yale.edu>

#ifndef NMR_RELAX_H
#define NMR_RELAX_H

#include "structure.h"

class NmrParams {
public:
    // default constructor and destructor
    NmrParams();
    ~NmrParams();

    // set functions
    void setNmrParams( float, float, float, float, float, float
);
    void setTc( float );
    void setTl( float );
    void setTs( float );
    void setVol0( float );
    void setTmix( float );
    void setSfreq( float );

    // get functions
    float getTc() const;
    float getTl() const;
    float getTs() const;
    float getVol0() const;
    float getTmix() const;
    float getSfreq() const;
    float getW0AB() const;
    float getW1AB() const;
    float getW2AB() const;
    float getW1AA() const;
    float getW2AA() const;

    // calculation functions
    void calcTransIso();                    // Calculate the
isotropic rates
    void calcTransAniso( float, float );    // Calculate the
anisotropic rates
    double rij2rho_iso( int, Structure * );
    double rij2rho_aniso( int, NmrParams, Structure * );
    double rij2sigma( int, int, double );
    double sigma2rij( int, int, int, int, double );

    // print functions
    void printNmrParams() const;
    void printTransitionRates() const;

private:
    float W0AB, W1AB, W2AB;  // Transition rates
    float W1AA;              // Self dipole transition rate
(not implemented)
    float W2AA;              // Self dipole transition rate
(not implemented)
    float tc;                // Iso correlation time (ns)
    float tl;                // Aniso long axis correlation
time (ns)
    float ts;                // Aniso short axis correlation
time (ns)
    float sfreq;             // Spectrometer frequency (MHz)
    float tmix;              // NOE mixing time (s)
    float vol0;              // Normalized autopeak volume at
tmix=0
};

/*****************************************
 NMR relaxation subroutines available
*****************************************/

int rate2rij_iso( NmrParams, double **, Structure * );
int rij2rate_iso( NmrParams, Structure *, double ** );
int rij2rate_aniso( NmrParams, Structure *, double ** );
int vol2rate( int, NmrParams, double **, double ** );
int rate2vol( int, NmrParams, double **, double ** );

#endif
```

Nmr_relax.c C++ source code (NmrParams object definition)

```
// NMR_RELAX.C
// member function definitions for the NmrParams class
// Jon Lapham <lapham@tecate.chem.yale.edu>

#include <iostream.h>
#include <fstream.h>
#include <math.h>

#include "defs.h"               // PI and Q
#include "nmr_relax.h"          // Access the NMR relaxation
object
#include "myalloc.h"
#include "lapack_functions.h"

/************************************************************
    NmrParams Class
************************************************************/
// NmrParams default constructor, initializes all data members
to zero
NmrParams::NmrParams()
{
    W0AB = W1AB = W2AB = W1AA = W2AA = tc = tl = ts = 0;
    sfreq = tmix = vol0 = 0;
}

// NmrParams default destructor
NmrParams::~NmrParams()
{
    W0AB = W1AB = W2AB = W1AA = W2AA = tc = tl = ts = 0;
    sfreq = tmix = vol0 = 0;
}

// Set the NmrParams all at once
void NmrParams::setNmrParams( float tc_in, float tl_in, float
ts_in, float tmix_in, float sfreq_in, float vol0_in )
{
    tc = tc_in;
    tl = tl_in;
    ts = ts_in;
    sfreq = sfreq_in;
    tmix = tmix_in;
    vol0 = vol0_in;
}

// Set NmrParams individually
void NmrParams::setTc( float tc_in ) { tc = tc_in; }
void NmrParams::setTl( float tl_in ) { tl = tl_in; }
void NmrParams::setTs( float ts_in ) { ts = ts_in; }
void NmrParams::setSfreq( float sfreq_in ) { sfreq = sfreq_in;
}
void NmrParams::setTmix( float tmix_in ) { tmix = tmix_in; }
void NmrParams::setVol0( float vol0_in ) { vol0 = vol0_in; }

// Get NmrParams individually
float NmrParams::getTc() const { return tc; }
float NmrParams::getTl() const { return tl; }
float NmrParams::getTs() const { return ts; }
float NmrParams::getSfreq() const { return sfreq; }
float NmrParams::getTmix() const { return tmix; }
float NmrParams::getVol0() const { return vol0; }
float NmrParams::getW0AB() const { return W0AB; }
float NmrParams::getW1AB() const { return W1AB; }
float NmrParams::getW2AB() const { return W2AB; }
float NmrParams::getW1AA() const { return W1AA; }
float NmrParams::getW2AA() const { return W2AA; }

// Calculate the transition rates using the isotropic rotation
model
void NmrParams::calcTransIso()
{
    /****
     Calculate the isotropic transition rates, W0, W1 and W2
    ****/

    double sfreq_rad = (sfreq / 1000) * (180 / PI); // GHz

    // Use the isotropic definition of the spectral density fns
    double J0 = 2 * tc;
    double J1 = 2 * tc / (1 + pow((sfreq_rad * tc), 2) );
    double J2 = 2 * tc / (1 + pow((2*sfreq_rad * tc), 2) );

    W0AB =  0.5 * Q * J0;
    W1AB = 0.75 * Q * J1;
    W2AB =    3 * Q * J2;

    // These calculation have not been implemented yet
```

```
    W1AA = 0;
    W2AA = 0;
}

// Calculate the transition rates using the anisotropic
rotation model
void NmrParams::calcTransAniso( float beta, float S2 )
{
    /*********************************************
    Calculates the W0AB, W1AB and W2AB transition rates using
    an anisotropic rotation model for the spectral density
    function
    beta = angle WRT principal axis of rotation
    S2 = order parameter
    *********************************************/
    // Declare local variables
    double a1, a2, a3;
    double t1, t2, t3;
    double j1, j2, j3;
    double J0, J1, J2;

    double beta_rad  = beta * ( 2 * PI / 360 );
    double sfreq_rad = (sfreq / 1000) * ( 2 * PI / 1 ); //
sfreq_rad in GHz

    // Calculate the angle dependent coefficients a1, a2 and a3
    a1 = 0.25 * pow( (3 * pow(cos(beta_rad), 2) - 1 ), 2);
    a2 = 3.00 * pow(cos(beta_rad), 2) * pow(sin(beta_rad), 2);
    a3 = 0.75 * pow(sin(beta_rad), 4);

    // Calculate the t1, t2 and t3 values
    t1 = tl;
    t2 = 6 * tl * ts / (tl + (5 * ts));
    t3 = 3 * tl * ts / (ts + (2 * tl));

    // Calculate J0
    j1 = 2 * t1;
    j2 = 2 * t2;
    j3 = 2 * t3;
    J0 = a1 * j1 + a2 * j2 + a3 * j3;

    // Calculate J1
    j1 = 2 * t1 / (1 + pow((sfreq_rad * t1), 2) );
    j2 = 2 * t2 / (1 + pow((sfreq_rad * t2), 2) );
    j3 = 2 * t3 / (1 + pow((sfreq_rad * t3), 2) );
```

```
    J1 = a1 * j1 + a2 * j2 + a3 * j3;

    // Calculate J2
    j1 = 2 * t1 / (1 + pow((2 * sfreq_rad * t1), 2) );
    j2 = 2 * t2 / (1 + pow((2 * sfreq_rad * t2), 2) );
    j3 = 2 * t3 / (1 + pow((2 * sfreq_rad * t3), 2) );
    J2 = a1 * j1 + a2 * j2 + a3 * j3;

    // Transition rates (sans the r^-6 component)
    W0AB =  0.5 * Q * J0 * S2;
    W1AB = 0.75 * Q * J1 * S2;
    W2AB =    3 * Q * J2 * S2;


