THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

STRUCTURAL DETERMINATION AND REFINEMENT OF THE GRAMICIDIN A TRANSMEMBRANE CHANNEL AS STUDIED BY SOLID STATE NUCLEAR MAGNETIC RESONANCE SPECTROSCOPY

By

RANDAL R. KETCHEM

A Dissertation submitted to the
Program in Molecular Biophysics
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 1995

The members of the Committee approve the dissertation of Randal R. Ketchem defended on November 8, 1995.

[illegible]
Timothy A. Cross
Professor Directing Dissertation

[illegible]
William T. Cooper, III
Outside Committee Member

[illegible]
Terry Gullion
Committee Member

[illegible]
Randolph L. Rill
Committee Member

DEDICATION

To my wife, Paula Ketchem, who supported me through the long years of this work, and to my parents, Fred and Shareen Ketchem, for their lifetime of love and encouragement.

ACKNOWLEDGMENTS

Many have helped me obtain my Ph.D., without whom this goal would have been impossible to reach. Although I could never acknowledge all of them here, I would like to take this opportunity to thank a few of the people that kept me going.

When applying to the Institute of Molecular Biophysics, I listed Tim Cross as one of the people I would like to meet with during my interviews. He discussed with me the various projects in his group and built my excitement about the program and his research. After being accepted, I twisted his arm sufficiently to be admitted into his research group. I made the right decision for a major professor. Through all of my fumbling and experimental disasters he never lost his patience, though I often did. He has guided my project by allowing me to follow my own scientific curiosity. He has been generous, providing support for me to attend national conferences and present our research. He has headed the research group in such a way to promote active support and collaboration. Through all of these things he has taught me not only a great deal of science, but also much about the art of science. I will always be thankful for this.

John Riehm at the University of West Florida gave me my first job in science. For this I am ever grateful. He not only taught me the technical aspects of protein synthesis and purification, he gave me my love for protein structure research. His initial guidance was instrumental in my following years of research, and will continue to affect me for years to come.

I would like to give special thanks to Linda Nicholson and Quike Teng, both of whom were graduate students in the group when I first joined. They both taught me a great deal about the research projects and got me started on my own. Linda spent many long evenings teaching me how to run the spectrometers. She also taught me many useful computational techniques. Quike started my project before I joined the group, so my thanks go to him for getting the ball rolling.

The computational refinement procedure was incorporated into CHARMM through a collaboration with Dr. Benoit Roux at the University of Montreal. The maturity of the refinement procedure could not have occurred without many helpful discussions with Benoit, for which I am ever grateful.

None of the experimental data used in my research would have been possible without the help of Dick Rosanske and Tom Gedris, staff in the Chemistry NMR facility. Tom taught me how to run the Bruker instruments my first semester and was always available for my questions. Dick was always ready to rebuild the radio frequency components that I would burn out or otherwise destroy. When nothing would work, when no signal could be found from a sample, Dick would come in and fix it all.

The gramicidin samples used in this research could not have been produced without others. Kwun-Chi Lee and Myriam Cotten synthesized isotopically labeled gramicidin for me to order. Weidong Hu and Fang Tian both lent their expertise in organic chemistry to prepare the isotopically labeled FMOC amino acids used in the syntheses. The staff of the BASS lab, Hank Henricks and Umesh Goli, were always ready to help with the peptide synthesis and helped me a great deal with peptide purification.

All of the members of Tim Cross's group have contributed in some way to this work. They have all provided helpful discussions and given me ideas to pursue. I wish to thank them all.

My parents always told me that I could be anything I wanted when I grew up. They did not leave it all up to me, however. They pushed me in the right direction and cultivated my abilities before I even knew I had them. They helped me with my homework, they met with my teachers, they were integral to my education. Without their help and guidance I would never have made it this far.

My brother, Fred Ketchem, has been an inspiration to me. He stopped in the middle of his Bachelor's degree, just as I later did. He joined the Navy and went back to school, eventually completing a Master's degree. I also went back to school, but without the military service. When I completed my Associate's degree, he talked me into going to the University of West Florida, for which I am ever grateful. Since he had attended UWF, he set up appointments for me and showed me the ropes, making my start there much easier.

Special thanks go to my wife, Paula. She has helped me in many ways through these years of work. She listens to my ideas, praises me in my triumphs, comforts me in my failures, and supports me always.

Most of all I would like to thank God. He has guided me well in this life and I trust that He will continue to do so in the future. His love and support have never failed me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The determination of protein structures is central to the field of structural biology. Although techniques exist and are well developed for the determination of proteins in solution and proteins that are readily crystallized, techniques for the determination of protein structures in the solid state are just now being developed. Described here is such a technique. The structure of the membrane spanning, monovalent cation channel gramicidin A is solved through the use of experimentally determined Solid State Nuclear Magnetic Resonance orientational constraints. The structure is solved for the peptide in its native conformation in a fully hydrated lipid bilayer by taking advantage of the wealth of experimental observations. The resulting structure is computationally refined by a method described here using a simulated annealing protocol which defines a penalty function based on the experimental observations and the CHARMM energy. This refinement strategy produces a structure that meets the experimental data and has a reasonable global energy.

CHAPTER 1
INTRODUCTION

1.1 Dissertation Overview

Molecular biophysics is the branch of science that utilizes the application of physics to explore biological processes and phenomena on the molecular level. Many of the biological processes of life are carried out by complex systems composed of biological macromolecules, such as proteins. Since these molecules exist in the atomic world, physics is well suited to study their structure. The determination and understanding of the structure of these proteins at an atomic level is central to the understanding of their function.

Although most proteins exist within an aqueous media, many find their roles within an anisotropic environment, such as a cell membrane. Developing methods for the determination of the structure of proteins in such an environment is necessary. Solid State Nuclear Magnetic Resonance (SSNMR) is a useful tool for studying such systems when coupled with appropriate observations. The results of these observations can be interpreted to provide conformational constraints on the structure of the observed system. These constraints may then be used to build a high resolution structural representation of the protein.

This dissertation will introduce and discuss a method for the experimental structural determination and computational refinement of the peptide gramicidin A (gA) in a fully hydrated lipid bilayer. Chapter 1 gives a brief introduction to proteins, gramicidin, phospholipid membranes, SSNMR, and computational methods. Chapter 2 contains a description of the materials and methods used in this study. Chapter 3 provides details of the method for the determination of the initial gA structure, focusing on the peptide backbone. Chapter 4 describes the computational refinement of the entire gA structure constrained by the SSNMR observables. Chapter 5 discusses the final structure. An appendix has been included that contains the SSNMR spectra obtained for many of the observables, the final atomic coordinates, an analysis of the final structure, and the source code used to obtain the complete structure.

1.2 Proteins

Proteins are extremely important components in biological systems and are present in all living systems. They provide structure inside and between cells. They transport, such as hemoglobin and membrane channels. They synthesize and metabolize as enzymes. Proteins are the most versatile of all biological macromolecules. DNA is said to be the blueprint of life, but it is actually a blueprint for proteins.

1.2.1 Protein Composition

Despite the variable roles of proteins, they are composed of very few common building blocks known as amino acids. Twenty different amino acids are used to form the wide array of proteins necessary for the survival of an organism. The difference between the amino acids lies in the R group which is attached to the alpha carbon, as shown in Figure 1.1. Different substitutions on the alpha carbon form different amino acids. The amino acids covalently link in specified sequences to form a polymer, as shown in

Figure 1.1. The possible combinations of these twenty amino acids is sufficient to form all of the proteins required by an organism.

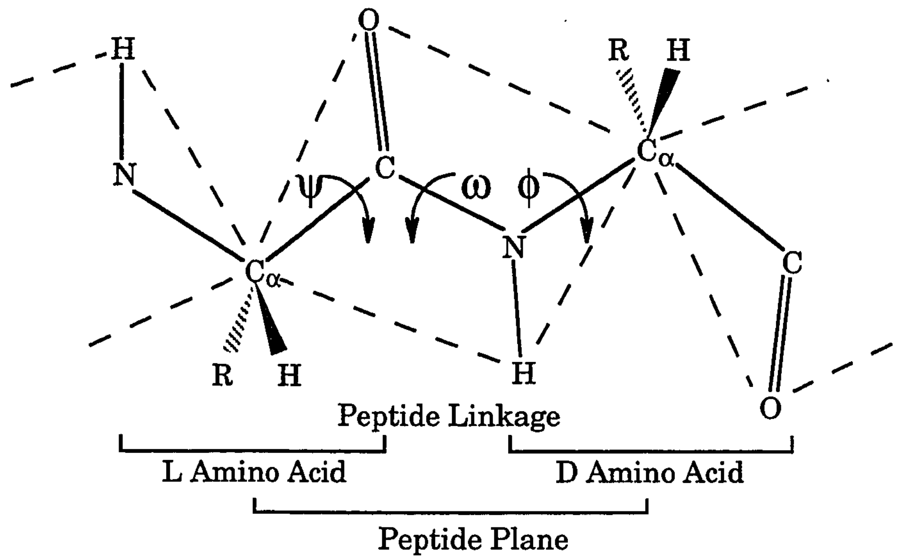


Figure 1.1: Dipeptide. Two amino acids link together to form a dipeptide. The amino acids link together by forming a covalent bond known as the peptide bond. The six atoms in a peptide linkage (C_α, C, O, N, H, C_α) define a plane if the ω torsion angle is either 0° or 180°. The R group defines the amino acid type.

All of the individual amino acids have a specific stereochemistry about the central alpha carbon, except glycine which has a hydrogen for an R group. The chirality of an amino acid is determined by comparing its rotation of polarized light with glyceraldehyde. D-glyceraldehyde (dextrorotatory) rotates polarized light to the right, so amino acids that behave in the same manner are D amino acids. Amino acids that rotate polarized light opposite to that of D-glyceraldehyde are termed L (levorotatory) amino acids. D and L amino acids are shown in Figure 1.1. All amino acids found in nature that are used by ribosomes for the synthesis of proteins are L amino acids.

1.2.2 Protein Structure

The sequence of amino acids is termed the protein primary structure. Proteins fold by changing the torsion angles about the backbone bonds. The peptide linkage, affected by the ω torsion angle, is considered planar due to its resonant bond structure, so changes in the ϕ and ψ torsion angles dominate the backbone structure of proteins. Local regions in the protein can fold into distinct secondary structures, such as alpha helices, beta strands and beta turns. These local secondary structures in turn fold together to form tertiary structures. Separate sequences folded into tertiary structures can bind together into a single functional unit leading to quaternary structure.

The final structure of a protein is dependent not only upon its primary structure, but also upon the environment in which it exists. The different amino acids have properties, such as hydrophobicity and charge that vary with environment. For example, a globular protein in an aqueous environment will have a shell of mainly hydrophilic amino acids surrounding a hydrophobic core. Proteins that span a lipid bilayer will have mainly hydrophobic amino acids within the lipid environment. When protein structure is discussed, it is important to consider both its sequence and environment.

1.3 Gramicidin

The membrane protein system used in this study is the peptide antibiotic gramicidin A (gA). gA is a pentadecapeptide that forms a monovalent cation channel in membranes. The gA monomer consists of fifteen amino acids, VGALAVVVWLWLWLW, with alternating L and D stereochemistry. The N-terminus is blocked by a formyl group and the C-terminus is blocked by ethanolamine (Sarges and Witkop, 1965). Gramicidin has been studied for more than fifty years as a simple membrane protein model. Despite the extensive research done on this channel, the complete structural description of the peptide in hydrated lipid bilayers has never been presented before and is the subject of this dissertation.

Gramicidin in nature exists as a mixture, denoted gramicidin D, composed of 80% gramicidin A, 5% gramicidin B and 15% gramicidin C (Weinstein *et al.*, 1980). Substitution of Trp_{11} in the sequence stated above with Phe yields gramicidin B, while substitution with Tyr yields gramicidin C. Synthesis is performed by multi-enzyme complexes that build the peptide from the N-terminal valine and ending with the addition of the C-terminal ethanolamine (Lipmann, 1980; Kurahashi, 1981). The naturally occurring L-residues are converted to D-conformers during the process (Akashi *et al.*, 1977; Akashi and Kurahashi, 1977).

Gramicidins are antibiotics produced by the bacterium *Bacillus brevis* during sporulation (Katz and Demain, 1977). The gramicidin molecules form monovalent cation channels across biological membranes causing lysis of many Gram positive bacteria. The disruption of the ion gradient in the targeted bacteria is lethal, thus providing *B. brevis* with needed resources for sporulation. Gramicidin has also been shown to induce sporulation in *B. brevis* (Mandl and Paulus, 1985).

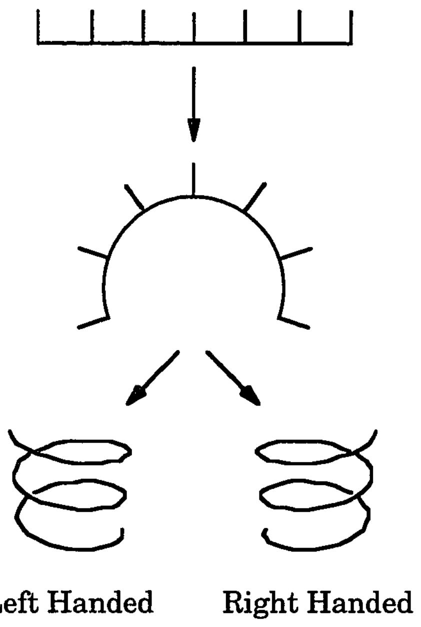


Figure 1.2: Beta Sheet. The primary structure of gA is such that the individual amino acids alternate in stereochemistry. The odd residues are (L) and the even residues are (D). When the structure is formed into a beta sheet, all of the amino acid sidechains lie on the same side of the sheet. This causes the β-sheet to bend into a β-helix. Depending on which end crosses the other, the helix can be either right handed or left handed. In the case of gA, the helix is right handed.

Due to the alternating L, D amino acids that make up the primary structure of gA, the structural motif formed by gA in lipid bilayers is that of a β-helix, with a backbone hydrogen bonding pattern similar to that in β-sheets (Urry, 1971). Since the amino acids alternate in stereochemistry, all of the sidechains are on one side of the peptide strand, forcing the backbone to curve, thus causing the formation of a helix, as shown in Figure 1.2. Since the backbone hydrogen bonding pattern is that of a β-sheet, the structure is termed a β-helix. The helical pitch has been measured by X-ray diffraction and found to be 4.7 ± 2 Å (Katsaras *et al.*,

1992) with 6.3 residues per turn. This gives the channel a length of 26 Å with a pore diameter of 4 Å.

As a helix, the hydrophobic sidechains point away from the helical axis, providing a hydrophobic environment on the surface of the helix. This allows the helix to exist in the hydrophobic region of phospholipid membranes, while at the same time providing a hydrophilic channel. The carbonyl oxygens, as well as the amide protons, are also typical of the β-sheet structural motif in that they alternately point parallel and anti-parallel to the helix axis. The presence of the carbonyl oxygens, along with the absence of the sidechains, in the channel pore produces a hydrophilic environment in the interior of the channel that allows for the passage of both water and monovalent cations.

The general structural motif as outlined above has been confirmed by SSNMR in phospholipids (Nicholson and Cross, 1989; Ketchem *et al.*, 1993; Mai *et al.*, 1993) and by solution NMR studies in sodium dodecyl phosphate (SDS) (Bystrov *et al.*, 1987; Lomize *et al.*, 1992). Spectroscopic data has been used to suggest that the helix is composed of roughly 6.3 residues per turn (Prosser *et al.*, 1991), and to determine that the helix is right handed in lipids (Nicholson and Cross, 1989). Further studies have shown that the monovalent cation channel is formed by an N-terminus to N-terminus head to head dimer of gA in hydrated lipid bilayers (Bamberg and Lauger, 1987).

gA in hydrated lipid bilayers does not lend itself well to conventional structural studies. The structural species formed in organic solvents have been solved by solution NMR (Bystrov *et al.*, 1987; Pascal and Cross, 1992; Pascal and Cross, 1993) and by X-ray crystallography (Wallace and Ravikumar, 1988; Langs *et al.*, 1991), but these structures are not consistent with channel function. The formation of co-crystals with lipids has proven difficult (Wallace and Janes, 1991). Solution NMR has been used with gA in SDS (Bystrov *et al.*, 1987; Lomize *et al.*, 1992), but the sidechain conformations differ from those determined by SSNMR (Hu *et al.*, 1993). The gA channel system is well suited for study by SSNMR since the channel can be easily incorporated into oriented lipid bilayers (Moll and Cross, 1990; Mai *et al.*, 1993) and, therefore, the channel can be oriented with respect to the external magnetic field allowing for structural characterization.

The structural study of gA in hydrated lipid bilayers will establish a method that can be applied to similar systems. The experimental techniques for obtaining structural constraints, the means by which these observations are converted to direct structural information and the computational structure refinement based on these observations are important for the study of membrane protein structures.

1.4 Phospholipid Membranes

Membranes are essential in the function of all cells. They are used to compartmentalize specific regions such as the nucleus, mitochondria, and the entire cell. Membranes also facilitate communication between the inside and the outside of these compartments, taking form in the passage of ions or conformational changes in the membrane. Also, proteins embedded in membranes are used for channels, communication and recognition. The major role of membrane lipids is to form lipid bilayers.

The peptide gA forms a monovalent cation channel in hydrated lipid bilayers. In preparing samples of gA in oriented lipid bilayers, the phospholipid dimyristoyl phosphatidylcholine (DMPC) is used in this study.

Phospholipids consist of a polar head group and long hydrocarbon tails, and are therefore amphipathic. As such, phospholipids readily form bilayers in aqueous media that have the polar head group of the phospholipid interacting with the polar solvent and the hydrophobic tails on the interior, as shown in Figure 1.3. The most significant interaction in an aqueous solution is the hydrophobic interaction. The nonpolar molecules cannot participate in the hydrogen bonding in the aqueous solution. The absence of hydrogen bonding between the nonpolar molecule, such as the lipid tails, and the water, rather than a favorable interaction between the nonpolar groups themselves, is a major factor in the stability of proteins and membranes.

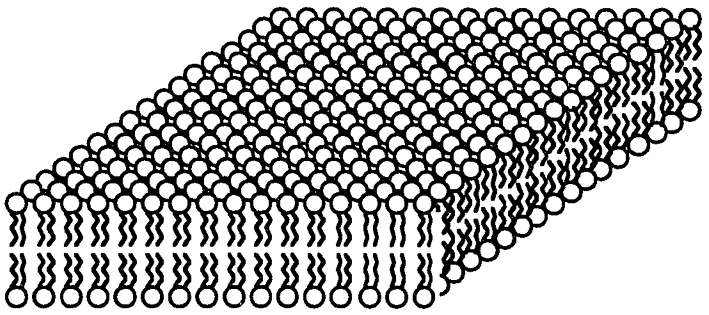


Figure 1.3: Lipid Bilayer. This schematic representation of a lipid bilayer shows the polar phospholipid head groups pointing outwards, thus interacting with the polar, aqueous media in which the bilayer exists. The apolar hydrocarbon tails group together forming a hydrophobic region.

The lipid bilayer forms a membrane around a biological cell which serves to protect it, and provides an effective barrier to most small molecule solutes. For a substance such as a monovalent cation to enter or leave the cell directly through the cell membrane, it would have to pass through the hydrophobic region of the membrane. The cation would have to break all of its bonds and shed its waters of hydration in order to do this, which would be energetically unfavorable. There is a constant flux of polar and ionic substances across the membrane, however. This transport is protein mediated by channels (or pores) and transporters. Channels are often pictured as tunnels across the membrane in which binding sites for the solutes being transported are accessible from either side of the membrane at the same time. Channel proteins do not require conformational changes to transport the solute, while transporters do. Channels do undergo conformational changes, though, as a regulation mechanism. In order to facilitate the passage of monovalent cations, to the demise of the host cell, gA forms a channel through the bilayer, thus destroying the cation gradient maintained by the membrane.

1.5 Solid State Nuclear Magnetic Resonance

The structure and refinement of gA in hydrated DMPC bilayers requires experimentally derived structural constraints. The method described here to obtain these constraints is SSNMR of oriented samples. SSNMR does not require that samples be crystallized, as in x-ray crystallography, nor does it require the system to undergo fast isotropic motions, as in solution NMR. SSNMR can be used to characterize interactions in an anisotropic environment or to determine structural information. Structural information can be obtained by either magic angle spinning to average the anisotropic interactions to obtain distance information, or by aligning the molecule of interest with the external magnetic field to obtain orientational information. The latter approach is used in this study.

A number of texts describing NMR in detail have been written (Abragam, 1961; Slichter, 1990; Sanders and Hunter, 1993). I will give a basic description of NMR here to facilitate an understanding of the experiments used in this study.

1.5.1 NMR

Nuclear spins have associated with them angular momentum described by the spin quantum number, I. The magnitude of the angular momentum, P, for a spin is

$$P = \hbar[I(I+1)], \qquad (1\text{-}1)$$

where $\hbar$ is Planck's constant divided by 2π. The angular momentum of a positively charged nucleus has a magnetic moment, $\vec{\mu}$, which interacts with an applied magnetic field. The magnetic moment and the angular momentum are related by

$$\vec{\mu} = \gamma\vec{P}, \qquad (1\text{-}2)$$

where γ is the gyromagnetic ratio of the observed nucleus.

The interaction of the nuclear magnetic moment with the external magnetic field, B_0, is termed the Zeeman interaction. This interaction removes the degeneracy of the different spin states so that

$$E_m = -\mu_Z \cdot B_0, \qquad (1\text{-}3)$$

where

$$\mu_Z = \gamma\hbar m \qquad (1\text{-}4)$$

is the projection of $\vec{\mu}$ on the external magnetic field.

For spin 1/2 nuclei (I = 1/2), such as the ^{15}N, ^{13}C and ^{1}H used in this study, two energy levels are generated in a magnetic field, as shown in Figure 1.4. The energy states are described by the nuclear spin quantum number, m, and are separated by an amount ΔE. The total number of energy states is 2I + 1 with values ranging from -I, -I + 1, ..., +I. The difference in the energy states is field dependent, described by

$$\Delta E = \hbar \gamma B_0. \qquad (1\text{-}5)$$

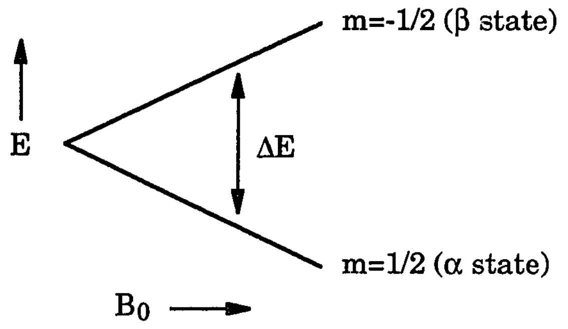


Figure 1.4: Energy Level Diagram for a Spin 1/2 Nucleus. As the magnetic field, B_0, increases, the separation between the energy states increases. The energy states are denoted with m.

For a single value of B_0, the frequency for the transition between the α and β energy states is given by

$$h\nu = |\hbar \gamma B_0 \Delta m| \qquad (1\text{-}6)$$

$$\nu_0 = |\gamma / 2\pi| B_0, \qquad (1\text{-}7)$$

where ν_0 is termed the Larmor, or observation, frequency. The gyromagnetic ratio is the proportionality constant that relates the Larmor frequency for a particular nucleus to the applied magnetic field. Different values of γ for different nuclei lead to different Larmor frequencies for the different nuclei on a particular spectrometer. The gyromagnetic ratio for ^1H is 26.7520×10^7 radians Tesla^{-1} s^{-1}. In a 4.7 Tesla magnetic field the

Larmor frequency is 200 MHz. An ^{15}N nucleus, with a gyromagnetic ratio of -2.712×10^7 radians Tesla^{-1} s^{-1}, has an observed frequency in the same field of 20.272 MHz (Fuller, 1976).

The two energy states for a particular nucleus do not have equal populations, and their ratio is given by the Boltzmann equation

$$N_\beta / N_\alpha = \exp(-\Delta E / kT), \tag{1-8}$$

where N_α is the population of the lower state, N_β is the population of the upper state, k is the Boltzmann constant and T is the absolute temperature. The population difference is directly related to the bulk magnetization and is dependent upon both the gyromagnetic ratio and the applied field. For protons at 200 MHz, the population difference is on the order of 1 in 10^5, which is very small. Despite this small population difference, NMR is able to measure the effects of the induced magnetization by resonance methods. The observable signal in NMR may last several seconds, allowing for various experiments to be performed leading to many different types of information.

The behavior of the bulk magnetization, M, can be described using vector diagrams. The bulk magnetization is the sum of the magnetization of the individual nuclear spins. M is initially aligned with B_0, as shown in Figure 1.5. The direction of B_0 is assigned to the Z axis of the laboratory Cartesian coordinate system. M will remain along B_0 unless it is perturbed. Once M is perturbed, there is a force on M by B_0. The magnetization behaves similarly to a gyroscope in a gravitational field and, therefore, the torque generated by B_0 on M causes M to precess about B_0 at a frequency

$$\nu_0 = \gamma B_0, \tag{1-9}$$

known as Larmor precession. M will eventually relax to B_0 after a time T_1, but during much of this time a vector component of M will precess in the XY plane. If a coil is wrapped around an axis perpendicular to the applied field, the precessing magnetization will induce a current in the coil which is read as the NMR signal.

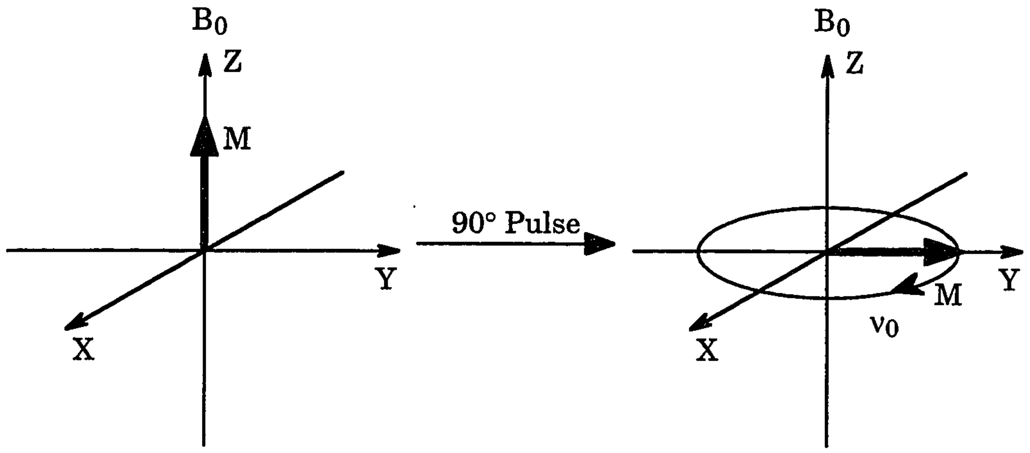


Figure 1.5: Bulk Magnetization. The bulk magnetization, M, is originally aligned along the applied magnetic field, B_0. After M is perturbed by a 90° pulse, the magnetization lies along Y and precesses about Z and B_0, at a frequency ν_0.

The precession of the magnetization is influenced by several interactions, such as chemical shift and dipole-dipole coupling. These interactions are observable by NMR experiments and provide useful information about the observed system. The observation and interpretation of these interactions are used in this study to provide structural constraints on gA leading to a structural solution for this peptide.

1.5.2 Chemical Shift

The precession frequency of the bulk magnetization of a single spin population is influenced by its electronic environment. Electrons circulate around the observed nucleus generating a local magnetic field in opposition to the external field. This has the effect of shielding the nucleus from B_0 and is described by an induced field

$$\vec{B}_{ind} = -\hat{\sigma}\vec{B}_0, \tag{1-10}$$

where $\hat{\sigma}$ is the second rank chemical shift tensor.

The chemical shift tensor is described by a 3×3 matrix and can be visualized as three orthogonal vectors. The magnitudes of the tensor components define an ellipse and the orientation of this ellipse with respect to the applied magnetic field defines a chemical shift value, as shown in Figure 1.6. If the coordinate system expressing the tensor is oriented along the semi-axis of the ellipsoid, the tensor is said to be in the Principal Axis System (PAS). The matrix describing the PAS is

$$\sigma_{PAS} = \begin{bmatrix} \sigma_{XX} & 0 & 0 \\ 0 & \sigma_{YY} & 0 \\ 0 & 0 & \sigma_{ZZ} \end{bmatrix}, \tag{1-11}$$

where σ_{XX}, σ_{YY}, and σ_{ZZ} are the principal elements of the tensor. The tensor elements are labeled as $|\sigma_{ZZ} - \sigma_{iso}| \geq |\sigma_{XX} - \sigma_{iso}| \geq |\sigma_{YY} - \sigma_{iso}|$, where $\sigma_{iso} = (\sigma_{XX} + \sigma_{YY} + \sigma_{ZZ})/3$.

If the population of nuclei giving rise to the observed signal is undergoing fast isotropic motion, much faster than the frequency range defined by the chemical shift anisotropy, then the chemical shift will be observed as the isotropic average of the chemical shift tensor and is termed the isotropic chemical shift. If the nuclei are in all possible orientations with respect to B_0 and are undergoing slower motions, then the NMR spectrum will exhibit a dispersion of peak positions due to the chemical shift anisotropy. The principal values of the chemical shift tensor can be measured in this way. If the nuclei giving rise to the signal have a unique orientation with respect to B_0, only a single chemical shift value will be observed.

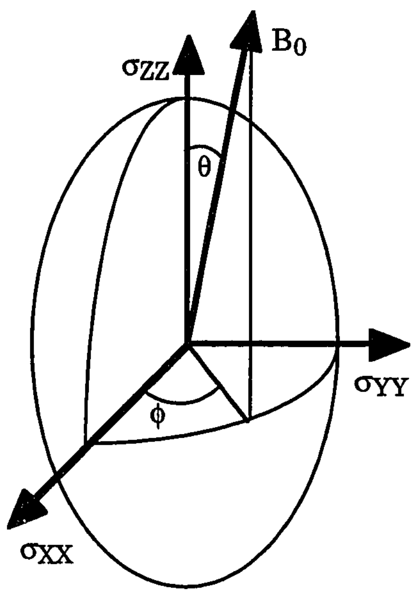


Figure 1.6: Chemical Shift Tensor. The chemical shift tensor is defined by the tensor elements σ_{XX}, σ_{YY} and σ_{ZZ} in the Principal Axis System (PAS) and is oriented with respect to the applied magnetic field, B_0, through the Euler angles θ and ϕ. The tensor elements define an ellipse and the orientation of this ellipse with respect to B_0 defines the chemical shift value.

The value of the chemical shift is dependent upon the orientation of the nuclei in the magnetic field and can range from the maximum to the minimum tensor element value. Since the magnetic field is aligned with the laboratory Z axis, the chemical shift value is described fully by the Z component of the tensor as

$$\sigma_{LAB} = \sigma_{XX}\cos^2\phi\sin^2\theta + \sigma_{YY}\sin^2\phi\sin^2\theta + \sigma_{ZZ}\cos^2\theta \tag{1-12}$$

where ϕ and θ are shown in Figure 1.6.

Since the chemical shift has an orientational dependence with the applied magnetic field, structural information can be gained from the observed chemical shift value of a sample oriented with respect to the magnetic field when coupled with knowledge of the chemical shift tensor. The orientation of the chemical shift tensor with respect to the molecular frame can be measured (Teng *et al.*, 1992). By then observing the oriented chemical shift value, the orientation of the molecular frame with respect to the magnetic field can be constrained, providing a powerful structural constraint (Teng *et al.*, 1991; Ketchem *et al.*, 1993).

1.5.3 Dipole-Dipole Coupling

Spin 1/2 nuclei generate a spatial dipolar field around themselves, and the field is modulated by surrounding spin 1/2 nuclei due to the interactions between the nuclei through space. The dipole-dipole interaction is a function of the two nuclear magnetic moments, the distance between the two nuclei and the orientation of the dipolar tensor with respect to the magnetic field, but is not a function of the strength of the external magnetic field.

The dipolar coupling tensor, $\hat{D}$, describes the dipole interaction and is an axially symmetric and traceless tensor (Mehring, 1983) in which the principal axis lies along the vector connecting the interacting nuclei. The dipolar interaction Hamiltonian for two nuclei I and S is

$$\hat{H}_D = \frac{\gamma_I\gamma_S\hbar^2}{r^3}\bar{I}\hat{D}\bar{S} \tag{1-13}$$

where $\bar{I}$ and $\bar{S}$ are the nuclear spin operators and r is the distance between the spins I and S. The dipolar interaction is weak compared to the Zeeman interaction, and therefore only the first order contribution needs to be considered. As a result, only the Z component of the dipolar coupling tensor is considered, which is

$$\hat{H}_D = \frac{\gamma_I \gamma_S \hbar^2}{r^3}\left(3\cos^2\theta - 1\right) I_Z S_Z \tag{1-14}$$

where θ is the angle between the unique axis of the dipolar interaction and the external magnetic field. For a spin 1/2 nucleus, the detected NMR signal from a dipole-dipole interaction is

$$\Delta\nu = \nu_{\parallel}\left(3\cos^2\theta - 1\right) \tag{1-15}$$

where

$$\nu_{\parallel} = \frac{\gamma_I \gamma_S \hbar}{r^3}, \tag{1-16}$$

$\Delta\nu$ is the observed dipolar splitting in hertz and $\nu_{\parallel}$ is the magnitude of the dipolar interaction.

If the length of the vector between spins I and S is known, then the observed dipolar splitting for an oriented sample serves to define the orientation of the internuclear vector and the external magnetic field. This provides another structural constraint.

1.5.4 Quadrupole Interaction

Nuclei with I > 1/2 have an ellipsoidal spatial charge distribution in the nucleus. The ellipsoid can be either prolate or oblate to guarantee a symmetric distribution about the nuclear spin axis. The distribution of charged particles generates an electric quadrupole moment defined by eQ, where e is the charge of a proton and Q, in length squared, is a function of the spatial distribution of the positive charges within the nucleus.

The electric quadrupole moment has no net interaction with a homogeneous electric field, but interacts with an inhomogeneous electric field. Such an interaction is termed the quadrupole interaction.

$$\hat{H}_{QI} = \bar{I}\hat{Q}\bar{I} \tag{1-17}$$

$$\hat{Q} = \frac{eQ}{2I(2I-1)\hbar}\hat{V} \tag{1-18}$$

where $\hat{H}_{QI}$ is the Hamiltonian for the quadrupole interaction of spin I, $\hat{Q}$ is the quadrupole interaction tensor, and $\hat{V}$ is the electric field gradient (EFG) tensor.

For a nucleus with I = 1, such as deuterium, there are two resonance frequencies corresponding to the three energy levels caused by the quadrupole interaction. The difference between these two frequencies, observed as quadrupole splittings, is expressed in the LAB frame as (Sillescu, 1982; Spiess, 1983; Spiess, 1985)

$$\Delta\nu = \frac{3}{4}\frac{e^2qQ}{\hbar}\left[\left(3\cos^2\theta - 1\right) - \eta\cos 2\phi\sin^2\theta\right], \tag{1-19}$$

where $V_{ZZ} = eq$ ($|V_{ZZ}| \geq |V_{XX}| \geq |V_{YY}|$) is the unique principal element of the EFG tensor,

$$\eta = \frac{V_{XX} - V_{YY}}{V_{ZZ}} \tag{1-20}$$

is the asymmetry factor, ϕ and θ are the Euler angles that describe the orientation of the EFG to the magnetic field, and $\frac{e^2qQ}{\hbar}$ is the quadrupole coupling constant (QCC).

Unlike the dipolar interaction, often the quadrupolar interaction is slightly asymmetric. Because the asymmetry is small (η = 0.03-0.05) in aliphatic hydrocarbon systems, it can be ignored and the unique tensor element assumed to lie on the internuclear vector. It is therefore reasonable to set η = 0, making equation (1-19)

$$\Delta\nu = \frac{3}{4}QCC\left(3\cos^2\theta - 1\right). \tag{1-21}$$

If QCC is known, the quadrupole splitting for a specific site serves to orient that C-^{2}H bond with respect to the magnetic field, providing another structural constraint.

1.5.5 Use of SSNMR Observed Interactions

In this study, isotopic labels are used in the synthesis of gA to allow for the observation of specific bond orientations with respect to the external magnetic field, such as ^{15}N-^{1}H and ^{15}N-^{13}C$_1$. This is accomplished through the use of SSNMR experiments to obtain chemical shifts, chemical shift tensor orientations and dipolar splittings. Since samples of gA in hydrated DMPC can be made such that the gA channels are aligned parallel to the magnetic field, the individual molecular bond orientations can be observed as a function of dipolar splitting and the chemical shift tensor orientations can be observed. The entire structure can be solved by piecing together these orientational constraints.

1.6 Computational Methods

1.6.1 Simulated Annealing

1.6.1.1 General description. An introduction to and application of simulated annealing is provided in Numerical Recipes in C (Press *et al.*, 1992). I will give a brief introduction here.

An analogy of simulated annealing to thermodynamics is the way in which a metal cools or anneals. At high temperatures, the molecules have high mobility. As the metal cools, this mobility is slowly lost and the molecules are often able to line themselves up to form a minimum energy conformation. If the metal is cooled quickly, or quenched, it does not reach the global minimum but ends up in an amorphous state. The process of slow cooling allows ample time for molecular redistribution.

The method of simulated annealing (Kirkpatrick *et al.*, 1983) is a technique suitable for large scale optimization problems in which a desired global minimum is hidden among many local minima. In direct minimization the system is brought from its initial configuration to a local minimum by a direct path. This technique often fails to reach the global conformational minimum. Nature handles minimization differently by allowing changes in configuration to occur both in uphill and downhill directions. This is shown by the Boltzmann Probability Distribution:

$$\mathrm{Prob}(E) \sim \exp(-E / kT), \qquad (1\text{-}22)$$

where E is the system energy, k is the Boltzmann constant, and T is the system temperature. This says that a system in thermal equilibrium at temperature T has its energy probabilistically distributed among all energy states E. At high T, there exists a high probability for the existence of a high energy state. At low T, the probability for a high energy state is decreased. Therefore, the system is allowed to move uphill, but uphill movement becomes less likely as T decreases.

When a system undergoes a change from E_1 to E_2, it does so with a probability

$$p = \exp[-(E_2 - E_1) / kT]. \qquad (1\text{-}23)$$

For $E_2 < E_1$ (downhill), $p \equiv 1$ (always allow)

For $E_2 > E_1$ (uphill), $p < 1$ (sometimes allow)

As T decreases, p decreases, so the probability of taking an uphill step is decreased. This general method, where a downhill step is always accepted and an uphill step is sometimes accepted based on T, is called the Metropolis algorithm (Metropolis *et al.*, 1953).

1.6.1.2 Application of the Metropolis algorithm. The application of simulated annealing to a system requires the following:

1. Description of possible system configurations.
2. Generation of random configuration changes.
3. Objective function for E (analog of energy). Minimization of this parameter is the goal of the procedure.
4. Control parameter T and annealing schedule (how to lower T).

This procedure is demonstrated by the classic Traveling Salesman problem, in which a salesman is required to visit N cities and return to his city of origin. Each city is to be visited once and the route is to be as short as possible. The number of possible solutions is (N-1)!. The computation time of this problem for an exact solution increases with N as exp(cN), where c is a constant. This becomes quickly prohibitive as N increases. The objective function E for this problem has many local minima. The annealing procedure locates a minimum that, even if not absolute, cannot be significantly improved upon, while limiting its calculations to scale as a small power of N.

The application requirements are as follows:

1. Description of possible system configurations.

   Number cities i = 1...N with coordinates (x_i, y_i).
2. Generation of random configuration changes.

   a) Reverse a section of path ($1\ \underline{2\ 3\ 4}\ 5 \rightarrow 1\ \underline{4\ 3\ 2}\ 5$).

   or

   b) Move a section of path to another location ($1\ \underline{2\ 3}\ 4\ 5 \rightarrow 1\ 4\ \underline{2\ 3}\ 5$).
3. Objective function for E (analog of energy). Minimization of this parameter is the goal of the procedure.

   $$E \equiv \text{Total Path Length} = \sum_{i=1}^{N+1}\left[(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2\right]^{\frac{1}{2}}$$

   $N + 1 \equiv 1$ (the path is cyclic)

4. Control parameter T and annealing schedule (how to lower T).

   In order to get an idea of what the starting T should be and how to reduce T, generate several random rearrangements of the order of the cities. From the random rearrangements, calculate an average ΔE for random rearrangements. Choose a starting $T >> \Delta E$. For this application, T is started at 0.5. Proceed downward ($T \rightarrow 0$) by 10% decreases in T. Hold each T for 100N rearrangements or 10N successful rearrangements, whichever is first. Stop rearrangements when no further successful rearrangements are found.

The following is a result found for 35 cities. The coordinates for the cities are assigned randomly in a square of area 1. A plot of the path is shown in Figure 1.7. The path length is significantly shortened in only 4 seconds of computer time. A complete search of all possible paths would require 2.95×10^{38} configurations. A comprehensive search of all configurations at a rate of 50000 per second (a reasonable number on current workstations), would require 2×10^{26} years, which is not feasible.

```
Order of cities before anneal:
    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15
   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30
   31   32   33   34   35

Initial Path Length:    20.68

Order of cities after anneal:
   17   21    5   27   24   23   14   22   31   16   15    2   11   25   35
   34    7   28    3   20    4   33    8    1   13   26    6   10   19   18
   12   32   29    9   30

Final Path Length:    4.55
```

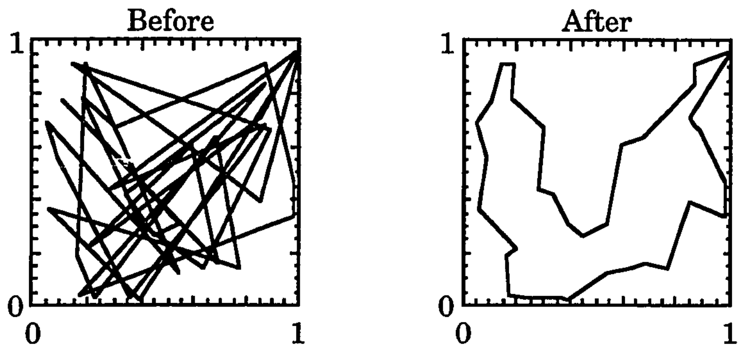


Figure 1.7: Traveling Salesman Path Before and After Simulated Annealing. The path before simulated annealing is obviously not optimized for the shortest path length. After simulated annealing, however, the path is significantly shortened.