    /*
    cout << "Beta=" << beta << " Beta_rad=" << beta_rad << "
S2=" << S2 << endl;
    cout << "Sfreq=" << sfreq << " Sfreq_rad=" << sfreq_rad <<
endl;
    cout << "tl=" << tl << " ts=" << ts << endl;
    cout << "a1=" << a1 << " a2=" << a2 << " a3=" << a3 <<
endl;
    cout << "t1=" << t1 << " t2=" << t2 << " t3=" << t3 <<
endl;
    cout << "J0=" << J0 << " J1=" << J1 << " J2=" << J2 <<
endl;
    cout << "W0=" << W0 << " W1=" << W1 << " W2=" << W2 << endl
<< endl;
    */
}

// Print a current listing of the NmrParams "user adjustable
data members"
void NmrParams::printNmrParams() const
{
    cout << " Current values for the NMR Parameters" << endl;
    cout << " tc=" << tc << " tl=" << tl << " ts=" << ts << "
tmix=";
    cout << tmix  << " sfreq=" << sfreq << " vol0=" << vol0 <<
endl << endl;
}

// Print a current listing of the NmrParams transition rate
data members
void NmrParams::printTransitionRates() const
```

```
{
    cout << " Current values for the Transition Rates" << endl;
    cout << " W0AB=" << W0AB << " W1AB=" << W1AB << " W2AB=" <<
W2AB;
    cout << " W1AA=" << W1AA << " W2AA=" << W2AA << endl <<
endl;
}

double NmrParams::rij2rho_iso( int i, Structure *XYZptr )
{
    /*******
      Calculate the T1-relaxation rate, rho
        i is the current i atom number
        n[i] and n[j] are the number of equivalent atoms for i,
j
        rij is the distance between the i,j atom pair
      This calculation is only valid for Rigid Isotropic motion
model

      rho is defined for two spins, A and B as,
      rho_A = 2(nA - 1)(W1AA - W2AA) + nB(W0AB + 2W1AB + W2AB)
+ rho_ext
      for more than two spins, sum up all nB(W0AB + 2W1AB +
W2AB) interactions
    *****/

    // Declare local variables
    int j;
    int N = XYZptr->getN();
    int nj;
    double rij;
    double rho = 0;
    double leakage = 0;  // Not supported yet, what should it
be?

    // Initially, calculate the self dipole contribution for
    // equivalent atoms and the leakage rate
    rho = 2*(XYZptr->getn(i) - 1) * (W1AA + W2AA) + leakage;

    // Look at every i j pair, sum up the rho
    for (j=0; j<N; j++) {
        // But not at i=j atom pair
        if ( j != i ) {
            nj = XYZptr->getn(j);
            rij = XYZptr->getRij(i,j);

            // Check to make sure the atoms are not
UNPHYSICALLY close
            if ( rij < 1 ) rij = 1;

            rho += ( -1 * nj * pow( rij, -6 ) *
                (W0AB + 2*W1AB + W2AB) );
        }

    }

    // Debugging output
    // cout << rho << endl;
    return (rho);
}

double NmrParams::rij2rho_aniso( int i, NmrParams NMR,
Structure *XYZptr )
{
    /*******
      Calculate the T1-relaxation rate, rho
        i is the current i atom number
        n[i] and n[j] are the number of equivalent atoms for i,
j
        rij is the distance between the i,j atom pair
      This calculation is only valid for Rigid Anisotropic
motion model

      rho is defined for two spins, A and B as,
      rho_A = 2(nA - 1)(W1AA - W2AA) + nB(W0AB + 2W1AB + W2AB)
+ rho_ext
      for more than two spins, sum up all nB(W0AB + 2W1AB +
W2AB) interactions
    *****/

    // Declare local variables
    int j;
    int nj;
    double rij;
    float S2;    // order parameter
    int N = XYZptr->getN();
    double rho = 0;
    double leakage = 0;  // Not supported yet, what should it
be?
```

```
    // Initially, calculate the self dipole contribution for
    // equivalent atoms and the leakage rate
    rho = 2*(XYZptr->getn(i) - 1) * (NMR.getW1AA() +
NMR.getW2AA()) + leakage;

    // Look at every jth atom, sum up the rho
    for (j=0; j<N; j++) {
        // But not at j=i atom pair
        if ( j != i ) {
            // Recalculate the Transition rates at this beta
            S2 = XYZptr->getS(i) * XYZptr->getS(j);
            NMR.calcTransAniso( XYZptr->getBeta( i, j ), S2 );
            nj = XYZptr->getn(j);
            rij = XYZptr->getRij(i,j);
            if (rij < 1) rij = 1;

            rho += ( -1 * nj * pow(rij, -6) *
                (NMR.getW0AB() + 2*NMR.getW1AB() +
NMR.getW2AB()) );
        }

    }

    // Debugging output
    // cout << rho << endl;

    return (rho);
}

double NmrParams::rij2sigma( int ni, int nj, double rij )
{
    /*******
      Calculate the cross-relaxation rate, sigma
        i, j are the current atom numbers
        n[i] and n[j] are the number of equivalent atoms for i,
j
        rij is the distance between the atoms
      This calculation should be valid for all rigid body
        motion models (such as Rigid Isotropic and Rigid
        Anisotropic)
    *******/

    double sigma;

    // Check to make sure the atoms are not UNPHYSICALLY close
```

```
    if ( rij < 1 ) rij = 1;

    sigma = ( ni * nj * pow(rij, -6) * (W0AB - W2AB) );

    // Debugging output
    // cout << "sigma=" << sigma << " W0AB=" << W0AB << "
W2AB=" << W2AB << endl;
    return sigma;
}

double NmrParams::sigma2rij( int i, int j, int ni, int nj,
double sigma )
{
    /*******
      Calculate the distance between i,j, rij
        i, j are the current atom numbers
        n[i] and n[j] are the number of equivalent atoms for i,
j
        sigma is the cross-relaxation rate between the two
atoms
      This calculation should be valid for all rigid body
        motion models (such as Rigid Isotropic and Rigid
        Anisotropic)
    *******/

    double rij6;
    double rij;

    // Calculate rij, notice that the rate
    // is multiplied by the numbers of equivalent atoms
    // for each i j atom (ie: for a methyl, n=3)
    rij6 = ( ni * nj * (W0AB - W2AB) ) / sigma;
    rij = pow(rij6, 1.0/6.0);

    // Print debugging
    // cout << rij << " " << rij << endl;
    return rij;
}

/*****************************************
 NMR relaxation subroutines

  double **vol2rate( int N, NmrParam NMR, double **VOL )
  double **rate2vol( int N, NmrParam NMR, double **RATE )
```

```
  void rate2rij_iso( NmrParams NMR, double **RATE, Structure
XYZ )
  double **rij2rate_iso( NmrParams NMR, Structure XYZ )

*****************************************/

int vol2rate( int N, NmrParams NMR, double **VOL, double
**RATE )
{
    int i, j, error;

    // We are going to need a bunch of temporary matrices
    // Allocate them dynamically
    double **VOLEVAL = NEW2D_D( N, N );
    double **VOLEVEC = NEW2D_D( N, N );
    double **VOLEVECINV = NEW2D_D( N, N );
    double **VOLEVAL_LN = NEW2D_D( N, N );
    double **MATTEMP = NEW2D_D( N, N );

    // Diagonalize the Volume matrix, giving VolumeEvecs and
VolumeEvals
    if ( lapack_eigen_symm( N, VOL, VOLEVEC, VOLEVAL ) != 0 )
return (1);

    /*  DEBUG PRINT EVALS TO FILE NAMED "temp.vol.evals" */
    ofstream evals_out;
    evals_out.open("temp.vol.evals");
    for (i=0; i<N; i++) {
        evals_out << i << " " << VOLEVAL[i][i] << endl;

    }
    evals_out.close;
    /*******************************************************/