The application of simulated annealing to gA is accomplished generally the same way and is described in detail in Chapter 4.

1.6.2 Protein Structure Determination and Refinement

1.6.2.1 Solution NMR. Techniques for the determination and refinement of protein structures in solution using NMR are well developed. Distance constraints based on nuclear Overhauser effect intensities and angle constraints based on coupling constants are used to generate an ensemble of initial structures and to refine the structures. Knowledge of the covalent geometry for the protein of study, such as the stereochemistry of its sidechains, and imposed energy functions describing its geometry are used in conjunction with the experimental constraints during the refinement procedure. Methods based on this strategy have received many recent reviews (Braun, 1987; Altman and Jardetzky, 1989; Clore and Gronenborn, 1989; Wüthrich, 1989; Gippert *et al.*, 1990; Brünger and Karplus, 1991; Clore and Gronenborn, 1991; Havel, 1991).

Protein structure determination is usually done in two steps. First, a set of initial structures is generated that approximately satisfy the covalent and experimental constraints. Second, an average of the structural ensemble is refined using the same constraints. The methods used for the steps are quite different. The initial structure is found by searching wide regions of conformational space and generating structural ensembles with random distributions within the imposed constraints. The final structure is generated by refining initial structures by improving the quality of the fit to the experimental and covalent constraints.

1.6.2.2 Solid State NMR. Protein structure determination in the solid state is similar to the method used in solution NMR, but the details are quite different. While solution NMR uses a large number of constraints to control the iterative process in determining an ensemble of initial structures, the SSNMR method described in Chapter 3 of this dissertation solves the initial structure analytically by using constraints that define the orientations of the individual peptide planes with respect to the external magnetic field. Ambiguities in the constraints lead to a set of initial structures. The initial structures are refined multiple times using the full set of SSNMR data available for the system and energy functions describing its geometry, generating a structural ensemble. An average structure is generated from this ensemble and is refined, producing a final structure.

The development of SSNMR as a structural tool is being pursued by various research groups. Internuclear distances as structural constraints (Smith, 1993) can be determined by either rotational resonance (Griffiths and Griffin, 1993) or by rotational echo double resonance (REDOR) (Gullion and Schaefer, 1989). These techniques provide quantitative distance measurements as opposed to NOE derived distance constraints in solution NMR which are qualitative. Orientational constraints are being used as structural constraints not only as discussed in this dissertation, but also as constraints on the allowed conformational space in polar angles relative to the axis of orientation (Opella *et al.*, 1987). This is achieved by determining the allowed regions ($\alpha\beta$ plots) for individual spin interactions at a structural unit, such as a peptide plane, and overlapping the regions to constrain the orientation of the structural unit. These techniques and their applications provide ample evidence that SSNMR is a valid technique for structure determination.

1.6.2.3 CHARMM. The program CHARMM (Chemistry at HARvard Macromolecular Mechanics) (Brooks *et al.*, 1983) is used in this study to supply the energy of the system, as described in Chapter 4. The energy is composed of several terms. Internal energy terms consist of bond lengths, bond angles, dihedral angles and improper torsions. A Urey-Bradley term is included to add a distance energy between atoms A and C in an A-B-C bonded atoms triplet. The improper torsion term is used to maintain chirality about a tetrahedral extended heavy atom and to maintain planarity about certain atoms, such as a carbonyl carbon. Non-bonded interactions consist of van der Waals (VDW) interactions and electrostatic potentials. Hydrogen bonding is described solely in terms of non-bonded interactions.

CHAPTER 2
MATERIALS AND METHODS

2.1 Materials

2.1.1 Dimyristoyl Phosphatidylcholine

The oriented samples used in this study were prepared using dimyristoyl phosphatidylcholine (DMPC) at 99+% purity purchased from Sigma Chemical Corp. and was used without further purification or modification. DMPC is used as the lipid in this study due to its acyl chain length which is well suited to form bilayers having the same thickness as the gramicidin channel. Samples produced using a mixture of gA and DMPC have been well characterized (Nicholson *et al.*, 1987; Killian *et al.*, 1988; Moll and Cross, 1990), and are therefore routinely made.

2.1.2 Gramicidin A

Isotopically labeled L and D amino acids were purchased from Cambridge Isotope Laboratories. FMOC-L and -D amino acids were synthesized in our lab for use in peptide synthesis using FMOC N-hydroxysuccinimide ester by Weidong Hu and Fang Tian. Isotopically labeled gA samples were synthesized by solid phase peptide synthesis using FMOC blocking groups on an Applied Biosystems 430A automated peptide synthesizer (Fields *et al.*, 1989) by Kwun-Chi Lee and Myriam Cotten. The C-terminus ethanolamine blocking group was added during the cleavage process to remove the synthesized gA from the resin. Single site isotopically labeled gA was consistently 98% pure after recrystallization as analyzed by reverse phase HPLC and no further purification was necessary. Double labeled ^{15}N-$^{13}C_1$ gA required purification and was done by reverse phase HPLC (22 × 250 mm column, C18, 10 micron pore, irregular silica) in a solution of 85% MeOH and 15% phosphate buffer (Fields *et al.*, 1989).

2.2 Oriented Sample Preparation

Oriented samples of gA in hydrated DMPC bilayers were prepared based on an earlier method used in this lab (Nicholson *et al.*, 1987). The method was modified slightly to accommodate larger sample sizes. Samples containing approximately 30 mg gA in hydrated DMPC bilayers were prepared by codissolving isotopically labeled gA with DMPC (1:8 molar ratio) in a 95% benzene / 5% ethanol solvent system (Mai *et al.*, 1993). This solution is spread on 25 thin glass slips measuring 5 mm × 20 mm and the slips are vacuum dried. They are then stacked into a square glass tube with one end sealed and hydrated to approximately 50% by weight. Because of the known gA solvent history dependence (Killian *et al.*, 1988; LoGrasso *et al.*, 1988) in which gA forms different conformations in solution based on the solvents used, the tube is fully sealed and incubated at 40 °C for a minimum of two weeks, resulting in a sample containing gA channels in fully hydrated lipid bilayers. The orientation of the channel axis of the gA helix is parallel to the normal of the glass surface (Fields *et al.*, 1988). The uniform orientation of the DMPC lipids in the oriented samples is confirmed by ^{31}P NMR (Moll and Cross, 1990). The orientational mosaic spread of the sample can be assessed from the resonance lineshapes and has been shown to be as little as 0.3° (Cross *et al.*, 1992). The resulting samples are highly stable, sometimes lasting many months, though repeated use of the sample often leads to degradation.

2.3 NMR Experiments

The experimentally observed orientational constraints used in this structure determination were acquired using two spectrometers, one built around a Chemagnetics data acquisition system and an Oxford Instruments 400/89 magnet, the other a heavily modified IBM/Bruker 200SY spectrometer equipped with a solids package. The ^{15}N-$^{13}C_1$ dipolar splittings were measured on the IBM/Bruker instrument due to the relatively larger size of the dipolar interaction with respect to chemical shift at lower field, and the ^{15}N-^{1}H dipolar splittings were measured on the Chemagnetics instrument due to the superior software for data acquisition.

The experiments were run at room temperature under a constant stream of room temperature air to keep the sample at constant temperature. The temperature of the coil region was measured during a typical experiment using a thermocouple and was found to be approximately 28 °C.

The ^{15}N 90° pulse length was determined using $(^{15}NH_4)SO_4$ by modifying the pulse length to find the point at which no signal was seen, indicating a 180° pulse. Half of this time was taken as the 90° pulse length. The power match for the cross polarization, which is explained in the next section, was set by changing the ^{1}H spin lock power to obtain the best signal and line shape from a dry powder sample containing ^{15}N-acetyl-glycine. The spectra were ^{15}N chemical shift referenced using a saturated solution of $^{15}NH_4NO_3$.

2.3.1 Cross Polarization

The sensitivity of the ^{15}N nucleus is very low due to its low abundance and low gyromagnetic ratio (-2.712×10^7 radians Tesla^{-1} s^{-1}) relative to ^{1}H (26.7520×10^7 radians Tesla^{-1} s^{-1}) (Fuller, 1976). Isotopically labeling the gA at a single site avoids the low ^{15}N abundance, but does not help avoid the low gyromagnetic ratio. Magnetization can be transferred, however, from a relatively high γ nucleus such as ^{1}H through a process known as polarization transfer or cross polarization (Hartmann and Hahn, 1962). The aim of cross polarization is to detect an NMR signal from a low sensitivity nucleus S surrounded by high abundance spins I with high sensitivity. This is accomplished when the Hartmann-Hahn match condition,

$$\gamma_I B_1^I = \gamma_S B_1^S, \tag{2-1}$$

is met. This says that the strength of the radio frequency generated magnetic fields at the Larmor frequencies for the I and S spins are adjusted so that the rate of precession for these rotating frames (in which the frame of reference rotates about the laboratory Z axis at a frequency equal to $2\pi\nu_1$ radians s^{-1}, where ν_1 is the precessional frequency in the B_1 field) are equal. At this point the spin state transition energies for the two spins are equal and the energy can be transferred from the abundant spin I to the low sensitivity spin S. Coupling takes place through a dipole-dipole interaction. The I spins are irradiated with a field B_1^I along X so that the I spin magnetization is placed along Y. The I spins are then spin locked along Y for the duration of the magnetization transfer, typically 1 ms. A contact pulse, also along Y, on the S spins will bring its magnetization along Y. If the match condition in equation (2-1) is met, the I magnetization will transfer to the S spins by resonance in the rotating frame. Acquisition of the NMR signal is done on the S spins while decoupling the I spins. Since the magnetization originates from the I spins, repetitive scans must be separated by a time dictated by the T_1 time of the I spins. The cross polarization pulse sequence is shown in Figure 2.1.

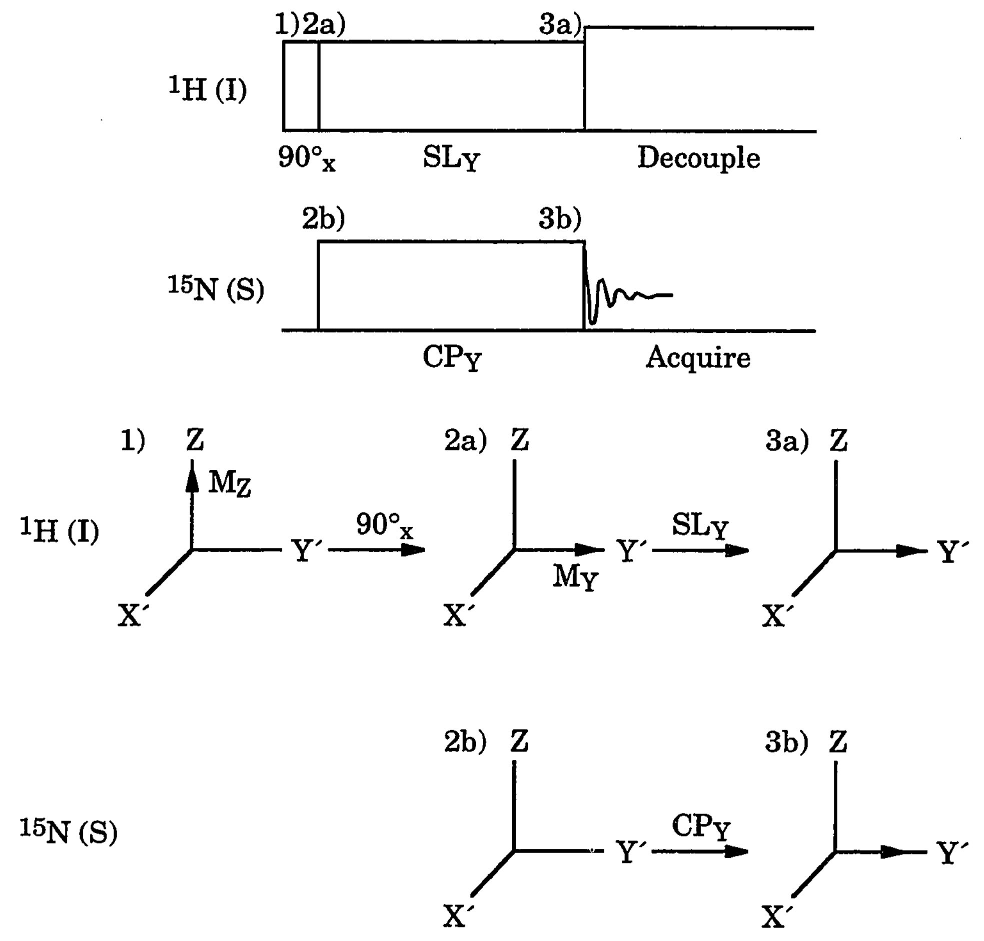


Figure 2.1: Cross Polarization. The magnetization from the relatively abundant I spins is transferred to the S spins. The magnetization of spin I is initially aligned with the magnetic field as shown in 1). The application of a 90° pulse along X places the magnetization of spin I along Y, shown in 2a). 2b) shows the bulk magnetization for spin S before cross polarization. Cross polarization transfers the magnetization of spin I to spin S as shown in 3a) and 3b).

2.3.2 Cross Polarization Echo

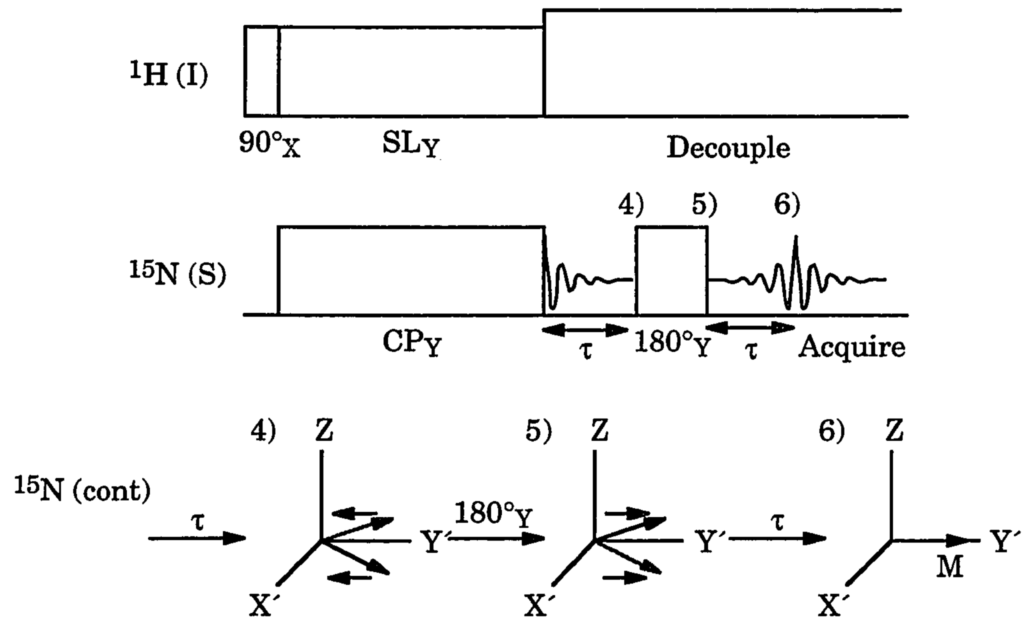


Figure 2.2: Cross Polarization Echo. The cross polarization echo pulse sequence is shown to refocus the magnetization along Y. The vector diagrams begins with the magnetization along Y as a continuation of Figure 2.1. Due to field inhomogeneities and various local environments, some spins precess faster than others, as shown in 4). A 180° pulse after a time τ along Y causes the magnetization to refocus, shown at 5). After an additional time τ the magnetization is refocused along Y, as in 6).

Under ideal conditions the CP experiment would be adequate to observe the free induction decay (FID) (the response of a nuclear spin system to a radio frequency pulse) of spin S. However, SSNMR uses high power which can lead to acoustic probe ringing, in which current is still in the coil after the termination of the pulse sequence. In order to alleviate this problem, a Hahn echo (Hahn, 1950) is used. The Hahn echo allows the spin magnetization in the rotating frame to dephase in the XY plane and then refocuses the spin magnetization for the observation of the echo after the acoustic ringing has stopped. The fact that the spins do not all precess at precisely the same frequency leads to a loss in phase coherence. If after a time τ a 180° pulse is applied, the spins are flipped and will refocus after another time τ, as shown in Figure 2.2. A typical value of τ is 60 μs. At the point of refocus, the echo can be acquired without the acoustic ringing.

This pulse sequence is used to observe the ^{15}N resonance when it is dipolar coupled with the $^{13}C_1$ nucleus. The dipolar interaction between the ^{15}N and $^{13}C_1$ leads to a dipolar splitting of the observed ^{15}N frequency which gives information about the N-C_1 bond orientation with respect to the magnetic field. Sample experimental spectra are shown in Appendix A.1.1.

2.3.3 Separated Local Field

To measure the ^{15}N-^{1}H dipolar splitting a method to reintroduce the dipolar coupling between the ^{15}N and its bonded ^{1}H while still being able to decouple for acquisition must be used. The separated local field (SLF) experiment (Waugh, 1976) is a two dimensional experiment that produces the ^{15}N chemical shift in the first dimension and the ^{15}N-^{1}H dipolar splitting in the second dimension. The SLF pulse sequence, shown in Figure 2.3, is a slight modification of the CPECHO pulse sequence shown in Figure 2.2.

The SLF pulse sequence uses an evolution time, t_1, to vary the time that the ^{15}N and ^{1}H dipolar coupling is present. At a small t_1, little dipolar coupling occurs. As a result, nearly the full intensity is seen in the chemical shift peak. As the evolution time is increased, the dipolar interaction time increases and the chemical shift signal intensity decreases. Over the course of the 16 1D experiments, the signal intensity oscillates, producing a FID in the second dimension. The ^{15}N-^{1}H dipolar splitting is obtained by Fourier transform of the second dimension. The width of the dipolar dimension is 50 kHz since the time increment is 20 μs (1/20 μs). Sample experimental spectra are shown in Appendix A.1.1.

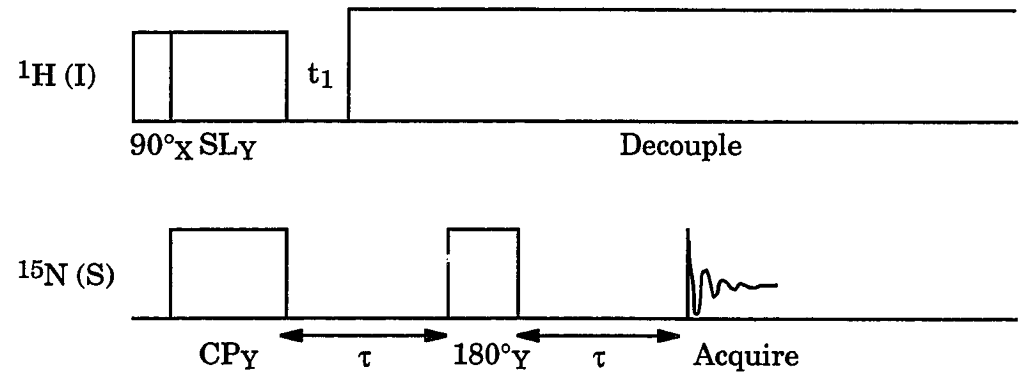


Figure 2.3: Separated Local Field. The SLF experiment is used to determine the ^{15}N-^{1}H dipolar splitting. The time t_1 is varied from 2 to 302 μs in steps of 20 μs for a total of 16 one dimensional experiments using a τ value of 360 μs. This leads to a dipolar dimensional width of 50 kHz. The dipolar interaction between the ^{15}N and ^{1}H nuclei is reintroduced as a function of the t_1 evolution period.

2.4 NMR Data Processing

The experimental NMR spectra were all processed on a Silicon Graphics Indigo II Extreme using the Felix NMR data processing software from Biosym Technologies. The CPECHO spectra were processed by first zero filling by a factor of four, applying an exponential multiplication with

50-100 Hz line broadening, Fourier transforming the echo, and applying phasing as required. The SLF experiments were processed in much the same way in the first dimension, and the dipolar dimension was processed by zero filling from 16 to 128 data points and Fourier transforming the FID.

2.5 Molecular Visualization and Analysis

Molecular visualization was required at many stages during this study and was carried out on a Silicon Graphics Indigo II Extreme using Insight II from Biosym Technologies. The diplane solutions from the SSNMR data, as will be discussed in Chapter 3, were assembled into the initial structure using Insight II. The sidechains were placed on the backbone and the final structure analyzed using this software. Procheck (Laskowski *et al.*, 1993) was used to analyze the properties of the final structure.

2.6 Computational Analysis

The individual (ϕ, ψ) torsion angles were calculated from the experimentally determined ^{15}N-$^{13}C_1$ and ^{15}N-^1H dipolar splittings using the program CNFCS (CoNFormation with reduction by Chemical Shift) developed in this lab. The diplane coordinates were created by using the diplane direction cosine solutions along with the (ϕ, ψ) solutions, both output from CNFCS, with the program COORDS (Calculate cOORDinateS), also developed in this lab. The global computational refinement was performed using the program TORC (TOtal Refinement of Constraints), developed in this lab. The source code for these programs is listed in Appendix A.4. All computer work was performed on both a Silicon Graphics IndigoII Extreme and a Silicon Graphics 4×R8000 Power Challenge.