    // Invert the eigenvector matrix, giving VolumeInvEvecs
    if ( lapack_inverse( N, VOLEVEC, VOLEVECINV ) != 0 )
return(1);

    // Calculate the ln( V/V0 ) part, placing the result into
LnVolEvals
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (i == j) {
                // These are the diagonal terms
                if ( VOLEVAL[i][j] > 0 ) {
                    VOLEVAL_LN[i][j] = log (VOLEVAL[i][j] /
NMR.getVol0() );
                }
                else {
                    cout << "vol2rate: ERROR: negative
eigenvalue!!!  Arbitrarily set to ZERO!!" << endl;
                    VOLEVAL_LN[i][j] = 0;
                }
            }
            else {
                // These are off-diagonal terms
                VOLEVAL_LN[i][j] = 0;
            }
        }
    }

    // Now we must recast the matrix LnVolEvals back to the
original
    // basis set using the eigenvector and inverse eigenvector
matrices
    // Rate = VolumeEvecs * LnVolEvals * VolumeInvEvecs

    if ( lapack_mat_mul( N, N, N, VOLEVEC, VOLEVAL_LN, MATTEMP
) != 0) return (1);
    if ( lapack_mat_mul( N, N, N, MATTEMP, VOLEVECINV, RATE )
!= 0) return (1);

    // Free unused memory
    DELETE2D_D( VOLEVAL );
    DELETE2D_D( MATTEMP );
    DELETE2D_D( VOLEVEC );
    DELETE2D_D( VOLEVECINV );
    DELETE2D_D( VOLEVAL_LN );

    // Divide by the mixing time
    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            RATE[i][j] = RATE[i][j] / NMR.getTmix();
        }
    }

    // 0=no error
    return ( 0 );
}
```

```
int rate2vol( int N, NmrParams NMR, double **RATE, double
**VOL )
{
    int i, j, error;

    // We are going to need a bunch of temporary matrices
    // Allocate them dynamically
    double **RATEEVAL = NEW2D_D( N, N );
    double **RATEEVEC = NEW2D_D( N, N );
    double **RATEEVECINV = NEW2D_D( N, N );
    double **RATEEVAL_EXP = NEW2D_D( N, N );
    double **MATTEMP = NEW2D_D( N, N );

    // Diagonalize the RATE matrix, giving RATEEVEC and
RATEEVAL
    if ( lapack_eigen_symm( N, RATE, RATEEVEC, RATEEVAL ) != 0
) return (1);

    /*  DEBUG PRINT EVALS TO FILE NAMED "temp.rate.evals" */
    ofstream evals_out;
    evals_out.open("temp.rate.evals");
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (i == j) {
                evals_out << i << " " << RATEEVAL[i][j] <<
endl;
            }
        }
    }
    evals_out.close;
    /*******************************************************/

    // Invert the eigenvector matrix, giving RATEEVECINV
    if ( lapack_inverse( N, RATEEVEC, RATEEVECINV ) != 0 )
return (1);

    // cout << "NMR.tmix = " << NMR.getTmix() << " NMR.vol0 = "
<< NMR.getVol0() << endl;

    // Calculate the "exp (RATEEVAL * tmix)" manually
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (i == j) {
                RATEEVAL_EXP[i][j] = ( exp (RATEEVAL[i][j] *
NMR.getTmix()) ) * NMR.getVol0();
            }
            else {
                RATEEVAL_EXP[i][j] = 0;
            }
        }
    }

    // Volumes = RATEEVEC * RATEEVAL_EXP * RATEEVECINV
    if ( lapack_mat_mul( N, N, N, RATEEVEC, RATEEVAL_EXP,
MATTEMP ) != 0) return(1);
    if ( lapack_mat_mul( N, N, N, MATTEMP, RATEEVECINV, VOL )
!= 0) return( 1 );

    // Make sure that the Evecs x InvEvecs equals the unity
matrix
    // if ( lapack_mat_mul( N, N, N, RATEEVEC, RATEEVECINV, VOL
) != 0) return( 1 );

    // Free up memory
    DELETE2D_D( RATEEVAL );
    DELETE2D_D( RATEEVEC );
    DELETE2D_D( RATEEVECINV );
    DELETE2D_D( RATEEVAL_EXP );
    DELETE2D_D( MATTEMP );

    // Error checking, 0=no error
    return( 0 );
}

int rate2rij_iso( NmrParams NMR, double **RATE, Structure
*XYZptr )
{
    /*****
      Calculates the rij matrix in a structure object
      from a rate matrix
    *****/

    int i, j;
    double rate;  // Temporary storage of RATE[i][j]
    int N = XYZptr->getN();

    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            rate = RATE[i][j];
```

```
            if (i == j) {
                // No need to calculate rij
                XYZptr->setRij(i, j, 0);
            }
            else {
                // rij comes from sigma2rij
                XYZptr->setRij(i, j, NMR.sigma2rij( i, j,
XYZptr->getn(i), XYZptr->getn(j), rate ) );
            }
        }
    }
    return( 0 );
}

int rij2rate_iso( NmrParams NMR, Structure *XYZptr, double
**RATE )
{
    /****
     Calculates a rate matrix from a rij matrix using the Rigid
Isotropic
     motion model
    ****/

    int i, j;
    int ni, nj;
    int N = XYZptr->getN();
    double rij;

    for (i=0; i<N; i++) {
        // Do a rho calculation, i=j,
        // The NmrParams OBJECT knows how to do this
        RATE[i][i] = NMR.rij2rho_iso( i, XYZptr );

        for (j=i+1; j<N; j++) {
            // Do a sigma calculation
            // The NmrParams OBJECTS know how to do this
            ni = XYZptr->getn(i);
            nj = XYZptr->getn(j);
            rij = XYZptr->getRij(i,j);
            // cout << "rij " << i << " " << j << " =" << rij
<< endl;
            RATE[i][j] = NMR.rij2sigma( ni, nj, rij );
        }
    }
```

```
    // return 0=no error
    return ( 0 );
}

int rij2rate_aniso( NmrParams NMR, Structure *XYZptr, double
**RATE )
{
    /****
     Calculates a rate matrix from a rij matrix using the Rigid
Anisotropic
     motion model
    ****/

    int i, j;
    int N = XYZptr->getN();
    double rij;
    float S2;   // order parameter

    for (i=0; i<N; i++) {
        // Do a rho calculation, i=j,
        // The NmrParams OBJECT knows how to do this
        RATE[i][i] = NMR.rij2rho_aniso( i, NMR, XYZptr );

        for (j=i+1; j<N; j++) {
            // Recalculate the transition rates for each
specific beta angle
            S2 = XYZptr->getS( i ) * XYZptr->getS( j );
            NMR.calcTransAniso( XYZptr->getBeta(i, j), S2 );

            // Do a sigma calculation
            // The NmrParams OBJECTS know how to do this
            RATE[i][j] = NMR.rij2sigma( XYZptr->getn(i),
XYZptr->getn(j), XYZptr->getRij(i,j) );
        }
    }

    // return 0=no error
    return ( 0 );
}
```

7.3.7 Source code: structure.c and structure.h:

The following C++ source code files define the object "Structure" and allow for storage and retrieval of the Cartesian coordinates of a structure, calculation of distances between atoms, calculation of the center of mass, etc.

Strcuture.h C++ header file

```
// STRUCTURE.H
// class definitions for nmr relaxation calculations
// Jon Lapham <lapham@tecate.chem.yale.edu>

#ifndef STRUCTURE_H
#define STRUCTURE_H

class Structure {
public:
    // default constructor and destructor
    Structure( int );    // Input the number of atoms
    ~Structure();

    // read functions (read from STDIN)
    void readNxyz();     // does not include order parameters
    void readNxyzs();    // includes order parameters
    void readFull();
    void readPair();