CHAPTER 3
DETERMINATION OF THE INITIAL STRUCTURE

3.1 Introduction

The initial structure is determined not by model fitting the data, but by determining the structure entirely from the experimental data without using prior knowledge of the global structural characteristics. The determination of the backbone structure begins with the calculation of the individual backbone (ϕ, ψ) torsion angles from the experimentally determined ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings. Since the experimental data is measured as local orientational constraints with respect to an external reference, the magnetic field, the torsion angles can be determined locally and individually for each alpha carbon. Due to ambiguities in the observed dipolar splittings, multiple solutions are determined for each (ϕ, ψ) pair. Using the ^{15}N chemical shifts and C_α-^{2}H quadrupolar splittings as structural filters reduces the number of possible solutions. Four possible (ϕ, ψ) solutions remain for each alpha carbon, leading to 2^{n+1} initial backbone structures, where n is the number of alpha carbons. These solutions all have the same structural motif and differ subtly only in the peptide plane normal orientations with respect to the channel axis.

This chapter will explain the method for calculating the individual (ϕ, ψ) solutions, the reduction of these solutions using additional experimental constraints, the pairing of subsequent peptide planes to build the possible initial structures, and the reduction of the 2^{n+1} possible initial structures to four characteristic structures. The details of these initial structures will be discussed and the need for a computational refinement method introduced.

3.2 Calculation of the Individual Torsion Angles

The method for determining the backbone torsion angle pairs about a single alpha carbon has been previously described (Brenneman and Cross, 1990; Teng *et al.*, 1991). Recently this method has been modified and is implemented in the program CNFCS.

The gA channel is oriented in the lipid bilayer such that the helix axis is parallel to the normal of the lipid surface. The helix axis can then be aligned parallel to the external magnetic field of the spectrometer by positioning the sample in the magnetic field so that the glass plates are perpendicular to the magnetic field vector. Aligning the helix axis with respect to the magnetic field allows for the determination of individual spin interaction tensor orientations with respect to the magnetic field. Once the orientations of the tensors are known with respect to the molecular frame, the orientation of the molecular frame with respect to the magnetic field can be determined (Mai *et al.*, 1993). The relative orientations of the individual molecular frames can then be used to calculate the initial backbone torsion angles about individual alpha carbons (Teng *et al.*, 1991).

3.2.1 Determination of Possible Peptide Plane Orientations

As a first step in the determination of the (ϕ, ψ) solutions, the peptide plane linkage (ω) is assumed to be 180°. This causes the six backbone atoms on either side of the peptide bond to lie in a plane. Both the carbonyl oxygen and the amide proton are considered backbone atoms since the orientation of these atoms with respect to the main chain atoms cannot change without a modification in the peptide geometry. By the same reasoning, the H_α and C_β atoms are also considered backbone atoms and, therefore, data from these sites can be used as additional backbone structural constraints. Each peptide plane can be defined by two intersecting vectors, the N-H and N-C_1 bonds, as shown in Figure 3.1.

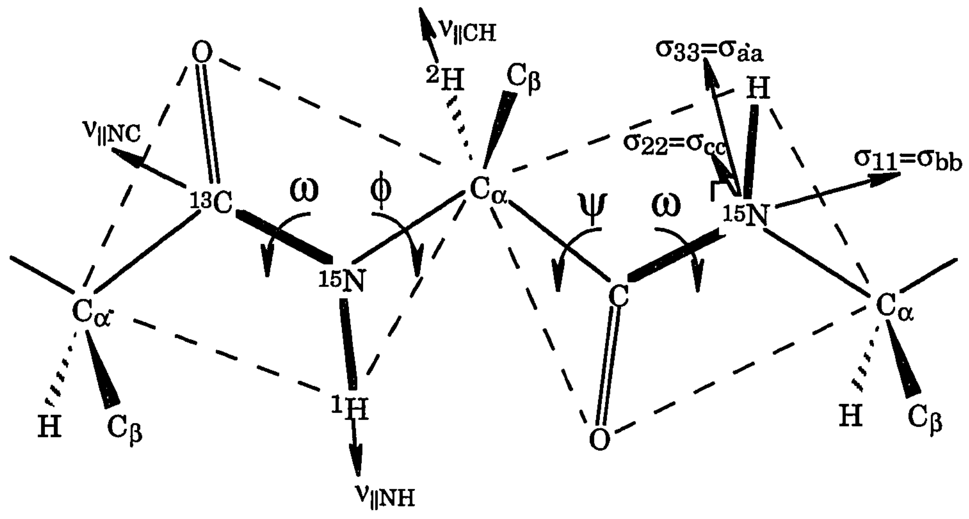


Figure 3.1: Peptide Plane Definition. The peptide plane is defined by the N-C_1 and N-H bonds. Since these two vectors intersect at the nitrogen, they are sufficient to define the orientation of the plane. The dipolar interactions that define the N-C_1 and N-H bond orientations are shown, as is the ^{15}N chemical shift tensor and the C-^{2}H quadrupolar interaction. Subsequent peptide planes are linked by taking advantage of the tetrahedral geometry about the shared alpha carbon.

3.2.1.1 Experimental data used. The determination of the bond orientations is accomplished by the incorporation of isotopic labels into gA;

^{15}N-^{1}H for the determination of the N-H bond orientation and ^{15}N-$^{13}C_1$ for the determination of the N-C_1 bond orientation, as explained in Chapter 2.3. Experimental data for multiple gA sites is shown in Appendix A.1.1. For those ^{15}N-$^{13}C_1$ spectra that contain a single peak the dipolar splitting is obtained by observing the shift of the single peak from the known chemical shift of the ^{15}N only labeled sample. The unique axis of the dipolar interaction is the internuclear vector, so an observed dipolar splitting serves to orient this vector or bond with respect to the external magnetic field.

3.2.1.2 Bond orientations and experimental error. The ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings and the ^{15}N chemical shift is theoretically redundant, as shown by the following system of equations (Teng *et al.*, 1991):

$$\sigma_{cs} = \sigma_{11}\cos^2\theta_{11} + \sigma_{22}\cos^2\theta_{22} + \sigma_{33}\cos^2\theta_{33} \qquad (3\text{-}1)$$

$$\cos\theta_{33} = (\sin\beta_D / \sin\theta_{HNC})(\cos\theta_{NH} - \cos\theta_{NC}\cos\theta_{HNC}) + \cos\theta_{NC}\cos\beta_D \qquad (3\text{-}2)$$

$$\cos\theta_{11} = (\cos\beta_D / \sin\theta_{HNC})(\cos\theta_{NH} - \cos\theta_{NC}\cos\theta_{HNC}) - \cos\theta_{NC}\sin\beta_D \qquad (3\text{-}3)$$

$$\cos\theta_{22} = (1 - \cos^2\theta_{11} - \cos^2\theta_{33}) \qquad (3\text{-}4)$$

where σ_{cs} is the observed chemical shift, σ_{XX} is the magnitude of the XX tensor element, θ_{XX} is the orientation of the XX tensor element with respect to the magnetic field, β_D is the orientation of the tensor with respect to the N-C_1 axis of the molecular frame and θ_{HNC} is the H-N-C_1 bond angle.

Table 3.1: Calculated versus Experimental ^{15}N-$^{13}C_1$ Dipolar Splittings. Differences in the actual peptide geometry are shown when the ^{15}N-$^{13}C_1$ dipolar splittings are calculated using the ^{15}N chemical shifts and the ^{15}N-^{1}H dipolar splittings.
†Relaxation effects have compromised our ability to obtain ^{15}N-$^{13}C_1$ dipolar splitting data for this site. The value shown is determined from the existing data and is used only in the determination of the initial structure.

Residue	Experimental (kHz)	Orientation (degrees)	Calculated (kHz)	Orientation (degrees)
Val_1	0.463	48, 63	0.988	40, 74
Gly_2	0.910	41, 72	0.804	42, 70
Ala_3	0.670	44, 67	0.815	42, 70
Leu_4	0.820	42, 70	0.724	44, 68
Ala_5	0.572	46, 65	0.913	41, 72
Val_6	0.626	45, 66	0.369	49, 61
Val_7	0.519	47, 64	0.819	42, 70
Val_8	0.702	44, 67	0.413	48, 62
Trp_9	0.487	47, 63	0.957	40, 73
Leu_{10}	0.700†	44, 67	0.018	54, 55
Trp_{11}	0.365	49, 61	0.114	53, 57
Leu_{12}	0.779	43, 69	0.760	43, 69
Trp_{13}	0.454	48, 62	0.568	46, 65
Leu_{14}	0.657	45, 66	0.475	47, 63
Trp_{15}	0.507	47, 63	0.560	46, 64

Equation (3-1) says that the chemical shift is a function of the magnitudes of the chemical shift tensor elements and the orientation of the chemical shift tensor with respect to the magnetic field. Equations (3-2), (3-3) and (3-4) solve for the orientations of the tensor elements using the N-H and N-C_1 bond orientations. If the observed chemical shift is known and the chemical shift tensor characterized, then with knowledge of either the ^{15}N-^{1}H or ^{15}N-$^{13}C_1$ dipolar splitting, the unknown dipolar splitting can be calculated, as shown in Table 3.1. However, the error associated with the observables and the imperfectly characterized dynamics results in typical errors of a few degrees and as much as 12°. As a result, the observation of ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings as well as the ^{15}N chemical shift has been useful for the structural characterization.

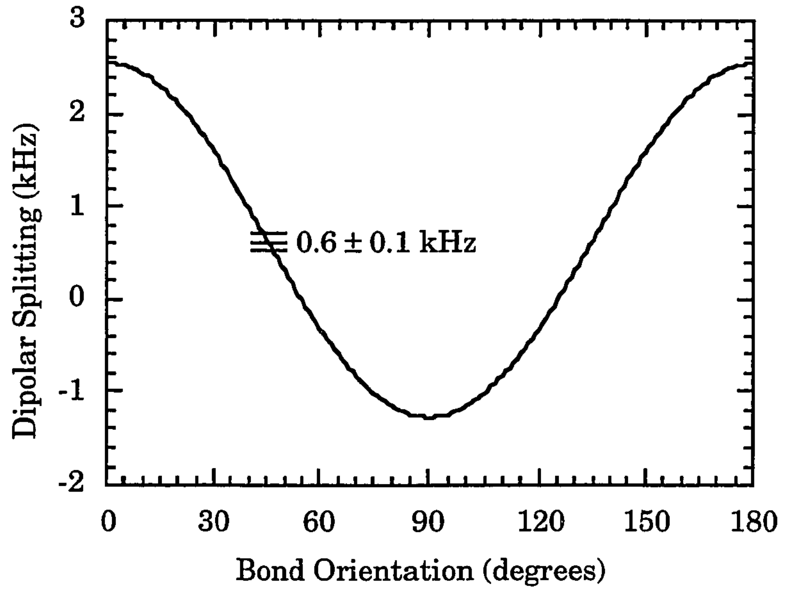


Figure 3.2: ^{15}N-$^{13}C_1$ Dipolar Splitting as a Function of N-C_1 Bond Orientation. The range of experimental error associated with a single observable may be large in the dipolar splitting dimension, but results in only a small range of possible bond orientations.

Table 3.1 illustrates the high resolution of these constraints. Since the experimentally determined bond orientations with respect to the magnetic field are all within 41°–49° or 61°–72°, which are both in regions of steep slope in a plot of dipolar splitting versus bond orientation (Figure 3.2). The range of experimental error about an observed splitting defines a very small range of possible bond orientations. For example, an ^{15}N-^{13}C$_1$ dipolar splitting of 0.600 typically has an error of ±0.1 kHz, which leads to a possible bond orientation range of only 47°–44°.

3.2.1.3 Bond orientation ambiguities. The nature of the dipolar interaction is such that the determined individual bond orientations are not unique. This orientational ambiguity arises from $\Delta\nu_{obs} = \nu_{||}\,(3\cos^2\theta - 1)$ in that the sign of $\cos\theta$ is undetermined. Furthermore, the sign of $\Delta\nu_{obs}$ is unknown unless the magnitude of $\Delta\nu_{obs}$ is greater than $\nu_{||}$. In the case of gA, the N-C$_1$ $\Delta\nu_{obs}$ is always less than $\nu_{||}$, while the N-H $\Delta\nu_{obs}$ is always greater than $\nu_{||}$. As a result, there are four possible N-C$_1$ orientations and only two possible N-H orientations. This leads to eight possible plane orientations based on the dipolar splitting data alone.

If the ^{15}N chemical shift tensor is oriented with respect to the molecular frame, the ^{15}N chemical shift can be used as an additional orientational constraint for the individual peptide plane orientations. Not only does the ^{15}N chemical shift tensor provide an orientational constraint, the calculation of the observed chemical shift from the plane orientation introduces information about the covalent geometry of the peptide plane through the HNC$_1$ bond angle (equation (3-2)). N-H and N-C orientation solutions must therefore meet this constraint. It can be shown (Brenneman *et al.*, unpublished results) that the introduction of the ^{15}N chemical shift constraint reduces the number of possible peptide plane orientations by a factor of four, leaving two possible orientations for each plane.

3.2.2 Linking the Peptide Planes

Once the independent orientations of two adjacent peptide planes are determined with respect to the magnetic field, the relative orientations of the two peptide planes can be defined in terms of the ϕ and ψ torsion angles about the linking alpha carbon (Figure 3.1). Two orientations are possible for each peptide plane, which leads to four possible diplane solutions consistent with the tetrahedral geometry about the linking alpha carbon.

Of the four diplane solutions, two are structurally identical to the other two, differing only in the orientation with respect to the XY plane. In gramicidin, the channel exists as a head to head dimer. The upper and lower monomers are identical with respect to their orientation to the magnetic field, but point in different directions. A diplane in the upper monomer is identical to the same diplane in the lower monomer after a 180° rotation about a vector in the XY plane. As a result, only two diplane solutions need to be considered.

With the introduction of the above constraints on the possible peptide plane orientations and the diplane combinations, a set of possible (ϕ, ψ) torsion angles is generated for the diplane (Teng *et al.*, 1991). Each diplane combination is associated with eight possible (ϕ, ψ) torsion angles (Brenneman *et al.*, unpublished results), resulting from sign ambiguities in the spherical angles used to calculate the torsion angles from the diplane direction cosines. In the case of gA, two possible diplane combinations exist for each alpha carbon, resulting in sixteen possible (ϕ, ψ) torsion angle pairs, as shown in Table 3.2.

The individual (ϕ, ψ) solutions are named by first stating the direction cosine number and then the (ϕ, ψ) number. The letter g is added at the beginning of each name to designate gramicidin. For example, the (91°, 73°) (ϕ, ψ) solution shown in Table 3.2 would be named g11 and the (-121°, 121°) solution would be named g28.

Table 3.2: Possible Ala$_3$ Diplane Torsion Angle Pairs. The two possible diplane direction cosine solutions lead to sixteen possible (ϕ, ψ) torsion angle pairs. The number of diplane direction cosine solutions, and hence the number of possible (ϕ, ψ) torsion angle pairs, is the same for each alpha carbon along the gA backbone.

Direction Cosines

	Plane 1		Plane 2	
	^{15}N-^{1}H	^{15}N-^{13}C	^{15}N-^{1}H	^{15}N-^{13}C
g1	0.9871	-0.3970	0.9160	-0.7405
g2	0.9871	-0.3970	-0.9160	0.7405

Combination 1 Torsion Angle Solutions

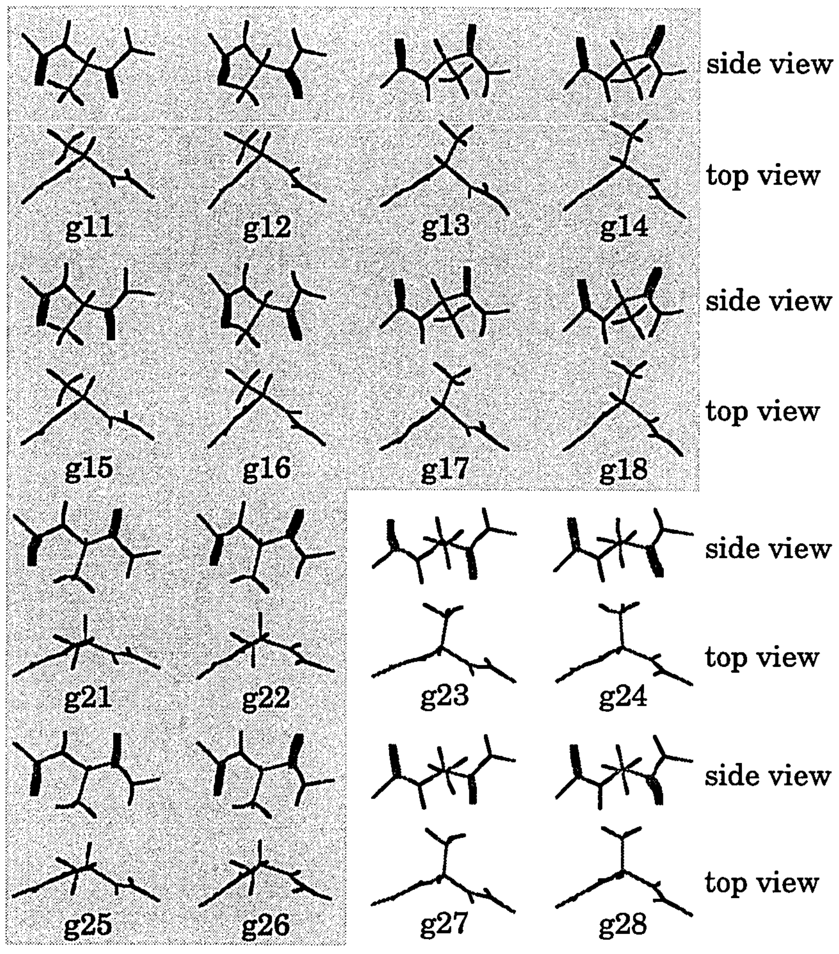
	g11	g12	g13	g14	g15	g16	g17	g18
ϕ°	91	91	-91	-91	82	82	-82	-82
ψ°	73	41	-73	-41	73	41	-73	-41

Combination 2 Torsion Angle Solutions

	g21	g22	g23	g24	g25	g26	g27	g28
ϕ°	130	130	-130	-130	121	121	-121	-121
ψ°	-153	-121	153	121	-153	-121	153	121

3.2.3 Reduction of the Possible Torsion Angle Solutions

In order to determine the initial structure of gA, the number of possible (ϕ, ψ) torsion angles about each alpha carbon must be reduced. To do this a vector outside of the peptide plane is needed. C_α-^2H quadrupolar splittings have been measured (Lee and Cross, 1994; Lee *et al.*, 1995) and are well suited for this purpose. The Ala$_3$ C_α-^2H quadrupolar splitting has been measured at 195 kHz (Lee *et al.*, 1995). From the sixteen possible (ϕ, ψ) torsion angles, shown in Figure 3.3, for a single alpha carbon, only four, the g23, g24, g27 and g28 solutions, have calculated C_α-^2H quadrupolar



Possible Solutions

Figure 3.3: Possible Ala3 (ϕ, ψ) Solutions. All sixteen possible solutions consistent with the dipolar and chemical shift constraints for a single alpha carbon are shown. Only four of the solutions predict the C_{α}-^2H quadrupolar splitting consistent with the observed data and are shown with a white background. The remaining four solutions are very similar, but the orientations of the peptide planes with respect to the channel axis differ.

splittings about the central alpha carbon consistent with the observed data (Table 3.3). The upper eight solutions in Figure 3.3 have the C-O bonds in the same direction while the alternating C-O bond orientations in the lower eight solutions are consistent with β-type structures. The C_α-^2H quadrupolar splittings provide an experimental means to reduce the number of possible solutions, and the C-O bond directions in the four possible solutions confirm the β-type structural motif.

Table 3.3: Experimental C_α-^2H Quadrupolar Splittings Compared with the Sixteen Possible Ala_3 (ϕ, ψ) Solutions. The quadrupolar splittings are calculated as $\Delta\nu_{obs} = 0.75$ QCC ($3\cos^2\theta - 1$), where QCC (160.0 for this site) is the quadrupole coupling constant, $\Delta\nu_{obs}$ is the observed quadrupolar splitting and θ is the angle between the magnetic field and the quadrupolar interaction axis, in this case the C_α-H bond.
†These solutions are consistent with the observed quadrupolar splittings.

Solution	Calculated Quadrupolar Splitting (kHz) (bond orientation)	Calculated - Observed Quadrupolar Splittings (kHz)
g11	25 (59°)	170
g12	25 (59°)	170
g13	139 (148°)	56
g14	139 (148°)	56
g15	25 (59°)	170
g16	25 (59°)	170
g17	139 (148°)	56
g18	139 (148°)	56
g21	111 (81°)	84
g22	111 (81°)	84
g23†	206 (162°)	11
g24†	206 (162°)	11
g25	111 (81°)	84
g26	111 (81°)	84
g27†	206 (162°)	11
g28†	206 (162°)	11

3.2.4 Characteristics of the Remaining Diplanes

The four remaining diplanes are very similar. They are based on a single diplane direction cosine solution, and therefore differ only in the single plane normal orientations with respect to the magnetic field, as shown in the top views in Figure 3.3. The first plane in the diplanes is identical in the g23 and g24 solutions, as it is in the g27 and g28 solutions. The g23 and g27 solutions have identical second planes, as do the g24 and g28 solutions. All four of the solutions are right handed and β-helical. A very interesting aspect of these solutions is the orientations of the C-O bonds with respect to the channel. The g23 and g28 solutions have C-O orientations that alternate in and out of the channel, while the g24 and g27 solutions have the C-O orientations either all out or all in. The sidechain orientation is identical in the four structures, as shown in the side views in Figure 3.3. This reflects the fact that the calculated C_α-^2H quadrupolar splittings are identical in these solutions.

3.3 Building the Initial Backbone Structure

The initial structure is built by combining the possible diplane solutions for each alpha carbon with subsequent diplanes. The method by which the diplanes are linked to form an initial backbone structure is shown in Figure 3.4. One diplane shares a peptide plane with the next diplane. Only the diplanes that share an identically oriented peptide plane can overlap. In other words, for the (ϕ, ψ) solution for residue i to be matched with a (ϕ, ψ) solution for residue i+1 it is necessary that the shared peptide plane joining residues i and i+1 be identical for the two diplane solutions. This reduces the number of possible initial backbone structures from 4^n to 2^{n+1}. Each alpha carbon produces g23, g24, g27 and g28 torsion angle solutions. The solutions pair such that $g23_i$ pairs with either $g27_{i+1}$ or $g28_{i+1}$, $g24_i$ pairs with either $g23_{i+1}$ or $g24_{i+1}$, $g27_i$ pairs with either $g27_{i+1}$ or $g28_{i+1}$, and $g28_i$ pairs with either $g23_{i+1}$ or $g24_{i+1}$. For the fifteen amino acids in gA, this leads to 65,536 (2^{n+1}) initial structures.

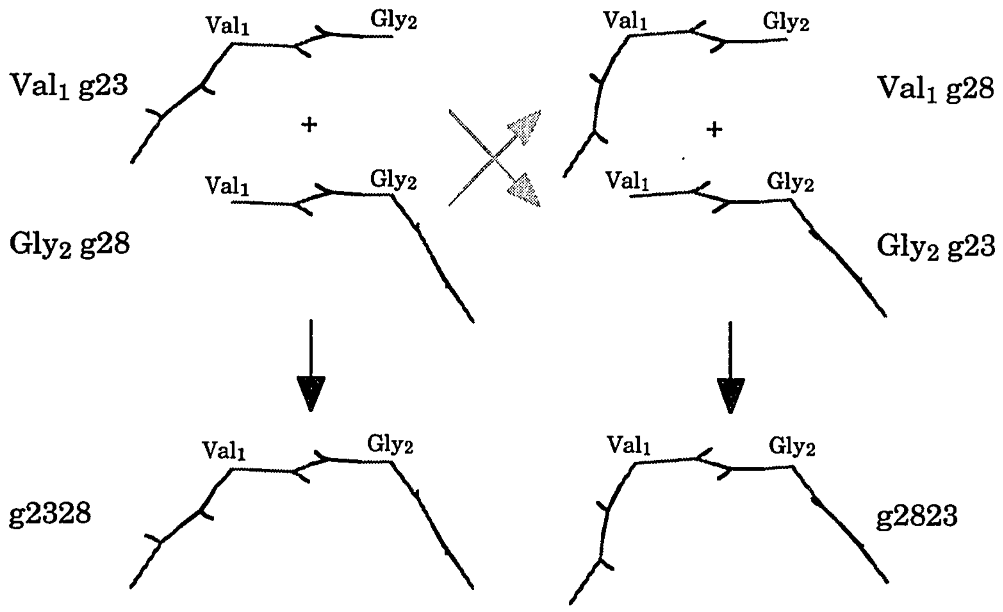


Figure 3.4: Peptide Plane Overlap. Although four (ϕ, ψ) solutions exist about each alpha carbon, only 2^{n+1} structures are possible. This is shown here in the method by which the diplanes are overlapped using shared diplanes for subsequent alpha carbons. Only those diplanes that share a plane with identical orientations can be overlapped to form a triplane structure. Solid arrows indicate identical orientations and gray arrows a mismatch.

The full array of initial structures all have the same structural motif and all display the same intramolecular hydrogen bonding pattern. The only difference between them is a sign ambiguity in the orientation of the peptide plane normal with respect to the channel axis. Based on this observation, four characteristic initial structures are all that need to be considered; g2328 has the carbonyls of the peptide planes oriented in such a way that they alternate in and out of the channel, g2424 has the carbonyls

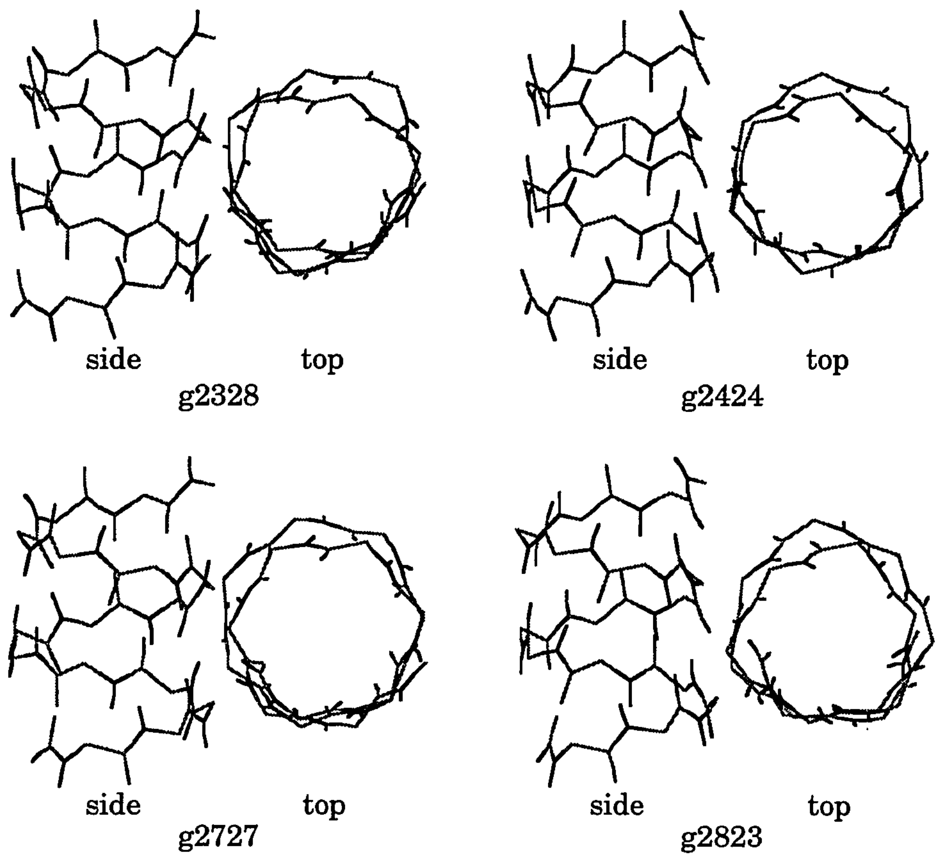


Figure 3.5: The Four Initial Backbone Structures. These four initial structures represent the orientational variability in the individual peptide plane orientations with respect to the channel axis. All remaining initial structures are simply permutations of the peptide plane orientations, not changes in the general structure. All initial structures are right handed β-helices with the intramolecular hydrogen bonding pattern clearly defined.

all out, g2727 has the carbonyls all in, and g2823 has the carbonyls alternating out and in. The naming scheme follows the one described previously, with the first two numbers describing the diplane on either side of residue i and the second two numbers describing the diplane on either side of residue i+1. The four initial backbone structures are shown in Figure 3.5. They are all right-handed β-helices and have approximately 6.6 residues per turn and a helical pitch of 4.8 Å. Since they were built with the same data, they all match the observed N-H and N-C_1 bond orientations and are reasonably close to the observed ^{15}N chemical shifts and C_α-^2H quadrupolar splittings. The intramolecular hydrogen bonds are clearly identifiable. Making full use of the experimental data and optimizing the hydrogen bonds is discussed in Chapter 4.

3.4 Addition of Sidechains to the Backbone

Although the determination of the sidechain structures did not fall within my research, the process by which they were obtained is described here in order to give an understanding of the method by which the entire initial structure is determined. The sidechain torsion angles are calculated using experimentally determined C-^2H quadrupolar splittings and ^{15}N constraints for the Trp indole rings from uniformly aligned samples. The quadrupolar splittings define the orientation of the C-H bonds with respect to the magnetic field and can be used to orient the sidechains with respect to the channel axis and peptide backbone resulting in the definition of the sidechain torsion angles. As stated earlier, the C_α-C_β bond is part of the initial backbone structure. The four initial structures, while containing different peptide plane orientations with respect to the channel axis, have identical C_α-H and hence C_α-C_β orientations. This means that the existence of four base initial structures does not lead to four separate sidechain structure solutions. The sidechain torsion angles are systematically varied along with a narrow range of C_α-C_β consistent with the backbone structure and an appropriate error bar. The rmsd between the observed and calculated quadrupolar splittings is used as a means by which the acceptable conformational states of the sidechain are determined. If multiple conformational states are shown at a single β_2 polar angle (Lee *et al.*, 1995), the solution that falls near a rotameric state is accepted. The Val_1 and Val_7 sidechain data represents fast exchange between rotameric states (Lee *et al.*, 1995). These sidechains are the only ones that show large amplitude local motions. Their static orientations are therefore represented by modified data such that only the dominant rotameric state is considered.

The final step in the initial structure determination is the joining of the backbone and sidechain structures. Since the backbone and the sidechains share C_α-C_β bond vectors, the sidechains can be placed directly on the backbone while maintaining their relative orientations with respect to the channel axis. Since the C_α-C_β bonds are the start of the various sidechains, the sidechains can be added to the initial backbone structure by continuing the covalent geometry of the individual sidechains from the beta carbon. Attaching the sidechains to the initial backbone structures leads to global initial structures. Since the sidechain orientations are determined independent of each other, interactions between the sidechains are not considered at this stage. Consequently, the initial structures do show VDW violations, and are discussed in Chapter 4.

3.5 Discussion

The method presented here has been shown to be successful in the determination of an entire initial protein structure in a membrane environment. The resulting structure is at a resolution adequate to define the peptide fold, identify the intramolecular hydrogen bonds and define the sidechain orientations. The ability of this method to obtain such a well defined structure with the limited amount of structural data is due to the quality of the orientational constraints obtained with SSNMR. The orientational constraints are at a very high precision (Hu *et al.*, 1993) and result in highly constrained peptide plane orientations. The absolute nature of the orientational constraints (i.e. the orientation of the molecular frame to the laboratory-fixed frame), as opposed to relative constraints such as distance measurements, allow for the determination of individual peptide plane orientations with respect to the external magnetic field without depending upon the entire structural solution. Each structural unit, the peptide plane, methylenes, methyls and indoles, are independent of the rest of the structure.

Since the initial structure is built using local orientational constraints, errors in the peptide plane orientations do not accumulate as the structure is sequentially assembled. This approach is fully capable of elucidating polypeptide secondary structure and, because the errors are small, tertiary structure should be effectively constrained as well. This result demonstrates the ability of SSNMR derived orientational constraints to accurately define three dimensional polypeptide structure.

The existence of four initial structures, the intramolecular hydrogen bonds, the sidechain VDW interactions, and the experimental data not used in the determination of the initial structure are all characteristics of the initial structure that demonstrate a need for a computational refinement procedure. Such a procedure is discussed in Chapter 4.

CHAPTER 4
COMPUTATIONAL REFINEMENT

4.1 Introduction

While solid state nuclear magnetic resonance spectroscopy (SSNMR) methods have been demonstrated for obtaining three dimensional structures of membrane bound polypeptides (Cross and Opella, 1983; Opella *et al.*, 1987; Ketchem *et al.*, 1993), computational refinement methods are needed for optimally utilizing these constraints in such a molecular environment. Methods for structural determination and refinement of macromolecules in solution have extensively evolved (Clore *et al.*, 1985; Havel and Wüthrich, 1985; Brünger *et al.*, 1986), but the nature of the constraints obtained for membrane proteins are such that a new refinement procedure must be developed. Described here is such a procedure that has the ability to optimize the structure of a membrane protein solved by SSNMR in order to best represent the experimental data and determine its high-resolution structure.

Chapter 3 introduced the means by which the initial gA structure is obtained. The initial backbone structure has many positive attributes. The use of local orientational constraints in the determination of the initial backbone structure leads to a structure whose local and intermediate range interactions are well defined. Even though the nature of the data is such that the constraints are local to the individual peptide planes, the initial backbone structure displays a high level of intermediate range conformational consistency. This is evident in that the secondary structure of the molecule is shown as a right handed β-helix and that the intramolecular hydrogen bonding pattern is clearly identified throughout the structure. The initial backbone structure is therefore in an appropriate starting position from which to consider further experimental constraints and a computational refinement protocol. Moreover, computational refinement is necessary in order to join the initial backbone and sidechain structures while incorporating all of the structural data and minimizing the structural energy.

Chapter 3 describes four initial structures that represent the extremes of conformational space for the possible initial structures. This chapter describes the computational refinement procedure used to refine these structures. The four base initial structures are all used as starting structures for the refinement procedure in Chapter 5 for determining the final structure. In this chapter the procedure using a single initial structure, g2328, is described.

4.1.1 General Strategy

The goal is to determine the three-dimensional structure of the channel that encompasses all the available data and to relax the covalent geometry used in the determination of the initial structure. However, when all data are weighted equally in the structure determination, the situation corresponds to a complex multidimensional optimization problem. The strategy for refining the structure involves setting up a generalized global penalty function, incorporating all available information extracted from SSNMR and other chemical information about the local geometries and energies of polypeptides. The refined structure is obtained from a geometrical search guided by minimizing this global penalty function. To perform the minimization in such a high-dimensional configurational space, an effective computational technique called simulated annealing is used (Metropolis *et al.*, 1953; Kirkpatrick *et al.*, 1983). Modifications in the structure are made by allowing the complete geometry of the polypeptide to vary. During this structural modification, the penalty function incorporates all available experimental data describing the backbone and sidechains, the intramolecular hydrogen bond distances and the full CHARMM empirical energy function.

4.1.2 Characteristics of the Initial Structure Indicating a Need for Refinement

4.1.2.1 Consideration of the experimental structural data. While the initial backbone structure defines the peptide fold, identifies the intramolecular hydrogen bonds and defines the sidechain orientations relative to the backbone, it does so by using a subset of the SSNMR data that has been acquired. The torsion angles for the backbone structure are calculated solely from the ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings, without consideration of the ^{15}N and $^{13}C_1$ chemical shifts or the C_α-^{2}H quadrupolar splittings other than as structural filters. Since the backbone structure is built using the dipolar splitting data, the structure meets this data. The calculation of all the acquired data from the initial structure shows that minor modifications in the initial structure are necessary in order to move the structure to within the experimental error of the observed data.

The sidechain structures are determined by making full use of the observed C-^{2}H data for each individual sidechain. However, sidechain torsion angles are determined relative to the backbone structure and, therefore, changes in the backbone structure will cause changes in the sidechain torsion angles to achieve the same C-^{2}H bond orientations with respect to the magnetic field. Also, the tryptophan indole ^{15}N chemical shifts and ^{15}N-^{1}H dipolar splittings have been observed (Hu *et al.*, 1993) and are included in the refinement to constrain the tryptophan orientations.

4.1.2.2 Consideration of structural aspects. Further aspects of the initial structure indicate that structural modifications are necessary in order to achieve an energetically reasonable global structure. The intramolecular backbone hydrogen bonds are clearly identified in the initial structure, but do not exhibit optimal hydrogen bond characteristics (Figure 4.1). Also, the sidechains are initially considered as independent local structures, so when they are included as extensions of the backbone structure, VDW overlap becomes evident between some of the sidechains (Figure 4.1). Another aspect of the initial structure is the assumption that the peptide linkage is planar (ω torsion angle = 180°). Preliminary experimental results suggest that a significant deviation from planarity exists for many of the peptide linkages. Analysis of the data calculated from the initial structure also indicates that deviations from planarity in the peptide linkages must be introduced in order to bring the structure within experimental error of all the observed SSNMR data. Another assumption in the determination of the initial structure is a static, identical geometry for the bond lengths and bond angles across all amino acid types. Analysis of various high resolution structures (Engh and Huber, 1991) indicates that the atomic geometry must be relaxed. Also, the gA channel exists as a head to head dimer. Interactions between the upper and lower monomers of the dimer must be considered in the refinement.

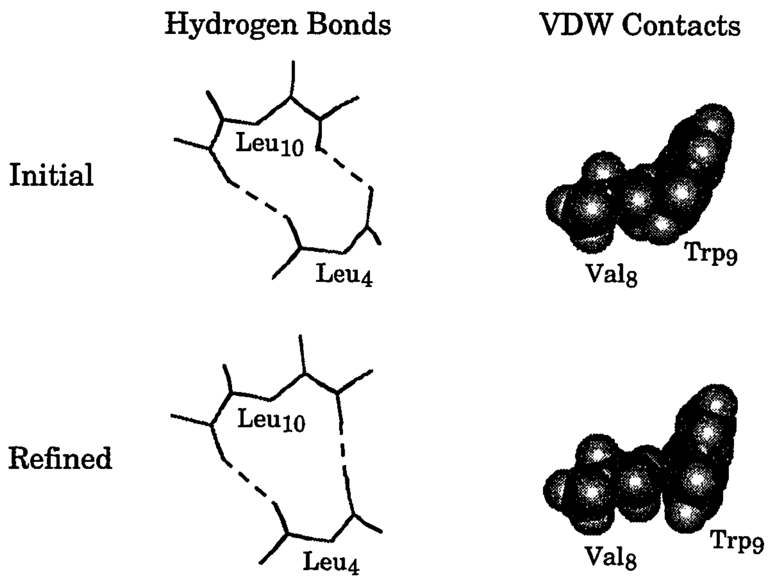


Figure 4.1: Hydrogen Bonds and VDW Interactions. An example of the hydrogen bonds and VDW interactions before and after computational refinement are shown. The initial backbone structure identifies the intramolecular hydrogen bonds and the sidechain orientations. Refinement is necessary in order to optimize these interactions.

It is evident that a suitable starting structure has been found, but computational refinement is necessary in order to meet the wealth of experimental data, to optimize the hydrogen bonds and VDW interactions, and to remove some of the assumptions made in calculating the initial structure.

4.2 Structural Refinement Using Simulated Annealing

4.2.1 Application of Simulated Annealing

The application of simulated annealing to this global optimization problem requires a definition for the system configuration, a method by which the configuration is varied, and a penalty function by which the structural variations are controlled. In this case, the system configuration is defined as the atomic coordinates of the peptide, from which a description of the structure can be obtained. Many methods for introducing structural variations exist, such as changes in the torsion angles or variations in the individual atomic coordinates themselves. A penalty function for controlling the structural variations is required. The primary focus of the penalty function is to refine the structure such that the experimental observations are met. To this end, the penalty function is composed primarily of the experimental data. Since direct modification of the individual atomic coordinates is one of the methods used for structural modification, functions that describe the atomic interactions are needed and must be included in the penalty function along with the experimental data.

4.2.2 Preparation of the Initial Structure

As stated earlier, the initial structures exhibit steric VDW contacts to a degree requiring structural modification. The VDW interactions for the leucine sidechains cause a substantial increase in the structural energy. In order to alleviate the worst of the bad contacts, the Leu_{10}, Leu_{12} and Leu_{14} sidechains and the ethanolamine were energy minimized using 250 steps of Adopted Basis Newton-Raphson (Brooks *et al.*, 1983) minimization within CHARMM. The orientational constraints for the resulting minimized sidechains were calculated and compared to experimental data. These structures were found to be in an acceptable orientation from which to begin the refinement procedure.

4.2.3 Structural Modification Strategy

Structural modification using both atom and torsional moves is required. Atom moves effectively modify the local structure and torsional moves efficiently search the conformational space. Even with a high diffusion parameter and a high starting temperature, atom moves alone are inhibited by the high energy contribution to the penalty resulting from VDW interactions. Including compensating and tunneling peptide plane moves along with atom moves allows for structural changes that search the conformational space defined by the range of initial structures discussed in Chapter 3.

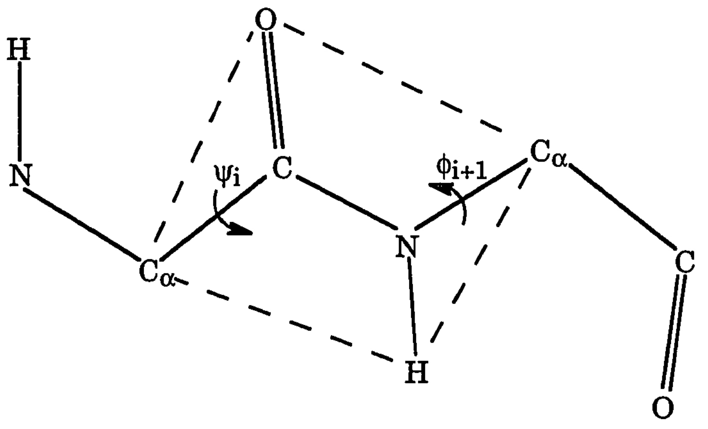


Figure 4.2: Compensating Peptide Plane Moves. By modifying the ψ_i torsion angle the same magnitude as the ϕ_{i+1} torsion angle, but in the opposite direction, the peptide plane orientation alone is modified without significant modification of the rest of the structure. By using this as one of the types of moves during the refinement procedure, the conformational space defined in the vicinity of the starting structure is searched.