    // set functions
    void setN( int );                // Number of atoms
    void setX( int, float );
    void setY( int, float );
    void setZ( int, float );
    void setn( int, int );           // Equivalent atoms
    void setS( int, float );
    void setRij( int, int, float );
    void setRijFix( int, int, float );
    void setBeta( int, int, float );

    // get functions
    int getN() const;                // Number of atoms
    float getX( int ) const;
    float getY( int ) const;
    float getZ( int ) const;
    int getn( int ) const;           // Equivalent atoms
    float getS( int ) const;
    float getRij( int, int ) const;
    float getRijFix( int, int ) const;
    float getBeta( int, int ) const;
    float getCoM_x() const;          // Center of mass, x
    float getCoM_y() const;          // Center of mass, y
    float getCoM_z() const;          // Center of mass, z

    // print functions (print to STDOUT)
    void printFull() const;
    void printPair() const;

    // write functions (write to a file)
    void fileFull( char [] ) const;
    void fileNxyzs( char [] ) const;

    // calculation functions
    void calcRij();    // Builds the rij matrix from current
x,y,z
    void calcCoM();    // Calculates current center of mass
    void calcBeta( float, float, float );  // Builds beta
matrix from supplied vector

private:
    float *x, *y, *z;               // Coordinates
    float **rij;                    // rij matrix
    float *s;                       // Order parameter matrix
    float **rij_fix;                // fixed rij matrix
    float **beta;                   // beta matrix
    int *n;
    int N;                           // Number of atoms
    float CoM_x, CoM_y, CoM_z;       // Centers of Mass
    int *res_num;                    // Residue number
    char **segid;                    // Segment ID (XPLOR)
    char **res_type;                 // Residue type
    char **atom_type;                // Atom type
};

#endif
```

Structure.c C++ source code (Structure object definition)

```
// STRUCTURE.C
// member function definitions for the NmrParams class
// Jon Lapham <lapham@tecate.chem.yale.edu>

#include <iostream.h>
#include <fstream.h>
#include <math.h>

#include "defs.h"
#include "structure.h"
#include "myalloc.h"

// Declare constants

/************************************************************
    Structure Class
************************************************************/
// Structure constructor initializes each data member to zero
Structure::Structure( int N_in )
{
    int i;
    N = N_in;
    CoM_x = 0;
    CoM_y = 0;
    CoM_z = 0;

    // Dynamically allocate memory for the arrays
    x = NEW1D_F( N );
    y = NEW1D_F( N );
    z = NEW1D_F( N );
    n = NEW1D_I( N );
    s = NEW1D_F( N );
    res_num = NEW1D_I( N );
    rij = NEW2D_F( N, N );
    rij_fix = NEW2D_F( N, N );
    beta = NEW2D_F( N, N );
    segid = NEW2D_C( N, 5 );
    res_type = NEW2D_C( N, 4 );
    atom_type = NEW2D_C( N, 5 );

    // Do I have to do this?
    for (i=0; i<N; i++) {
        x[i] = 0;
        y[i] = 0;
        z[i] = 0;
        res_num[i] = 0;
        n[i] = 1;
        s[i] = 1;  // default order parameter is 1 (rigid)
    }

}

// Structure destructor
Structure::~Structure()
{
    // Dynamically deallocate memory for the arrays
    DELETE1D_F( x );
    DELETE1D_F( y );
    DELETE1D_F( z );
    DELETE1D_I( n );
    DELETE1D_F( s );
    DELETE1D_I( res_num );
    DELETE2D_F( rij );
    DELETE2D_F( rij_fix );
    DELETE2D_F( beta );
    DELETE2D_C( segid );
    DELETE2D_C( res_type );
    DELETE2D_C( atom_type );

}

// Structure set functions
void Structure::setN( int N_in ) { N = N_in; }
void Structure::setX( int i, float x_in ) { x[i] = x_in; }
void Structure::setY( int i, float y_in ) { y[i] = y_in; }
void Structure::setZ( int i, float z_in ) { z[i] = z_in; }
void Structure::setn( int i, int n_in ) { n[i] = n_in; }
void Structure::setS( int i, float s_in ) { s[i] = s_in; }
void Structure::setRij( int i, int j, float rij_in ) {
rij[i][j] = rij_in; }
void Structure::setRijFix( int i, int j, float rij_in ) {
rij_fix[i][j] = rij_in; }
void Structure::setBeta( int i, int j, float beta_in ) {
beta[i][j] = beta_in; }

// Structure get functions
```

```
int Structure::getN() const { return N; }
float Structure::getX( int i ) const { return x[i]; }
float Structure::getY( int i ) const { return y[i]; }
float Structure::getZ( int i ) const { return z[i]; }
int Structure::getn( int i ) const { return n[i]; }
float Structure::getS( int i ) const { return s[i]; }
float Structure::getRij( int i, int j ) const { return
rij[i][j]; }
float Structure::getRijFix( int i, int j ) const { return
rij_fix[i][j]; }
float Structure::getBeta( int i, int j ) const { return
beta[i][j]; }
float Structure::getCoM_x() const { return CoM_x; }
float Structure::getCoM_y() const { return CoM_y; }
float Structure::getCoM_z() const { return CoM_z; }

// Structure read functions
// read in a NXYZ file from STDIN
void Structure::readNxyz( )
{
    int i, j;
    char firstline[80];

    // The first line is a header
    cin.getline(firstline, sizeof(firstline));
    for (i=0; i<N; i++) {
        cin >> n[i] >> x[i] >> y[i] >> z[i];
    }
}

// read in a NXYZ file from STDIN
void Structure::readNxyzs( )
{
    int i, j;
    char firstline[80];

    // The first line is a header
    cin.getline(firstline, sizeof(firstline));
    for (i=0; i<N; i++) {
        cin >> n[i] >> x[i] >> y[i] >> z[i] >> s[i];
    }
}

// Read in a Pair file from STDIN
void Structure::readPair( )
{
    int i, j;
    // char *firstline = NEW1D_C( 80 );
    char firstline[80];

    // The first line is a header
    cin.getline(firstline, sizeof(firstline));
    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            cin >> n[i] >> n[j] >> rij[i][j] >> beta[i][j];

            // We set aside the memory, may as well use it!
            rij[j][i] = rij[i][j];
            beta[j][i] = beta[i][j];
        }
    }

    // DELETE1D_C( firstline );
}

// Read in full structure description from STDIN
void Structure::readFull()
{
    int i, j;
    char firstline[80];
    // char *firstline = NEW1D_C( 80 );

    // The first line is a header
    cin.getline(firstline, sizeof(firstline));
    for (i=0; i<N; i++) {
        cin >> segid[i] >> res_num[i] >> res_type[i] >>
atom_type[i] >> n[i] >> x[i] >> y[i] >> z[i];
    }
    // DELETE1D_C( firstline );
}

// Structure write functions
// write out Full description of structure to a file
void Structure::fileFull( char FILE[] ) const
{
    int i, j;
    // The first line is a header
    ofstream coord_out;
    coord_out.open( FILE );
```

```
    coord_out << "Full structure file output" << endl;

    for (i=0; i<N; i++) {
        coord_out << segid[i] << " " << res_num[i] << " " <<
res_type[i];
        coord_out << " " << atom_type[i] << " " << n[i] << " "
<< x[i];
        coord_out << " " << y[i] << " " << z[i] << " " << s[i]
<< endl;;
    }

    coord_out.close;
}

// write out Full description of structure to a file
void Structure::fileNxyzs( char FILE[] ) const
{
    int i, j;
    // The first line is a header
    ofstream coord_out;
    coord_out.open( FILE );

    coord_out << "Full structure file output" << endl;

    for (i=0; i<N; i++) {
        coord_out << n[i] << " " << x[i] << " " << y[i];
        coord_out << " " << z[i] << " " << s[i] << endl;
    }

    coord_out.close;
}

// Structure calculation functions
// calculate the rij matrix
void Structure::calcRij()
{
    /*****
      Calculates a rij matrix from XYZ coordinates
    *****/

    int i, j;
    double Ax, Ay, Az;