4.2.3.1 Compensating moves. Compensating moves (Peticolas and Kurtz, 1980), illustrated in Figure 4.2, are performed by first choosing a random backbone ϕ or ψ torsion angle for modification. The degree of modification is chosen randomly in the range of -3.0° to +3.0°. The total change in an individual peptide plane is not constrained to fall within the defined limit, however, since a single plane may be chosen multiple times during a refinement. The compensating move is implemented by moving the chosen torsion angle the chosen amount and then moving the compensating torsion angle the same amount in the opposite direction. This has the affect of moving the peptide plane as a unit and thereby providing a means by which the peptide plane conformational space is readily searched without greatly distorting the helical and energetic parameters.

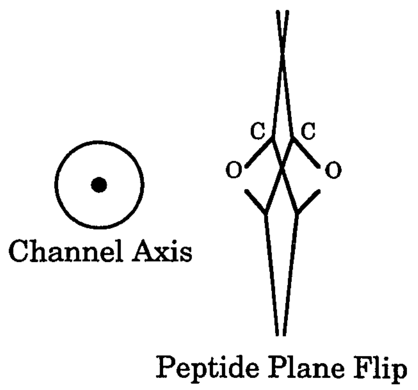


Figure 4.3: Tunneling Peptide Plane Moves. As in compensating moves, the peptide plane can be altered with respect to the channel axis without significantly affecting the rest of the structure. In order to search both possible orientations of the peptide plane normal with respect to the channel axis during refinement, tunneling moves are used to flip the peptide plane. The magnitude of the compensating move used is calculated from the orientation of the carbonyl bond with respect to the channel axis.

4.2.3.2 Tunneling moves. Tunneling moves are designed to search the conformational space that is most consistent with the experimental data. A random peptide plane is chosen and it is moved so that the sign of cosθ, the angle of the normal to the peptide plane formed by C_{α}-C=O with respect to Z, is changed. Like the compensating moves this involves simultaneous changes to ϕ_i and ψ_{i+1} of opposite magnitudes.

While compensating and tunneling moves are used as a means for moving out of local minima in the peptide plane orientations, they are primarily used to sample the peptide plane conformational space defined by the initial structures described in Chapter 3. This allows the refinement procedure to produce a single final structure while starting from each of the four base initial structures.

4.2.3.3 Atom moves. Atom moves are central to the refinement procedure. While the peptide plane moves search the conformational space defined by the initial structures, atom moves are responsible for relaxing the atomic geometry and minimizing the global penalty. The Cartesian coordinates are altered by introducing random displacements for each spatial dimension in either the positive or negative direction. By using a relatively small diffusion parameter, 0.0005 Å, as the magnitude within which the atoms move in each dimension, atom moves search a small conformation space. A result of the use of atom moves is that a precise structural agreement to the experimental data can be found while at the same time satisfying the constraints on the peptide geometry.

4.2.4 Penalty Function

The penalty function used to control the structural refinement is the sum of the structural penalties plus the energy.

$$Total\ Penalty = \sum_{i=1}^{M} \left(\lambda_i \cdot Structural\ Penalty_i \right) + \lambda_E \cdot Energy \tag{4-1}$$

where M is the number of structural penalties and λ is a scaling factor. The individual structural penalties are calculated as:

$$Structural\ Penalty = \sum_{j=1}^{N} \frac{1}{2} \left(\frac{Calculated - Observed}{Experimental\ \ Error} \right)^2, \tag{4-2}$$

where N is the number of measurements of a specific data type.

The use of the experimental error in the definition of the total penalty serves several purposes. One use is to equate the various data types used in the penalty, as shown in Figure 4.4. The data originates from several different observations, such as the chemical shift frequency and quadrupolar splittings. These different data types are associated with different observed magnitudes and are therefore difficult to equate directly. Each experimental error is of a magnitude relative to the observed interaction size and, therefore, division by the error has the result of both scaling the different data types so that they contribute equally to the total penalty and making the penalty for the individual data types dimensionless. The penalty functions resulting from various data types indicate that the different penalties are of equal magnitude as a function of the error. Also, it is important to have the ability to define separate error values within a particular data type, since experimental error may vary from site to site depending on the quality of the data. Incorporating the error into the penalty function allows for the appropriate quality of the structural constraint during the refinement procedure, as shown in Figure 4.5.

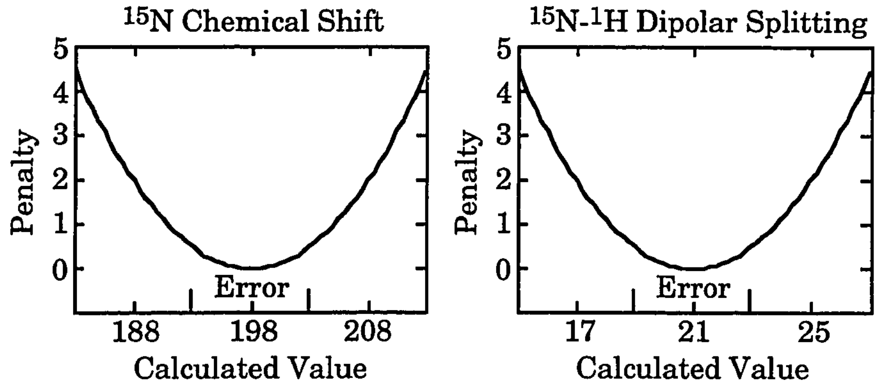


Figure 4.4: Penalty Function with Different Data Types. Even though the different contributors to the penalty originate from different interactions, the individual penalties generated from them as a function of deviation from experimental error are identical. This serves to equate the various data types so that no one interaction has undue influence on the refinement.

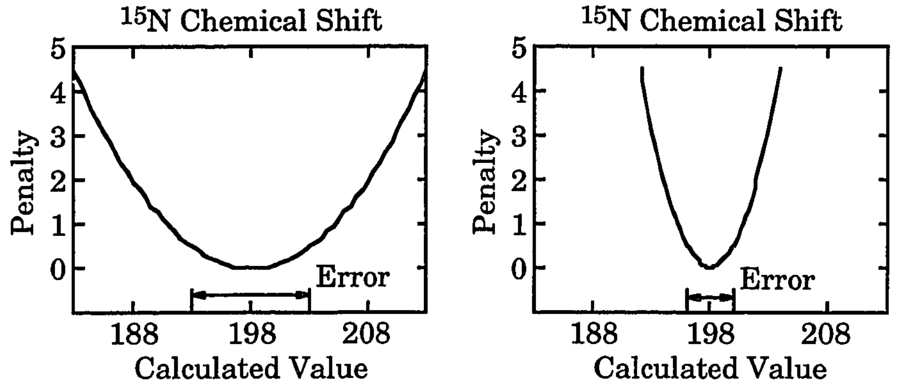


Figure 4.5: Penalty as a Function of Experimental Error. The experimental error used for a particular observation has an effect on the penalty for that site. Shown here are two different penalty plots for the same observation. The difference between them is the value of the experimental error used to calculate the penalty. The smaller the error, the faster the penalty increases as the structure deviates from the observed value.

4.2.5 Structural Constraints Used

The constraints imposed on the structure during refinement are fifteen ^{15}N chemical shifts, two $^{13}C_1$ chemical shifts, fourteen ^{15}N-$^{13}C_1$ dipolar splittings, fifteen ^{15}N-^{1}H dipolar splittings, twelve C_α-^{2}H quadrupolar splittings, fifty-four C-^{2}H quadrupolar splittings, four indole ^{15}N chemical shifts, four indole ^{15}N-^{1}H dipolar splittings, ten N-O and ten H-O hydrogen bond distances, and the energy, for a total of one hundred and forty-one constraints. Appendix A.1.2 lists the individual experimental constraints.

The ^{15}N and $^{13}C_1$ chemical shift tensors are characterized from unoriented samples so that the magnitudes of the tensor elements and the orientation of the tensor with respect to the molecular frame can be determined (Mai *et al.*, 1993). The chemical shifts observed from oriented samples are compared to chemical shifts calculated using the molecular coordinates and the tensor characteristics. A change in the orientation of the atomic coordinates leads to a change in the calculated chemical shifts and a resultant change in the penalty.

The ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings and the ^{2}H quadrupolar splittings are observed in oriented samples and reflect the orientation of the unique interaction tensor element, the internuclear vector, with respect to the external magnetic field. The magnitude of the dipolar interaction, $\nu_{||}$, is calculated as described in Chapter 1, whereas the magnitude of the quadrupolar interaction, QCC, is experimentally defined from model compound studies. The dipolar and quadrupolar splittings can be calculated from the atomic coordinates and compared to the observed splittings during refinement.

The initial structure identifies the intramolecular hydrogen bonds. The hydrogen bonds are therefore included as direct contributions to the penalty function as is routinely done in structural refinements from solution NMR (Case and Wright, 1993; Logan *et al.*, 1994). During the refinement the internuclear distances associated with the intramolecular hydrogen bonds are calculated and compared to accepted H-O and N-O internuclear distances for β-sheet structures with values of 1.96 ± 0.3 Å and 2.91 ± 0.3 Å, respectively (Jeffrey and Saenger, 1994). The large range of acceptable distance values serves to constrain the hydrogen bonding without distorting the final structure.

The all-atoms PARAM22 (Mackerell *et al.*, 1992) version of the force field of CHARMM was used to describe the internal energy (bonds, angles, dihedrals), as well as the non-bonded interactions (VDW and electrostatics). The IMAGE facility of CHARMM was used to impose the dimer symmetry of the gramicidin channel about X, and the MMFP facility was used to impose the channel axis along the Z axis of the coordinate system. The VDW and electrostatic non-bonded interactions were calculated on the basis of a group-based pair list. The interactions were smoothly truncated at a distance of 10 Å using a 2 Å switching function. A dielectric constant of 1 was used.

4.2.6 Annealing Strategy

The simulated annealing refinement procedure was performed according to the Metropolis Monte Carlo algorithm (Metropolis *et al.*, 1953). Acceptance of an attempted move is controlled by both the temperature and the difference in the penalty before and after the attempted move. A move which causes a decrease in the penalty is always accepted. A move which increases the penalty, however, is only sometimes accepted. The choice to accept an uphill move is made by first choosing a random number between 0 and 1. If this random number is less than exp(-Δpenalty / T), the uphill move is accepted. This is to say that exp(-Δpenalty / T) defines the probability of the uphill move. The higher this probability, the greater chance of choosing a random number that is less than the calculated probability and therefore accepting the move.

4.2.6.1 Control temperature and annealing schedule. The simulated annealing refinement procedure is controlled by a temperature parameter and an annealing schedule. The global minimization is controlled by an annealing schedule, i.e., the rate at which the temperature is lowered during the course of the refinement. The focus of this refinement strategy is to introduce minor structural modifications to the initial structure. Large changes would lead to conformational space that has already been shown to be excluded through the development of the initial structure described in Chapter 3. Therefore, the initial value of the temperature is set at 300K so that large structural changes are not possible. The system configuration undergoes 2000 modifications or 200 successful modifications, whichever is first, before the temperature is lowered by 1%. At the beginning of the refinement, however, an equilibration period of 5000 attempted steps is used during which the temperature remains constant. This allows for many structural modifications to occur at the relatively high starting temperature, extending the search of conformational space. In the beginning of the refinement, once the temperature begins being lowered, the temperature is dropped relatively fast as a function of attempted structural modifications since many successful moves are initially found.

As the refinement continues, the temperature is dropped less often since fewer accepted moves are found. The refinement is terminated when no successful structural modifications are found at a particular temperature.

4.2.6.2 Annealing parameters. Multiple parameters are used in the refinement procedure, such as the starting temperature, the diffusion parameter for the atom moves, the relative number of move types, and the λ values used to assign relative weights to the individual constituents of the penalty function. Combinations of various values for these parameters were used to determine the optimal conditions for refinement, taking into consideration the final fit to the experimental data, the final energy of the system, and the computational time required. The values for the optimized parameters are a starting temperature of 300K, a diffusion parameter of 0.0005 Å and a 0.5:0.2:0.3 ratio for compensating moves, tunneling moves and atom moves, respectively. The energy contribution tended to overwhelm the penalty calculated from the experimental constraints, so the λ values for all constituents of the penalty function other than the energy, for which λ was set to 1.0, were set to 3.0. Maintaining a λ value of at least 1.0 for the energy contribution is important since a λ value less than one corresponds to an effectively higher temperature and would lead to a non-realistic value for the energy of the refined structure, thus compromising the structural integrity.

4.2.6.3 Program considerations. The refinement procedure discussed here has been implemented in a program called TORC (TOtal Refinement of Constraints), which was written entirely in C. The source code for this program, as well as a Makefile for compiling the code and a sample input file for running the program within CHARMM, is included in Appendix

A.4. The general refinement procedure is shown in Figure 4.6. The code has been incorporated into CHARMM in order to take advantage of CHARMM's ability to calculate the structural energy for use in the refinement.

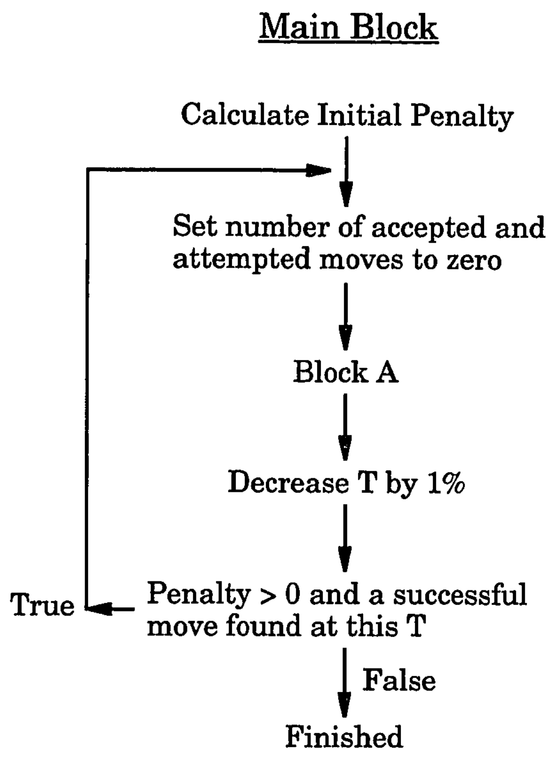

Main Block
Calculate Initial Penalty
Set number of accepted and attempted moves to zero
Block A
Decrease T by 1%
True
Penalty > 0 and a successful move found at this T
False
Finished

Figure 4.6: TORC Algorithm. This figure shows a schemcatic of the general algorithm used to drive the refinement procedure. This figure demonstrates the general means by which the refinement is executed. The source code for the entire refinement program, TORC, is shown in Appendix A.4.

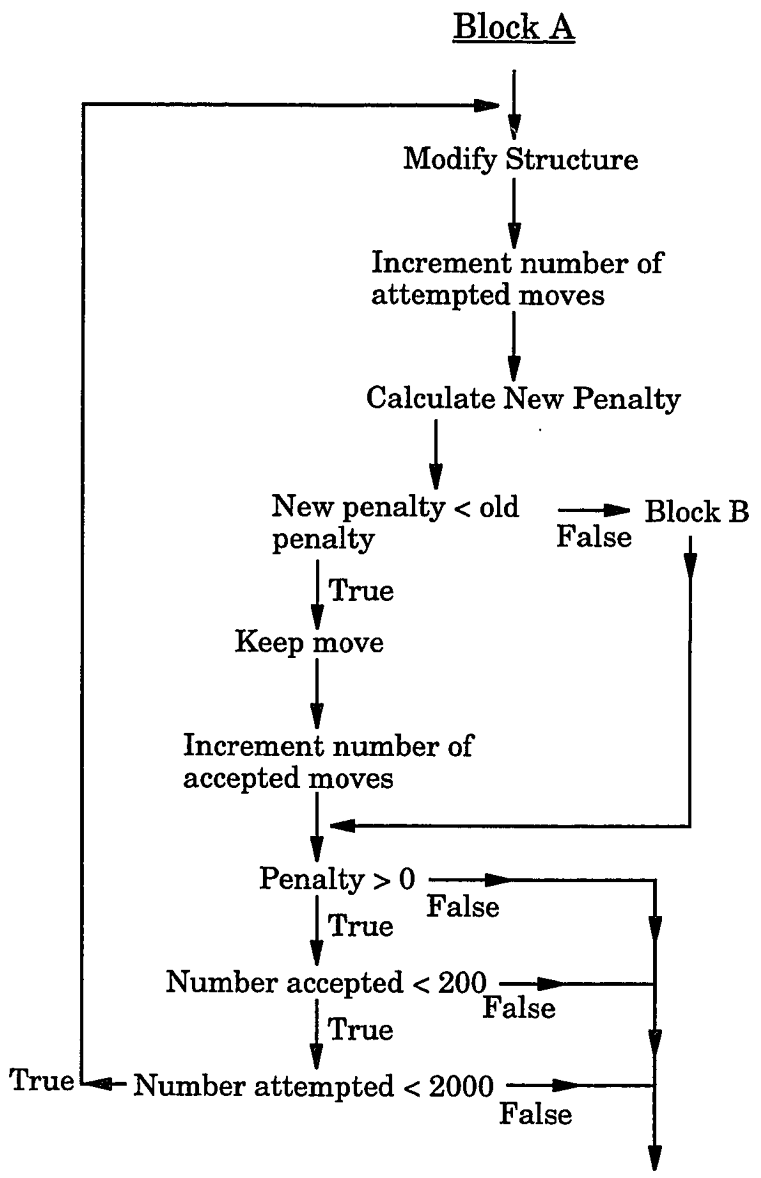


Figure 4.6: TORC Algorithm, continued.

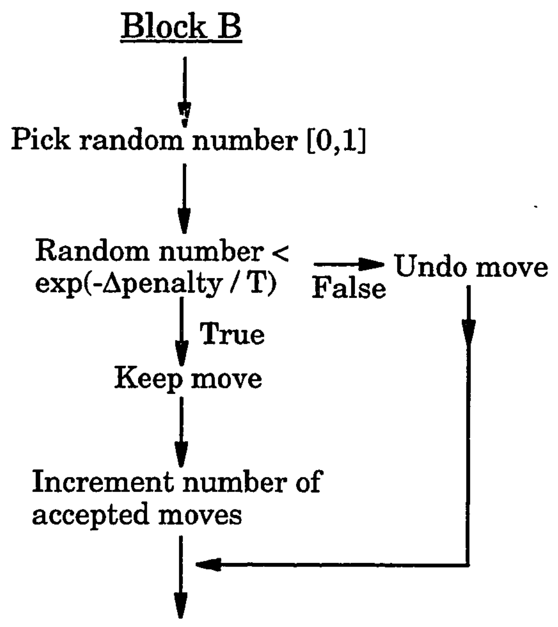

Block B
Pick random number [0,1]
Random number < exp(-Δpenalty / T)
False
Undo move
True
Keep move
Increment number of accepted moves

Figure 4.6: TORC Algorithm, continued.

4.3 Refinement Results

A typical single refinement requires approximately 12 CPU hours on a Silicon Graphics 4×R8000 Power Challenge running on a single CPU. During the course of the refinement some 550,000 attempted moves are made with approximately 90,000 of the moves being accepted. The number of various move types attempted and accepted are shown in Table 4.1. The majority of the accepted moves occur in the initial stages of the refinement when the temperature is high and the structure is in a high penalty state. The values of the individual contributions to the penalty function are shown in Table 4.2.

Table 4.1: Attempted and Accepted Refinement Moves. The various refinement moves are shown along with the number of each that was attempted and accepted during the refinement. 16% of the total of attempted moves were accepted, with a majority of these being atom moves. For this refinement, 50% of the attempted moves were compensating plane moves, 20% were tunneling plane moves and the remaining 30% were atom moves.

Move Type	Accepted	Attempted	Accepted/ Attempted	Accepted/ Total	Attempted/ Total
Compensate	41789	280955	0.15	0.07	0.50
Tunnel	806	112297	0.01	0.00	0.20
Atom	47861	169336	0.28	0.09	0.30
Total	90456	562588	0.16	0.16	1.00

Table 4.2: Penalty Distribution for the Initial and Refined Structures. The penalty values shown have been multiplied by the λ values indicated and therefore reflect the final penalty used during refinement and not the calculated penalty of the individual constraints. The penalties are decreased in all cases except for the ^{15}N-$^{13}C_1$ and ^{15}N-^{1}H dipolar splittings. Since the initial structure was built using this data, it matches this data. Refinement causes deviations from this in order to meet the other constraints, as is to be expected.

Penalty Type	λ	Initial Penalty	Refined Penalty
^{15}N Chemical Shift	3.0	23.1	1.2
^{13}C Chemical Shift	3.0	0.4	0.1
^{15}N Indole Chemical Shift	3.0	7.2	0.2
^{15}N-$^{13}C_1$ Dipolar Splitting	3.0	0.2	1.7
^{15}N-^{1}H Dipolar Splitting	3.0	0.4	1.7
^{15}N-^{1}H Indole Dipolar Splitting	3.0	4.4	0.6
Distance	3.0	85.1	9.8
$C_{\alpha,\beta,\ldots}$-^{2}H Quadrupolar Splitting	3.0	4961.7	8.9
CHARMM Energy	1.0	3400.9	396.7
Total Penalty		8483.4	420.9

The refinement is represented structurally in Figure 4.7. The average helical pitch changes from 4.9 Å to 5.2 Å, and the average residues per turn changes from 6.8 to 6.7. The characteristic most easily observed in the structural representation of the refinement is the alleviation of undesired VDW contacts as seen in the altered sidechain orientations.

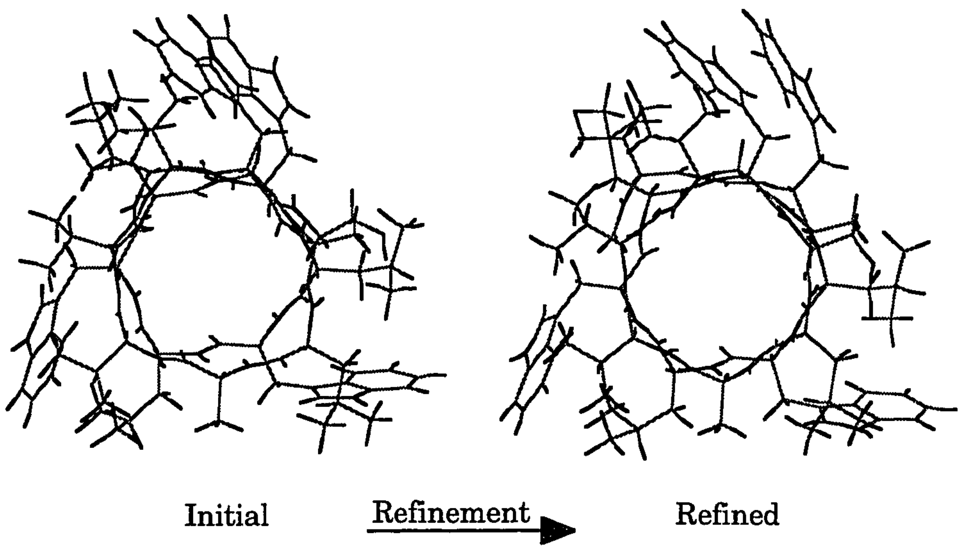


Figure 4.7: Structural Representation of the Refinement Procedure. The initial structure is refined by undergoing a series of structural modifications, such as peptide plane moves and atom moves, controlled by a penalty function that includes the SSNMR observables and the CHARMM energy. The resulting refined structure meets the observed data at a reasonable energy. The structures show that the hydrogen bonding and VDW interactions have been optimized.

4.3.1 Peptide Plane Orientations

As Figure 4.7 indicates, the peptide plane moves introduce substantial structural changes within the local conformational space defined by the initial structures. Table 4.3 shows the carbonyl orientations with respect to the channel axis for both the initial and refined structures. The initial structure used in this refinement example is g2328 which has the carbonyl orientations alternating in and out. The refined structure, however, has a changed carbonyl orientation pattern and indicates that the conformational space defined by the initial structures has been searched.

Table 4.3: Carbonyl Orientations. The carbonyl orientations with respect to the channel axis are changed dramatically by the refinement procedure. The initial structure begins with an alternating pattern of carbonyl orientations. The refined structure displays a pattern of carbonyl orientations that are mostly pointing into the channel. Carbonyls that are within ±3° of the channel axis are considered parallel to it. The orientations in the refined structure that have changed from the initial structure are indicated with †.

	Initial		Refined	
Residue	Orientation	Angle	Orientation	Angle
Val_1	in	12.1	out †	-17.0
Gly_2	out	-3.7	in †	7.1
Ala_3	in	15.4	in	13.7
Leu_4	out	-12.7	parallel †	2.7
Ala_5	in	14.9	parallel †	-1.7
Val_6	out	-6.5	parallel †	-2.5
Val_7	in	15.5	out †	-10.7
Val_8	out	-12.8	in †	5.8
Trp_9	in	22.7	parallel †	0.4
Leu_{10}	out	-13.7	in †	9.8
Trp_{11}	in	17.9	in	11.8
Leu_{12}	out	-13.4	in †	11.1
Trp_{13}	in	23.2	in	16.8
Leu_{14}	out	-12.6	in †	10.4
Trp_{15}	in	22.6	in	16.6

The attempted and accepted peptide plane moves per residue are shown in Table 4.4. Since compensating moves are small and introduce minor structural deviations, many are accepted. A large number of tunneling moves are also attempted, but most are rejected. The conformational space is being searched, but is found by the penalty function to be unacceptable. The number of tunneling moves that are accepted, however, are sufficient to determine a global structural solution independent of the initial structure, as will be shown in Chapter 5.

Table 4.4: Peptide Plane Moves Per Residue. The attempted and accepted peptide plane moves per residue indicate that many compensating moves are accepted, thus searching local conformational space. Though many tunneling moves are attempted, few are accepted. The conformational space is being searched, but is rejected.

	Compensating		Tunneling	
Residue	Accepted	Attempted	Accepted	Attempted
Val_1	1780	9667	6	7577
Gly_2	3756	19293	0	7536
Ala_3	3391	19299	2	7314
Leu_4	3486	19357	1	7370
Ala_5	3390	19920	0	7516
Val_6	3260	19474	0	7566
Val_7	3227	19532	1	7366
Val_8	2567	19647	1	7377
Trp_9	2715	19239	733	7411
Leu_{10}	2587	19175	1	7623
Trp_{11}	2254	19125	0	7569
Leu_{12}	2373	19256	51	7410
Trp_{13}	2456	19233	0	7559
Leu_{14}	2705	19275	10	7464
Trp_{15}	1842	19463	0	7639

A plot of the carbonyl orientations with respect to the channel axis is shown in Figure 4.8. Plot A shows the carbonyl trajectory for the entire refinement. Though the carbonyl begins by pointing out of the channel, peptide plane moves force a search of conformational space so that the carbonyl orientation is quickly changed a large degree. It is evident that at the start of the refinement the plane tunnels to its alternate orientation. Compensating moves provide a large degree of plane orientation modification, especially early in the refinement. Plot B is an expansion of the beginning of Plot A and illustrates the early plane reorientation.

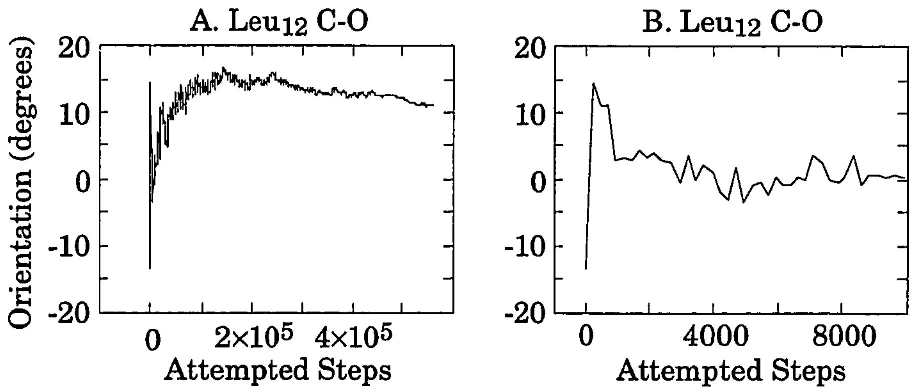


Figure 4.8: Backbone Carbonyl Orientation Trajectory. The backbone carbonyl orientation is dependent upon the peptide plane orientation. Changes in the peptide plane orientation during refinement by compensating and tunneling moves therefore affect the orientation of the backbone carbonyl. A is the full refinement and B is an expansion of the beginning of A. The carbonyl for this site is shown to begin the refinement by pointing out of the channel axis. Tunneling moves flip the plane orientation early, however, and compensating moves eventually brings the carbonyl well into the channel axis.

4.3.2 Refinement Trajectories

During the refinement procedure the calculated data for an individual amino acid and the values for the contributions to the penalty function are saved as a function of attempted moves. These trajectories serve to illustrate the refinement procedure since they show the fluctuations in the calculated data and the behavior of the energy and the data penalties during the course of the refinement. Trajectories for selected data types and the penalties resulting from the data are shown in Figure 4.9.

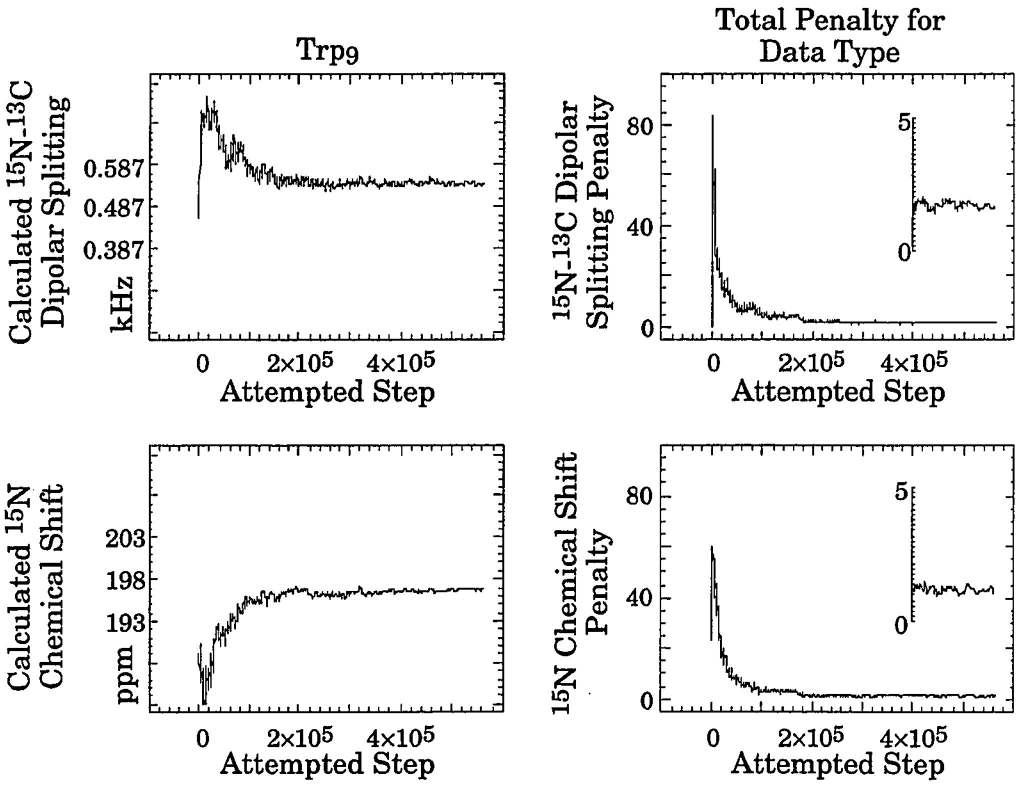


Figure 4.9: Refinement Trajectories for Selected Data Types. The left hand column illustrates the variability of the indicated SSNMR observable for a single site over the length of the refinement. The y-axis is labeled with the value for the experimentally observed interaction in the center and the range of the experimental error above and below the observed value. The right hand column is the summed penalty associated with the individual constraints. A magnification of the penalty trajectory tails are inset and follow the same value of the x-axis as the full trajectory.

Plots of the data values as a function of attempted moves, the left hand column in Figure 4.9, show the initial modification of the data, illustrating the level of the search of conformational space. The plots indicate that at the beginning of the refinement the structure can be modified to an extent that the calculated data moves well outside of the experimental error range for the observed data, thus allowing for structural deviation from a local minimum. As the refinement progresses, the structural modifications decrease in both amplitude and frequency and the calculated data moves toward an agreement with the observed data.

The data trajectories further indicate that the individual calculated data values are initially within or very close to the range of experimental error. Even for those sites that remain within experimental error during the entire refinement, structural modifications are apparent at the beginning of the refinement trajectory. The trajectories for the sites that lie closer to the edge of experimental error also show structural fluctuations at the beginning of the refinement and eventually settle within the experimental error.

Plots of the individual experimental data contributions to the penalty, the right hand column in Figure 4.9, show the ability of the refinement procedure to introduce structural modifications that significantly lower the penalty. At the beginning of the refinement, fluctuations in the calculated penalties illustrate the acceptance of both downhill and uphill structural changes. These structural changes are clearly directed toward a penalty minimum. Near the end of the refinement the stabilization of the penalty at or near zero for the individual contributors indicates that the refinement has succeeded in finding a structure that meets the experimental data, as previously shown in Table 4.2.

The energy trajectory for the refinement indicates that the initial structure has a relatively high energy. The major contribution to the high energy, 3401 kcal/mol, is undesired VDW contacts. The contacts are easily relieved early in the refinement and the energy contribution quickly reaches a state at which the driving force to minimize the penalty is the relaxation of the static geometry. Upon refinement, the energy moves to a reasonable 397 kcal/mol. The final energy is not due primarily to a single contribution, but is spread over all constraints contributing to the energy as shown in Table 4.5. The angle energy is high, though, and a few of the interactions have increased during refinement. This indicates that though the overall energy decreases significantly, the refinement is moving the structure so that the experimental observations are met.

Table 4.5: CHARMM Energy Distribution. Shown here is the CHARMM energy distribution before and after refinement. The full CHARMM energy is used as a constraint. The initial structure is at a high energy due mainly to steric VDW contacts. Refinement successfully alleviates these undesired interactions.

Energy Type	Initial	Refined
Bonds	45.1	18.1
VDW	2331.9	5.5
VDW (image)	601.8	-7.5
Electrostatics	102.9	92.3
Electrostatics (image)	73.1	67.9
Angles	109.8	122.1
Urey-Bradley	8.9	15.0
Dihedrals	72.4	78.6
Impropers	0.1	4.0
MMFP	55.0	0.9
Total	3401.0	396.9

Although the trajectories show that the refinement procedure is searching conformational space, the space is limited and biased to the regions in the vicinity of the initial structure and the experimental ambiguities. The all atom rmsd of 1.31 Å between the initial and final structure indicates that there is a close similarity between the initial and final structures and that only slight modifications of the initial structure were needed to achieve compliance with all constraints. The major conformational search occurs in the movement of the peptide plane orientation and not in the overall structural motif. The initial structural motif is defined analytically by the initial structure. Deviation from this motif is therefore undesired and would be a waste of computational effort.

4.3.3 Hydrogen Bonds

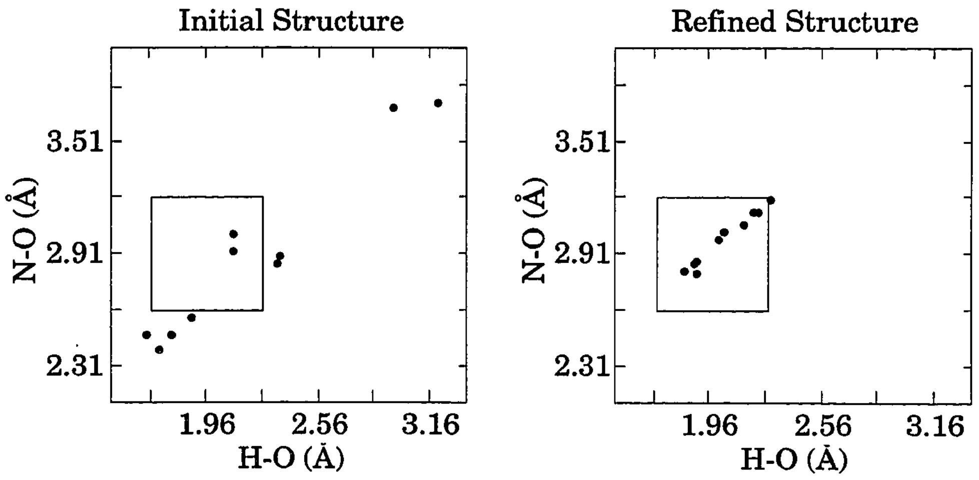


Figure 4.10: Intramolecular Hydrogen Bond Distances. The initial and refined hydrogen bond distances are shown. The box indicates the range of allowed H-O and N-O distances and is centered on the accepted H-O and N-O values. The initial structure identifies the intramolecular hydrogen bonds, but refinement is necessary in order to bring the hydrogen bond distances to within the range of accepted values.

The backbone hydrogen bond distances are shown in Figure 4.10. As the figure indicates, the hydrogen bonds in the initial structure, though close to the accepted values, indicate that structural refinement is necessary. Out of ten hydrogen bonds, only two are within the accepted H-O and N-O limits. Including the hydrogen bonds in the penalty function during refinement induces structural changes that bring the hydrogen bonds within the accepted range except for one, which lies very close to the edge of the range.

4.3.4 Structural Geometry

4.3.4.1 Omega torsion angle.

The refinement procedure has a substantial affect on the geometry of the peptide linkages by introducing significant deviations from planarity in several of the ω torsion angles. The refinement includes the energy and, therefore, the CHARMM force field is imposed on the ω torsion angles along with the experimental constraints. As a result, the average deviation from planarity for the ω torsion angles is 5.6° with the largest deviation being 17.6° and three deviations over 10°. These results show that even in the presence of the CHARMM force field the structural constraints result in significant non-planarity for the peptide linkages.

4.3.4.2 Influence of the CHARMM energy.

The incorporation of the CHARMM energy into the penalty function for the refinement has multiple effects on the final structure. As well as providing a means by which undesired VDW interactions are removed, the energy imposes subtle changes in the covalent geometry to match the definition set by the CHARMM force field. The initial structure maintains a static geometry using accepted values for bond lengths and angles (Fletterick *et al.*, 1971;

Momany *et al.*, 1975; Kvick *et al.*, 1977; LoGrasso *et al.*, 1989; Teng *et al.*, 1991). Constant values are used throughout the initial structure without consideration of the amino acid type or external interactions. The geometry of the refined structure, however, has been relaxed and therefore contains bond lengths and bond angles that more accurately correspond to their local environment.

The relaxation of the covalent geometry is evident in the consideration of the N-C bond lengths and the N-C_α-C bond angles. The N-C bond lengths are all set to an initial value of 1.340 Å. Refinement causes lengthening and shortening of these bonds for different amino acids with changes distributed from -0.015 Å to 0.024 Å, the average final N-C bond length remained 1.34 Å. The N-C_α-C bond angle is changed an average of 0.6° from 110.0° to 109.4°, and ranges from 105.5° to 113.9° for the different amino acids. These changes in the covalent geometry indicate that the local amino acid environments are taken into consideration during the structural refinement.

4.4 Discussion

The refinement procedure described here has been demonstrated to introduce minor structural modifications leading to a structure that encompasses the experimental data and the calculated structural energy. The simultaneous use of the experimental data and the energy as contributors to the penalty function produces a refined structure that satisfies well all imposed constraints without being biased toward either the data or the energy.

The characteristics of the initial structure that prompted the development of a computational refinement procedure have been successfully addressed. The refinement provides the means to obtain a global structural solution that meets the imposed constraints. The refined backbone structure is defined not only by the ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar interaction vectors used to analytically calculate the initial structure, but by all of the experimental backbone data. The refined global structure includes both the backbone and sidechains and therefore conforms to the complete set of experimental data used. The imposed hydrogen bond distance constraints are also met in the refined structure and result in a well defined helix. The inclusion of the energy in the refinement served to remove undesired VDW contacts in the structure and to relax the covalent geometry without adversely affecting the structural match to the experimental data.

The use of SSNMR as the experimental method by which the structural constraints are obtained has the advantage of defining the global peptide structure in terms of local structural units. The orientations of these units are individually determined and the initial structure can be analytically determined from these quantitative constraints without the need for a large number of qualitative constraints as is required for other methods, such as solution-state NMR. All that is required of the computational refinement is minor modifications within the local conformational space in order to best fit additional constraints imposed upon the structure.

Apparent in the refined structure is the fact that the final penalty from the experimental data is not zero, indicating that while the refined structure lies very close to the experimental data, the structure does not absolutely meet the data. The structure lies within the bounds of experimental error, however. Many reasons for a non-zero penalty exist, of which some can be addressed in the future and others are inherent in the method by which the data is obtained.

One contribution to a non-zero data penalty that can be addressed is the parameters used in the calculation of the experimental data from the atomic coordinates. For the calculation of the chemical shifts as a function of the tensor orientation with respect to the external magnetic field, the tensor elements and the tensor orientation with respect to the molecular frame are characterized from dry powders while the observed oriented chemical shifts are determined in fully hydrated lipid bilayers. In order to obtain more accurate calculated chemical shifts the tensors should be characterized in fully hydrated lipid bilayers to better represent the channel conformation of this peptide (Lazo *et al.*, 1993; Lazo *et al.*, 1995).

Two other parameters are the magnitudes of the dipole interactions, $\nu_{||}$, used in the calculation of the ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings from the atomic coordinates. These dipolar splittings, which serve to orient the N-H and N-C bond vectors, are proportional to the values of $\nu_{||}$. As stated earlier, $\nu_{||}$ is proportional to the product of the gyromagnetic ratios for the two nuclei and is inversely proportional to the cube of the internuclear distance. The internuclear distance is set to a static value for calculating $\nu_{||}$. This value for $\nu_{||}$ is then used without modification throughout the refinement procedure, even though atom moves vary the internuclear distance. This results in the calculated dipolar splitting being dependent upon the orientation of the bond vector alone and not upon the internuclear distance.

The values of QCC used to determine the C-^{2}H bond orientations are determined from model compounds. Although local dynamic averaging is taken into consideration for individual sites, the QCC values used could be improved. Experimental refinement of these values would lead to better defined C-^{2}H bond orientations.