    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            Ax = pow( ( x[i]-x[j] ), 2 );
            Ay = pow( ( y[i]-y[j] ), 2 );
            Az = pow( ( z[i]-z[j] ), 2 );
            rij[i][j] = rij[j][i] = sqrt( Ax + Ay + Az );
        }
    }
}

// calculate the beta matrix (angle WRT an external vector)
void Structure::calcBeta( float Bx, float By, float Bz )
{
    /*****
      Calculates a beta matrix from XYZ coordinates
    *****/

    int i, j;
    double angle, cos_angle, angle_rad;
    double Ax, Ay, Az, magA, magB;

    magB = sqrt( pow(Bx, 2) + pow(By, 2) + pow(Bz, 2) );

    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            Ax = pow( ( x[i]-x[j] ), 2 );
            Ay = pow( ( y[i]-y[j] ), 2 );
            Az = pow( ( z[i]-z[j] ), 2 );
            magA = sqrt( pow(Ax, 2) + pow(Ay, 2) + pow(Az, 2)
);

            if (magA*magB != 0) {
                cos_angle = (Ax*Bx + Ay*By + Az*Bz) /
(magA*magB);
                angle_rad = acos( cos_angle );
                angle = (180/PI) * angle_rad;
                // Check to see if we are over 90 degrees...
                if (angle > 90) { angle = 180 - angle; }
            }
            else {
                angle = 0;
            }
            beta[i][j] = angle;

        }
    }
}
```

```
// calculate the center of mass
void Structure::calcCoM()
{
    /*****
      Calculates center of mass from XYZ coordinates

      -need to make this rigorous, use masses...
    *****/

    int i;
    double x_sum=0;
    double y_sum=0;
    double z_sum=0;

    for (i=0; i<N; i++) {
        x_sum += x[i];
        y_sum += y[i];
        z_sum += z[i];
    }

    CoM_x = x_sum / N;
    CoM_y = y_sum / N;
    CoM_z = z_sum / N;
}

// print the Full coordinate file to STDOUT
void Structure::printFull() const
{
    int i, j;

    // The first line is a header
    for (i=0; i<N; i++) {
        cout << "Segid=" << segid[i] << " res_num=" <<
res_num[i] << " res_type=";
        cout << res_type[i] << " atom_type=" << atom_type[i] <<
" x=" << x[i];
        cout << " y=" << y[i] << " z=" << z[i] << endl;
    }
}

// print a "Pair" file to STDOUT
void Structure::printPair() const
{
    /****
       Prints to STDOUT a YARM "Pair" file
    ****/

    int i, j;

    // The first line is a header
    cout << "Yarm pair file\n";
    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            cout << n[i] << " ";
            cout << n[j] << " ";
            cout << rij[i][j] << " " << beta[i][j] << endl;
        }
    }
}
```

7.3.8 Source code: structure_refine.c

The following C++ source code is used in the calculations of model refinement (called by the Structure_Refine YARM subroutine).

strcuture_refine.c C++ source code

```
/*************************************************************
    structure_refine

    -Jon Lapham <lapham@tecate.chem.yale.edu>
    -Dec 16, 1997
*************************************************************/
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// YARM includes
#include "defs.h"
#include "myalloc.h"
#include "structure.h"
#include "lapack_functions.h"
#include "nmr_relax.h"

// Misc functions
void print_mat  ( int, double **, char * );
double calc_mat_rms  ( int, double **, double ** );
double calc_mat_q6  ( int, double **, double ** );
void print_mat_sum (int, double **, double ** );
void norm_mat (int, double **, double ** );

double mat_diff( Structure *, double ** );

int func_real( NmrParams, Structure *, double **, double **,
double **, double **, double ** );

double calc_vec_norm  ( int, double * );
double calc_3vec_norm  ( int, double *, double *, double * );

int calc_gradient ( Structure *, double, double **, double *,
double *, double * );
int calc_exprij  ( Structure *, double **, double **, double
**, double ** );
void conjugate_gradient ( Structure *, double *, double *,
double *, double *, double &, double & );
```

```
void move_atoms( Structure *, double, double *, double *,
double * );

main (int argc, char *argv[]) {

    /*********************************************************
**
      Declare variables
    *********************************************************
*/
    int i, j, K;
    int pass; int count;
    char *FILE = NEW1D_C( 30 );

    double max_rij=100;       // Maximum rij
    double lambda = 1;       // step size

    double rms, q6;          // Statistics

    double f_value;          // current function value
    double f_value_old = 0;
    double line_min;         // current line minimization value
    double line_min_old;     // last line minimization value
    double norm;             // gradient normalization factor

    // Conjugate gradient variables
    double gg, dgg, gam;

    /*********************************************************
**
      Read command line arguments ( including 'N', the # of
atoms )
    *********************************************************
*/
    // Number of atoms
    int N = atoi( argv[1] );      // Number of atoms
    Structure *XYZptr;       // Pointer to a structure object
    Structure XYZ( N );      // Structure OBJECT!!!!!!!!
    XYZptr = &XYZ;           // Point the pointer to this
structure

    XYZ.readNxyzs();
    strcpy( FILE, "refine.begin" );
    XYZ.fileNxyzs( FILE );
```

```
    NmrParams NMR;            // Create a NMR Parameters OBJECT

    // Read in command line arguments
    NMR.setTl   ( atof( argv[2] ) );    // Long axis
correlation time
    NMR.setTs   ( atof( argv[3] ) );    // Short axis
correlation time
    NMR.setSfreq( atof( argv[4] ) );    // Spectrometer
frequency
    NMR.setVol0 ( atof( argv[5] ) );    // tmix=0 volume
    NMR.setTmix ( atof( argv[6] ) );    // mixing time
    double Ax   = atof( argv[7] );      // x component of
principal axis
    double Ay   = atof( argv[8] );      // y component of
principal axis
    double Az   = atof( argv[9] );      // z component of
principal axis
    int num_pass= atoi( argv[10] );     // number of iterations

    /*********************************************************
**
      Declare other variables that depend on 'N'
    *********************************************************
*/
    double *gx = NEW1D_D( N );        // gx = X gradient
    double *gy = NEW1D_D( N );        // gy = Y gradient
    double *gz = NEW1D_D( N );        // gz = Z gradient
    double *hx = NEW1D_D( N );        // hx = X descent
    double *hy = NEW1D_D( N );        // hy = Y descent
    double *hz = NEW1D_D( N );        // hz = Z descent
    double *xix = NEW1D_D( N );       // next gradient
    double *xiy = NEW1D_D( N );       // next gradient
    double *xiz = NEW1D_D( N );       // next gradient

    double *R = NEW1D_D( N*N );        // R = residuals
    double **ExpVol = NEW2D_D( N, N );  // experimental NOE
volumes
    double **ExpRijFix = NEW2D_D( N, N );  // experimental
fixed rijs
    double **ExpRij = NEW2D_D( N, N );  // experimental rijs
    double **SimVol = NEW2D_D( N, N );  // simulated NOE
volumes
    double **SimRate = NEW2D_D( N, N ); // simulated rate
matrix
```

```
    /*********************************************************
**
       Read in the experimental volumes from STDIN
    *********************************************************
*/
    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            cin >> ExpVol[i][j] >> ExpRijFix[i][j];
            ExpVol[j][i] = ExpVol[i][j];
            ExpRijFix[j][i] = ExpRijFix[i][j];
        }
    }

    // XYZ.printFull();

    /*********************************************************
**
      BEGIN COORDINATE REFINEMENT PROGRAM
    *********************************************************
*/

    XYZ.calcBeta( Ax, Ay, Az );

    NMR.printNmrParams();

    // Calculate the experimental rijs
    if ( func_real( NMR, XYZptr, ExpVol, ExpRij, ExpRijFix,
SimRate, SimVol ) != 0 ) {
        cout << "ERROR in func_real function!\n";
        return(0);
    }

    // strcpy(FILE, "simvol.out");
    // print_mat( N, SimVol, FILE );

    // Calculate the root-square difference of the ExpVol and
SimVol matrices
    f_value = mat_diff( XYZptr, ExpRij );
    cout << " FUNCTION VALUE (ExpRij - SimRij) = " << f_value
<< endl;

    // Calculate the gradient
    if ( calc_gradient( XYZptr, max_rij, ExpRij, xix, xiy, xiz
) != 0) {
        cout << "error in calc_gradient function\n";
```

```
        return(0);
    }

    // initialize arrays
    for (i=0; i<N; i++) {
        gx[i] = -xix[i];
        gy[i] = -xiy[i];
        gz[i] = -xiz[i];

        xix[i]=hx[i]=gx[i];
        xiy[i]=hy[i]=gy[i];
        xiz[i]=hz[i]=gz[i];
    }

    /*********************************************************
**
      Iteratively determine the merged volume matrix
    *********************************************************
*/
    for ( pass=0; pass<num_pass; pass++ ) {
        // Header for the beginning of an iteration
        cout <<
"=============================================\n";
        cout << " Iterative pass number " << pass << " of " <<
num_pass << endl;


        // Positions of the first two atoms
        cout << " atom 0 x=" << XYZ.getX(0) << " y=" <<
XYZ.getY(0) << " z=" << XYZ.getZ(0) << endl;
        cout << " atom 1 x=" << XYZ.getX(1) << " y=" <<
XYZ.getY(1) << " z=" << XYZ.getZ(1) << endl;

        // Calculate the step size, lambda
        if ( f_value < f_value_old*0.9999999999 ) {
            cout << " lambda raised from " << lambda;
            lambda *= 1.2;
            cout << " to " << lambda << endl;
        } else if ( f_value_old != 0 ) {
            cout << " lambda lowered from " << lambda;
            lambda *= 0.5;
            cout << " to " << lambda << endl;
        } else {
            cout << " first time through, lambda remains " <<
lambda << endl;
```

```
        }

        // Check to see if things aren't moving anymore
        if ( lambda < 1e-4 ) {
            cout << " EARLY TERMINATION, lambda=" << lambda <<
" is less than 1e-4\n";
            strcpy( FILE, "refine.done" );
            XYZ.fileNxyzs( FILE );
            return (0);
        }

        // Maximum step size allowed is 5 angstroms
        if ( lambda > 5 ) lambda = 5;
        f_value_old = f_value;

        // We have to do this so we don't change the f_value
value..?
        line_min = f_value;
        line_min_old = 2*line_min;
        count = 0;

        // Move the atoms along the gradient until they settle
on a minimum
        // or make 5 steps at the current lambda2 value
        while( ( line_min < line_min_old ) && ( ++count < 6 ) )
{
            line_min_old = line_min;

            // move the atoms a lambda * (xix, xiy, xiz)
distance
            move_atoms( XYZptr, lambda, xix, xiy, xiz );

            // Calculate the root-square difference of the
ExpVol and SimVol matrices
            line_min = mat_diff( XYZptr, ExpRij );
            cout << " " << line_min;
        }
        // Back up one step if necessary
        if ( line_min > line_min_old ) {
            // move the atoms a lambda * (xix, xiy, xiz)
distance
            lambda *= -1.0;
            move_atoms( XYZptr, lambda, xix, xiy, xiz );
            lambda *= -1.0;
```

```
            // Calculate the root-square difference of the
ExpVol and SimVol matrices
            line_min = mat_diff( XYZptr, ExpRij );
            cout << " backup to " << line_min;
        }
        cout << endl;

        XYZ.calcBeta( Ax, Ay, Az );

        // Recalculate the experimental rijs using these new
atom positions
        if ( func_real( NMR, XYZptr, ExpVol, ExpRij, ExpRijFix,
SimRate, SimVol ) != 0 ) {
            cout << "ERROR in func_real function!\n";
            return(0);
        }

        // Recalculate the root-square difference of the ExpVol
and SimVol matrices
        f_value = mat_diff( XYZptr, ExpRij );
        cout << " FUNCTION VALUE (ExpRij - SimRij) = " <<
f_value << endl;

        // Recalculate the gradient at this new position
        if ( calc_gradient( XYZptr, max_rij, ExpRij, xix, xiy,
xiz ) != 0) {
            cout << "error in calc_gradient function\n";
            return(0);
        }

        /* BEGIN CONJUGATE GRADIENT CODE */
        dgg = gg = 0.0;

        for (i=0; i<N; i++) {
            gg += gx[i]*gx[i];
            gg += gy[i]*gy[i];
            gg += gz[i]*gz[i];

            // dgg += xix[i] * xix[i];   // This is Fletcher-
Reeves
            // dgg += xiy[i] * xiy[i];   // This is Fletcher-
Reeves
            // dgg += xiz[i] * xiz[i];   // This is Fletcher-
Reeves
```

```
            dgg += (xix[i]+gx[i]) * xix[i];   // This is Polak-
Ribiere
            dgg += (xiy[i]+gy[i]) * xiy[i];   // This is Polak-
Ribiere
            dgg += (xiz[i]+gz[i]) * xiz[i];   // This is Polak-
Ribiere
        }

        // Gamma definition. never let it get above 1!
        if ( gg == 0 ) { gam = 0; }
        else { gam=dgg/gg; }

        cout << " gamma=" << gam << endl;

        // Use this line to bypass conjugate gradient, called
"steepest descent"
        // gam=0;

        for (i=0; i<N; i++) {
            gx[i] = -xix[i];
            gy[i] = -xiy[i];
            gz[i] = -xiz[i];

            xix[i]=hx[i]=gx[i]+gam*hx[i];
            xiy[i]=hy[i]=gy[i]+gam*hy[i];
            xiz[i]=hz[i]=gz[i]+gam*hz[i];
        }

        /* END CONJUGATE GRADIENT CODE */

    }

    strcpy( FILE, "refine.done" );
    XYZ.fileNxyzs( FILE );

    // free up memory, do I have to do this?
    DELETE1D_D( gx );
    DELETE1D_D( gy );
    DELETE1D_D( gz );
    DELETE1D_D( hx );
    DELETE1D_D( hy );
    DELETE1D_D( hz );
    DELETE1D_D( xix );
    DELETE1D_D( xiy );
    DELETE1D_D( xiz );
```

```
    DELETE1D_D( R );
    DELETE2D_D( ExpVol );
    DELETE2D_D( ExpRij );
    DELETE2D_D( ExpRijFix );
    DELETE2D_D( SimVol );
    DELETE2D_D( SimRate );

    return(0);

}

void move_atoms( Structure *XYZptr, double size, double *gx,
double *gy, double *gz )
{
    int i;
    int N = XYZptr->getN();
    double oldx, oldy, oldz;
    double x, y, z;

    for (i=0; i<N; i++) {
        oldx = XYZptr->getX(i);
        oldy = XYZptr->getY(i);
        oldz = XYZptr->getZ(i);
        x = oldx + (gx[i] * size);
        y = oldy + (gy[i] * size);
        z = oldz + (gz[i] * size);
        XYZptr->setX(i, x);
        XYZptr->setY(i, y);
        XYZptr->setZ(i, z);
    }

    return(0);

}

int func_real( NmrParams NMR, Structure *XYZptr, double
**ExpVol, double **ExpRij, double **ExpRijFix, double
**SimRate, double **SimVol )
{
    int i,j;
    int N = XYZptr->getN();
    double q6, rms;
    double SimRij;

    // Simulate the NOE spectrum from the current coordinates
```

```
    cout << " Converting current xyz coordinates into rij
matrix" << endl;
    XYZptr->calcRij();

    cout << " Converting current rij matrix into Rate matrix"
<< endl;
    if ( rij2rate_aniso( NMR, XYZptr, SimRate ) != 0 )
return(1);

    cout << " Converting Rate matrix into Volume matrix" <<
endl;
    if ( rate2vol( N, NMR, SimRate, SimVol ) != 0 ) return(1);
    cout << " Finished building Volume matrix" << endl;

    // Calculate the "experimental rijs" by comparing the
    // experimental volumes to the simulated
    if ( calc_exprij( XYZptr, ExpVol, SimVol, ExpRijFix, ExpRij
) != 0 ) return(1);
    cout << " ExpVol[0][0]=" << ExpVol[0][0] << "
SimVol[0][0]=" << SimVol[0][0] << endl;
    cout << " SimRate[0][0]=" << SimRate[0][0] << endl;
    cout << " ExpRij[0][0]=" << ExpRij[0][0] << "
SimRij[0][0]=" << XYZptr->getRij(0, 0) << endl;
    cout << " ExpVol[0][1]=" << ExpVol[0][1] << "
SimVol[0][1]=" << SimVol[0][1] << endl;
    cout << " SimRate[0][1]=" << SimRate[0][1] << endl;
    cout << " ExpRij[0][1]=" << ExpRij[0][1] << "
SimRij[0][1]=" << XYZptr->getRij(0, 1) << endl;
    cout << " ExpVol[1][1]=" << ExpVol[1][1] << "
SimVol[1][1]=" << SimVol[1][1] << endl;
    cout << " SimRate[1][1]=" << SimRate[1][1] << endl;
    cout << " ExpRij[1][1]=" << ExpRij[1][1] << "
SimRij[1][1]=" << XYZptr->getRij(1, 1) << endl;

    rms = calc_mat_rms( N, SimVol, ExpVol );
    q6 = calc_mat_q6( N, SimVol, ExpVol );

    cout << " RMS=" << rms << " q^(1/6)=" << q6 << endl;

    return(0);
}

double mat_diff( Structure *XYZptr, double **ExpRij )
{
    int i,j;
```

```
    double f_value = 0;   // final function value
    double SimRij;
    int N = XYZptr->getN();
    float biggest = 0;   // largest difference
    double diff_sq = 0;  // difference squared

    XYZptr->calcRij();

    // subtract the square of each element in the two matrices
    for(i=0; i<N; i++) {
        for(j=i; j<N; j++) {
            if ( ExpRij[i][j] != 0) {
                SimRij = XYZptr->getRij( i, j );
                diff_sq += pow( (SimRij - ExpRij[i][j]), 2);
                f_value += diff_sq;
                // Find biggest
                if (biggest < diff_sq) {
                    biggest = diff_sq;
                }
            }
        }
    }
    f_value = sqrt (f_value);
    biggest = sqrt (biggest);

    cout << " Biggest rij difference was " << biggest << endl;

    return( f_value );

}


double calc_vec_norm( int N, double *Vec )
{
    /*****
       calculates the normalization factor to make a vector of
unit size
    *****/

    int i;
    double sum = 0;

    for( i=0; i<N; i++ ) {
        sum += pow( Vec[i], 2);
    }
```

```
    // return the norm factor
    return ( sqrt(sum) );
}

double calc_3vec_norm( int N, double *Vec1, double *Vec2,
double *Vec3 )
{
    /*****
       calculates the normalization factor to make a vector of
unit size
    *****/

    int i;
    double sum = 0;

    for( i=0; i<N; i++ ) {
        sum += pow( Vec1[i], 2) + pow( Vec2[i],2) + pow(
Vec3[i],2);
    }

    // return the norm factor
    return ( sqrt(sum) );
}

int calc_exprij( Structure *XYZptr, double **ExpVol, double
**SimVol, double **ExpRijFix, double **ExpRij )
{
    int i,j;
    double sim6, exp6;
    int N = XYZptr->getN();

    // Loop through the experimental and simulated volumes
    // if they both exist, determine an "experimental" rij by
    // comparing the 1/6th power of the volumes...
    for(i=0; i<N; i++) {
    for(j=i; j<N; j++) {
        /***
          The experimental rij matrix is ESTIMATED by taking
the ratio of the
          1/6 powers of the simulated and experimental volumes
and multiplying
          by the simulated rij
        ***/
        if ( i == j ) {
```

```
            ExpRij[i][j] = 0;
        }
        else if ( ExpRijFix[i][j] != 0 ) {
            // If this is a fixed distance, set it
            ExpRij[i][j] = ExpRij[j][i] = ExpRijFix[i][j];
        }
        else if ( ExpVol[i][j] > 0 ) {
            sim6 = pow( SimVol[i][j], 1.0/6.0 );
            exp6 = pow( ExpVol[i][j], 1.0/6.0 );
            ExpRij[i][j] = ExpRij[j][i] = ( sim6 / exp6 ) *
XYZptr->getRij(i, j);
        }
        else {
            // Use the SimRij for ExpRij if a ExpVol doesn't
exist!
            // ExpRij[i][j] = ExpRij[j][i] = XYZptr->getRij(i,
j);
            ExpRij[i][j] = ExpRij[j][i] = 0;
        }
    }
    }

    // 0=no error
    return( 0 );

}

int calc_gradient( Structure *XYZptr, double max_rij, double
**ExpRij, double *dx, double *dy, double *dz )
{
    /******
      calculate gradient

      g = nabla V(r)
      nabla = d/dx + d/dy + d/dz
      V(r) = delta_r = sRij - eRij
      dV/dx = xi-xj / sqrt( pow((xi-xj),2) + pow((yi-yj),2) +
pow((zi-zj),2) );
    ******/

    // Define variables
    int i,j;
    double dV_dx, dV_dy, dV_dz;   // Temp storage of ij
gradient
    double norm;
```

```
    double eRij, sRij;            // Temporary storage of exp
and sim Rijs
    double delta_r;               // delta_r = sRij-eRij
    int N = XYZptr->getN();       // Number of atoms

    // Loop through each i atom
    for(i=0; i<N; i++) {

        // Initialize the gradient to zero for each i atom
        dx[i] = 0; dy[i] = 0; dz[i] = 0;

        // Sum ith gradient WRT all j atoms
        for(j=0; j<N; j++) {

            // Don't use i=j atom pairs or eRij bigger than
max_rij
            eRij = ExpRij[i][j];
            if ( (i != j ) && (eRij < max_rij) ) {

            sRij = XYZptr->getRij(i, j);

            // This is important!  If we DON'T have
experimental data,
            // then there should not be any gradient!
            if (eRij != 0) delta_r = sRij - eRij;
            else delta_r = 0;

            // cout << " calc_gradient: delta_r=" << delta_r <<
endl;

            // The "potential energy" is delta_r, which goes to
zero when delta_r=0
            // we don't have to calculate this...
            // V = delta_r;
            dV_dx = ( XYZptr->getX(i) - XYZptr->getX(j) ) /
sRij;
            dV_dy = ( XYZptr->getY(i) - XYZptr->getY(j) ) /
sRij;
            dV_dz = ( XYZptr->getZ(i) - XYZptr->getZ(j) ) /
sRij;

            // Sum up the gradient on each atom i

            dx[i] += dV_dx * delta_r;
            dy[i] += dV_dy * delta_r;
```

```
            dz[i] += dV_dz * delta_r;
            }
        }
    }

    // Normalize the gradient vector
    norm = calc_3vec_norm( N, dx, dy, dz );
    // cout << " calc_gradient: norm_factor=" << norm << endl;

    // divide each gradient vector by the norm factor
    for (i=0; i<N; i++) {
        if (norm != 0) {
            dx[i] *= 1 / norm;
            dy[i] *= 1 / norm;
            dz[i] *= 1 / norm;
        }
    }

    // debugging printout
    cout << " N=" << N << endl;
    cout << " calc_gradient: atom 0 dx[0]=" << dx[0] << "
dy[0]=" << dy[0] << " dz[0]=" << dz[0] << endl;
    cout << " calc_gradient: atom 1 dx[1]=" << dx[1] << "
dy[1]=" << dy[1] << " dz[1]=" << dz[1] << endl;
    cout << " calc_gradient: atom 2 dx[2]=" << dx[2] << "
dy[3]=" << dy[2] << " dz[2]=" << dz[2] << endl;

    // 0 = no error
    return( 0 );
}


void print_mat( int N, double **MAT, char *FILE  )
{
    int i, j;
    double mat;   // Temporary storage variable for Rate**

    cout << "Writing MATRIX to file named " << FILE << endl;
    ofstream mat_out;
    mat_out.open(FILE);

    // Write out the matrix to file
    // Only send out the lower triangle of data
    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
```

```
            mat = MAT[i][j];

            mat_out << "i=" << i << " j=" << j << " " << mat <<
endl;
        }
    }
    mat_out.close();

}

double calc_mat_rms ( int N, double **MAT1, double **MAT2 )
{
    /****
      Calculate the rms between common elements between two
matrices
    ****/

    int i, j;
    double RMS = 0;
    double RMS_top = 0;
    double RMS_bottom = 0;
    double RMS_div = 0;
    double mat1, mat2;
    int count=0;

    // Do RMS statistics on the volume sets
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {

            // Use temp vars to make the equations more
readable
            mat1 = MAT1[i][j];
            mat2 = MAT2[i][j];

            if ( ( mat1 != 0.0 ) && ( mat2 != 0.0 ) ) {
                RMS_top    += pow( ( mat1 - mat2 ), 2 );
                RMS_bottom += pow( mat1, 2) + pow( mat2, 2 );
                ++count;
            }
        }
    }
    RMS_div = RMS_top / RMS_bottom;

    if ( RMS_div < 0 ) {
        RMS = 0;
```

```
    }
    else {
        RMS = sqrt( RMS_div );
    }

    // cout << "RMS = " << RMS << endl;
    // cout << " Used " << count << " elements in RMS
calculation! Top=" << RMS_top << " bottom=" << RMS_bottom <<
endl;
    return (RMS);
}

double calc_mat_q6 ( int N, double **MAT1, double **MAT2 )
{
    /****
      Calculate the Q6-factor between common elements between
two matrices
    ****/

    int i, j;
    double Q6 = 0;
    double Q6_top = 0;
    double Q6_bottom = 0;
    double mat1, mat2;

    // Do RMS statistics on the volume sets
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {

            // Use temp vars to make the equations more
readable
            mat1 = MAT1[i][j];
            mat2 = MAT2[i][j];

            if ( ( mat1 != 0.0 ) && ( mat2 != 0.0 ) ) {
                Q6_top    += fabs( pow( mat1, 1.0/6.0 ) - pow(
mat2, 1.0/6.0 ) );
                Q6_bottom += pow( mat1, 1.0/6.0 ) + pow( mat2,
1.0/6.0);
            }
        }
    }

    if ( Q6_bottom == 0 ) {
        Q6 = 0;
    } else {
        Q6 = Q6_top / (0.5 * Q6_bottom);
    }

    // cout << "Q6 = " << Q6 << endl;
    return (Q6);
}

void print_mat_sum ( int N, double **MAT1, double **MAT2 )
{
    /*****
      Print out the sum total of all shared elements between
two matrices
    *****/

    int i, j, k;
    double mat1, mat2;  // temporary storage or each Matrix
element
    double Sum_Mat1 = 0;
    double Sum_Mat2 = 0;

    for (i=0; i<N; i++) {
        for (k=0; k<i; k++) {
            mat1 = MAT1[k][i];
            mat2 = MAT2[k][i];

            // Sum up each non-zero shared element
            if ( (mat1 !=0) && (mat2 != 0) ) {
                Sum_Mat1 += mat1;
                Sum_Mat2 += mat2;
            }
        }
        for (j=i; j<N; j++) {
            mat1 = MAT1[i][j];
            mat2 = MAT2[i][j];

            // Sum up each non-zero shared element
            if ( (mat1 !=0) && (mat2 != 0) ) {
                Sum_Mat1 += mat1;
                Sum_Mat2 += mat2;
            }

        }
    }
```

```
    // Print a little report
    cout << " print_mat_sum: First matrix total = " << Sum_Mat1
<< endl;
    cout << " print_mat_sum: Second matrix total = " <<
Sum_Mat2 << endl;
}

void norm_mat ( int N, double **MAT1, double **MAT2 )
{
    /*****
      normalize two matrices
    *****/

    int i, j;
    double mat1, mat2;  // temporary storage or each Matrix
element
    double Sum_Mat1=0;
    double Sum_Mat2=0;

    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            mat1 = MAT1[i][j];
            mat2 = MAT2[i][j];

            // Sum up each non-zero shared element
            if ( (mat1 !=0) && (mat2 != 0) ) {
                Sum_Mat1 += mat1;
                Sum_Mat2 += mat2;
            }

        }
    }

    double norm_factor = Sum_Mat2/Sum_Mat1;

    for (i=0; i<N; i++) {
        for (j=i; j<N; j++) {
            MAT1[i][j] = MAT1[i][j] * norm_factor;
        }
    }

}
```

7.4 References

Bloembergen N, Purcell EM, Pound RV. 1948. Relaxation Effects in Nuclear Magnetic Resonance Absorption. *Physical Review 73*:679-712.

Boelens R, Koning TMG, Kaptein R. 1988. *J. Mol. Struct. 173*:299.

Borgias BA, James TL. 1990. MARDIGRAS-A procedure for matrix analysis of relaxation for discerning geometry of an agueous structure. *J Mag Res 87*:475-487.

Clore GM, Gronenborn AM. 1985. *J. Mag. Res. 61*:158.

Clore GM, Gronenborn AM. 1989. Determination of three-dimensional structures of proteins and nucleic acids in solution by nuclear magnetic resonance spectroscopy. *Crit. Rev. Biochem. Mol. Biol. 24*:479-564.

Lipari G, Szabo A. 1980. Model-free approach to the interpretation of nuclear magnetic resonance relaxation in macromolecules. *Journal of the American Chemical Society 104*:4546-4559.

Nerdal W, Hare DR, Reid BR. 1989. Solution structure of the *Eco*RI DNA sequence: refinement of NMR-derived distance geometry structures by NOESY spectrum back-calculations. *Biochemistry 28*:10008-10021.

Patel DJ, Shapiro L, Hare D. 1987. DNA and RNA: NMR studies of conformations and dynamics in solution. *Q. Rev. Biophys. 20*:35-112.

Post CW, Meadows RP, Gorenstein DG. 1990. *JACS 112*:6796.

Reid BR. 1987. Sequence-specific assignments and their use in NMR studies of DNA structure. *Q. Rev. Biophys. 20*:1-34.

Schmitz U, James TL. 1995. How to generate accurate solution structures of double-helical nucleic acid fragments using nuclear magnetic resonance and restrained dynamics. *Methods in Enzymology 261*:3-44.

Tropp J. 1980. Dipolar relaxation and nuclear Overhauser effects in non-rigid molecules: the effect of fluctuating internuclear distances. *J Chem Phys 72*:6035-6043.

Wagner G, Wüthrich K. 1982. Sequential resonance assignments in protein 1H nuclear magnetic resonance spectra. Basic pancreatic trypsin inhibitor. *J. Mol. Bio. 155*:347-366.