Furthermore, molecular dynamics are present in the sample (Nicholson *et al.*, 1991; Lazo *et al.*, 1993; North, 1993; North and Cross, 1993; Lazo *et al.*, 1995) that motionally average the observed experimental data. The amplitude and axis of the local backbone motions have been determined for many sites and these motions will affect the experimental constraints. Incorporating motional averaging will lead to more realistic bond orientations.

Molecular fluctuations occurring in the sample may be such that it is not possible to accurately represent all of the structural data in a single, static structure. In fact, what is refined is a motionally averaged structure, representing motional averaging that occurs on a time scale of msec or less. An accurate simulation of the molecular dynamics based on the experimentally observed backbone and sidechain dynamics would allow for the calculation of the time averaged observables and could be used to determine a structure that meets the motionally averaged experimental structural constraints.

The inclusion of the structural energy, while maintaining the covalent geometry and removing undesired VDW contacts, influences the structure in a way that may not correspond to the influence due to the experimental data. The energy is calculated for the structure in a vacuum and will therefore bias the structure. As a result, the penalty due to the experimental data is reduced to a point that is very close to zero but cannot be further reduced. At the same time, the calculated energy is significantly reduced during refinement but not to the extent possible in the absence of experimental constraints. This indicates that although the penalties from the two different types of structural constraints are both necessary and are reduced during refinement, they compete for control of the structural modifications.

The final structure obtained through this refinement procedure satisfies well the constraints imposed on the structure. The final penalty is very low and reflects the ability of the refinement to introduce structural modifications in such a way as to satisfy imposed constraints such as energy and hydrogen bonds while simultaneously satisfying the experimentally derived constraints.

This chapter has described the refinement procedure of a single initial structure. In order to obtain a final, refined structure, all four of the base initial structures must be used as starting structures and a refinement ensemble generated consisting of multiple refinements for each initial structure. The results of the individual refinements must then be averaged, and the average structure refined. The implementation of this strategy and its result is the topic of discussion in Chapter 5.

CHAPTER 5
DETERMINATION AND ANALYSIS OF THE FINAL STRUCTURE

5.1 Introduction

The structure of gA in hydrated DMPC bilayers has been successfully solved as an initial structure and a computational refinement method developed that takes advantage of both the large number of SSNMR constraints available for this peptide and the full energy described by the CHARMM force field. Refinement of a single structure has shown that the SSNMR observables are met to within the limits of experimental error, the intramolecular hydrogen bonds and the VDW interactions optimized, and the static geometry of the initial structure relaxed. During the refinement of a single initial structure, the alternating pattern of the carbonyl orientations with respect to the channel axis was disrupted, bringing many of the carbonyl oxygens into the channel. This is reasonable considering that the function of gA in membranes is the transportation of monovalent cations that could be solvated by the carbonyl oxygens (Hotchkiss, 1944; Harold and Baarada, 1967; Jordan, 1987; Roux and Karplus, 1991b; Roux and Karplus, 1991a).

This chapter will discuss the complete refinement of gA by applying the refinement method discussed in Chapter 4 to the four base initial structures described in Chapter 3. These initial structures are representative of the entire conformational space analytically determined for this peptide. The structures were not formed by model fitting the data, nor were any subjective judgments made while determining the structures. The assumptions that were used to aid in the determination of the initial structures, such as a static covalent geometry and a planar peptide linkage, are relaxed during the refinement procedure. A set of ten refinements for each initial structure will be generated and the results of each group discussed, such as carbonyl orientations and atomic root mean square deviations (rmsd), to gain insight into the structural variability sampled by the refinement. The refined structures from the different initial structures will be compared, showing that the refinement procedure consistently finds a structural solution that is nearly identical and hence nearly independent of the starting structure. The resulting forty refined structures will then be atomically averaged and the resulting structure refined with atom moves only in order to remain within the local conformational space defined by the all move refinement of the different initial structures. Detailed structural characteristics of the final refined structure will then be presented.

5.2 Refinement of the Individual Initial Structures

5.2.1 Structural Ensembles

The four base initial structures are each refined ten times with a different starting random number seed so that ten different structural refinements are performed. The same structural constraints are used in each refinement for each initial structure as described in Chapter 4. The different refinement ensembles for the initial structures are shown in Figure 5.1. The structures within the ensembles were first superimposed onto the first structure for each group so that translations and rotations about the channel axis during refinement would not lead to erroneous results in the analysis. The superposition is done by limiting the degrees of freedom so that rotations about X and Y, the axes perpendicular to the channel axis, are not allowed. Rotations about Z reflect rotations about the channel axis and are therefore allowed. Translations along all axes are

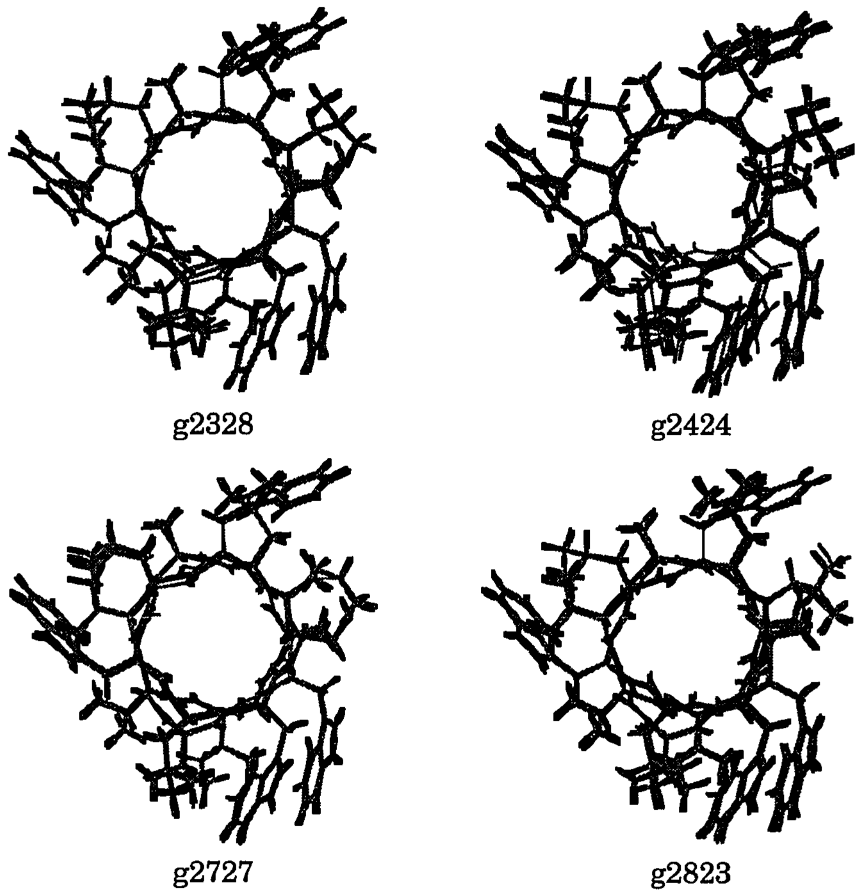


Figure 5.1: Initial Structure Refinements. The four initial structures were each refined ten times to produce forty refined structures. The structures within each group show little variability in the atomic positions.

also allowed. Moving the structures in these ways has no affect on the imposed constraints since the constraints are either relative to the Z axis, as in the SSNMR data, or are only internally relative, such as the intramolecular hydrogen bond distances and the energy. This method of atomic superposition will be followed throughout this chapter.

Each ensemble is in itself consistent with regard to the atomic positions. A few minor deviations are seen, such as the end region of the peptide in the g2424 ensemble that has undergone a plane flip after the Trp_{13} residue. While this has caused the chain to deviate from the rest of the ensemble, the final penalty for this structure is 413, which is comparable to the average penalty for this ensemble of 415. Other plane flips are seen in other structures, but do not affect the structure as the one just described.

Although the conformational space defined by the peptide plane orientations with respect to the channel axis are searched, only small changes in the atom positions are made. The peptide plane orientations refine to a single structure in most cases, but do so with peptide plane moves which have little affect on the global structure. As a result, the deviations within an ensemble are minimal, and the atomic rmsd's are very small: 0.15 Å, 0.15 Å, 0.11 Å and 0.17 Å for the g2328, g2424, g2727 and g2823 ensembles, respectively. The structures in the refinement ensembles all meet the imposed constraints, indicating that this stage of the refinement procedure is successful.

5.2.2 Backbone Carbonyl Orientations

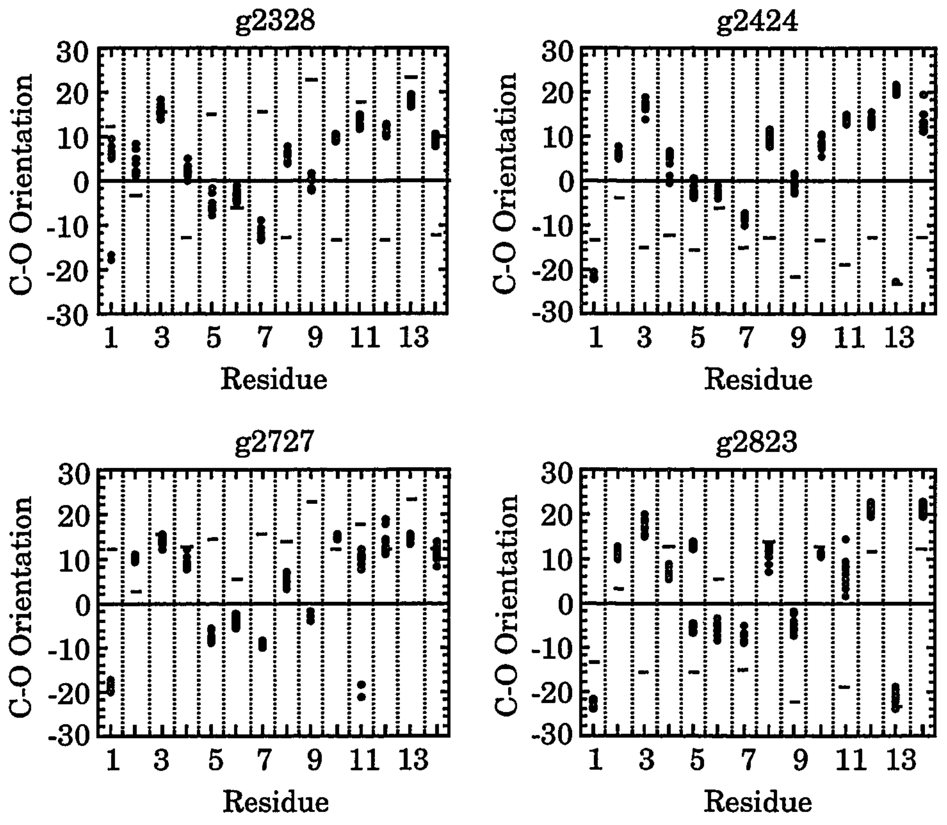


Figure 5.2: Carbonyl Orientations for Initial Structure Refinements. The carbonyl orientations with respect to the channel axis were monitored for each refinement group to determine the ability of the refinement procedure to reach a global refined structure. A positive orientation points into the channel. The flat line depicts the carbonyl orientations in the initial structure.

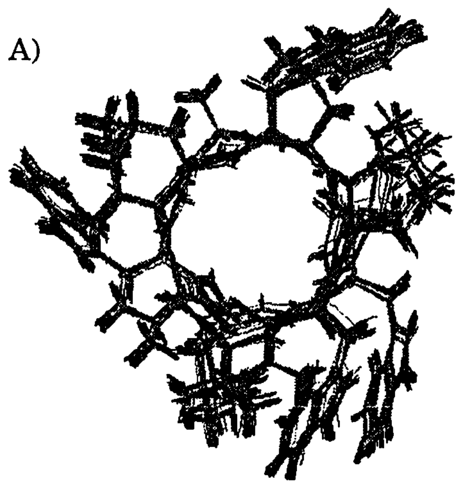
The backbone carbonyl orientations with respect to the channel axis are calculated for each refined structure. Since the conformational space defined by the initial structures is based on the peptide plane orientations, this information is used as one of the means to determine the success of the refinement. The refined structures from each starting configuration should have nearly the same peptide plane orientations if the refinement procedure is adequately searching conformational space and if the constraints are adequate for defining a unique structure. The carbonyl orientations for the individual initial structure refinement ensembles are shown in Figure 5.2. The flat lines in the plots represent the carbonyl orientations for each initial structure before refinement, and the other symbols represent the ten different refinements for each initial structure.

The carbonyl orientations are different for the four initial structures. Refinement, however, produces a set of ensembles that are nearly identical. The terminating residue, Trp_{15}, will not be considered in this analysis since it is affected by the attached ethanolamine, for which no experimental constraints were used. Minor discrepancies exist for a few of the residues where a small number of refined structures within an ensemble differ from the rest, as in the Val_1 site of the g2328 refinement and the Trp_{11} site in the g2727 refinement. The only major difference between the ensembles is the Trp_{13} residue in the g2823 refinement. This carbonyl orientation remains at or near its starting position in all ten of the refined structures. Tunneling of this peptide plane orientation is inhibited for reasons that are not clear.

5.2.3 Merging the Initial Structure Refinement Ensembles

The initial structure refinement ensembles must be joined in order to determine an average atomic structure that represents all of the refinement ensembles. It was shown in Figure 5.1 that the structures within the ensembles are nearly identical and in Figure 5.2 that the carbonyl orientations are very similar across the different ensembles. In order to join the different ensembles into a single refinement ensemble, the individual refined structures must first be superimposed. Since the refinement procedure includes constraints on the dimerization of the monomer (Chapter 4), all of the refined structures are superimposed onto a reference structure that is in a good starting position for dimerization. CHARMM rotates the monomer 180° about the X axis to form the dimer and the IMAGE facility is used to take the dimer into consideration during the calculation of the energy. The starting structure must therefore be translated to a suitable position so that this rotation produces a dimer with the proper intermolecular hydrogen bonds.

A)



B)

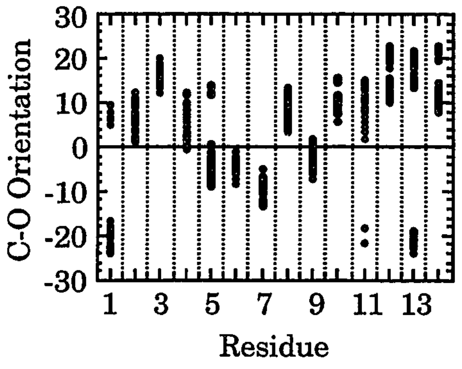


Figure 5.3: All Forty Refinements. The forty refined structures, ten from each of the four base initial structures, are shown superimposed in A). All forty of these refined structures meet the imposed constraints and, therefore, all are used to produce an atomic average that is later refined as the final structure. Although Leu_{10} appears to be disordered, it is well defined within the initial structure refinement ensembles as shown in Figure 5.1. The carbonyl orientations, shown in B), in the structures are generally consistent for individual amino acids.

The superposition produces a single ensemble, shown in Figure 5.3, representing all of the refined initial structures. The carbonyl orientations shown in Figure 5.2 are also shown in Figure 5.3 as a single plot representing the total ensemble. Except for deviations in residues Val_1, Trp_{13} and Trp_{15}, and minor deviations in residues Ala_5 and Trp_{11}, the general pattern of carbonyl orientations is the same across all forty refined structures.

The forty refined structures superimpose well to the reference structure, as shown by the low atomic rmsd of 1.20 Å. This rmsd is determined by first calculating the rmsd for each structure to the reference structure, resulting in forty rmsd values. The individual rmsd values are then averaged. Superimposing the structures onto the first refined structure produces an even lower atomic rmsd of 0.48 Å. This low rmsd value reflects the ability of the refinement procedure to determine a global refined structure, even when starting from different initial structures.

A single sidechain, Leu_{10}, shows variability in the global ensemble. Although it appears in Figure 5.3 as if this sidechain is disordered, it is well ordered within the individual ensembles shown in Figure 5.1. The atomic rmsd within each ensemble of this sidechain alone without further superposition, including backbone atoms, is 0.10 Å for g2328, 0.16 Å for g2424, 0.10 Å for g2727 and 0.22 Å for g2823. The (χ_1, χ_2) torsion angles for a single structure from each of the four initial structure refinements are (-78°, -64°) for g2328, (-69°, -79°) for g2424, (-55°, -56°) for g2727, and (-40°, +98°) for g2823. Taking the atomic average of the forty structures results in (χ_1, χ_2) torsion angles of (-59°, -71°).

The average structure is shown superimposed onto representative structures from each of the individual refinement ensembles in Figure 5.4. The initial refinements have an average penalty of 450. The final penalties were in the range of 411 to 513. Since no single structure has an abnormally high penalty with respect to the other structures, all initial refined structures are used to produce the average structure. Taking an atomic average distorts the bond lengths and bond angles as well as the structural orientations used in calculating the SSNMR values, so refinement of the average structure is required.

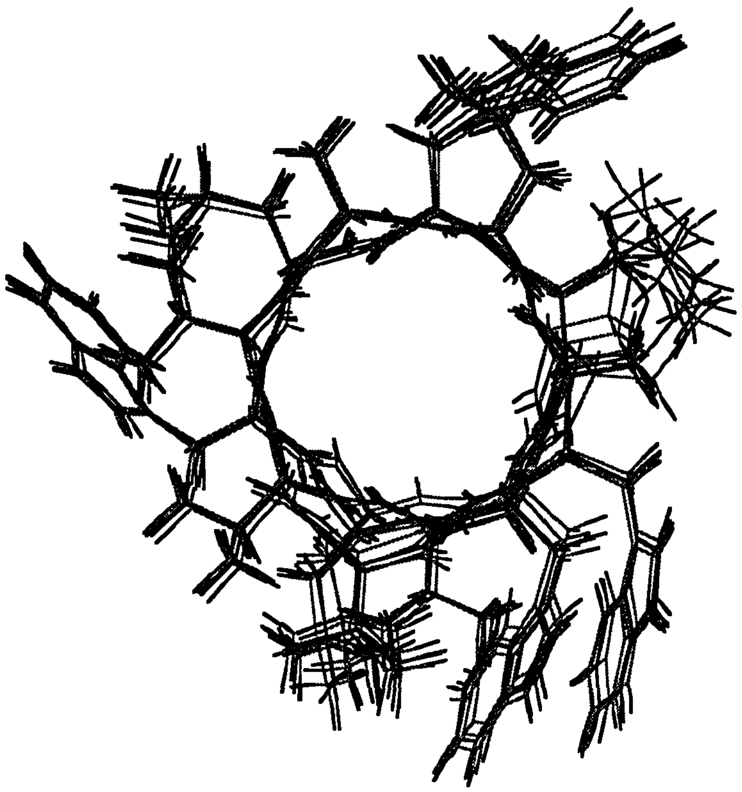


Figure 5.4: Average Structure with Four Initial Refined Structures. The average structure, shown as a solid line, is taken from all forty of the initial refinements. Shown here is a single representative refined structure from each of the four base initial structures along with the atomic average structure to be used for the final refinement.

5.3 Refinement of the Average Structure

The average structure is refined in much the same way as the initial structures. Since taking an average of the initial refinements has produced an average structure that contains only minor distortions in the covalent geometry and the orientations defined by the experimental constraints, only minor refinement is required to produce a final structure. Conformational space defined by the peptide planes has already been searched in the development of the average structure. Therefore, large structural deviations are not required in the final refinement phase. To remain within the local conformational space defined by the average structure, atom moves alone are used for introducing structural modifications. The same refinement parameters will be used in this procedure as those used in the refinement of the initial structures: a starting temperature of 300K and a diffusion parameter of 0.0005 Å. The same structural constraints are applied during this refinement.

The refinement penalties for the average and final structures are shown in Table 5.1. Refinement produces a final structure whose total penalty reflects the structural agreement to the imposed constraints with no single penalty making an unreasonable contribution. The penalty from the SSNMR interactions as a function of residue number is shown in Figure 5.5. The initial structure used for this plot is g2328. The penalty for several of the residues in the initial structure are severe. The final structure, however, has a per residue penalty that is nearly zero for each residue. Reducing the penalty scale to a maximum of 0.5, the value at which the difference between the calculated value and the observed value is equal to the experimental error, shows a variable per residue penalty, but all are within experimental error.

Table 5.1: Refinement Penalties for the Average and Final Structures. The average structure from the refinement ensemble displays high penalties for multiple constraints. The final refinement produces a structure that meets the imposed constraints.

Penalty Type	λ	Average Structure Penalty	Final Structure Penalty
^{15}N Chemical Shift	3.0	16.9	0.9
^{13}C Chemical Shift	3.0	0.1	0.0
^{15}N Indole Chemical Shift	3.0	0.1	0.1
^{15}N-$^{13}C_1$ Dipolar Splitting	3.0	10.3	1.4
^{15}N-^{1}H Dipolar Splitting	3.0	5.7	2.2
^{15}N-^{1}H Indole Dipolar Splitting	3.0	0.4	0.5
Distance	3.0	13.0	7.7
$C_{\alpha,\beta,...}$-2H Quadrupolar Splitting	3.0	313.6	6.0
CHARMM Energy	1.0	5623.2	370.2
Total Penalty		5983.3	389.0

In order to determine that the refinement of the average structure produces a reproducible structure, the refinement of the average structure was repeated. A comparison of the two structures showed an all atom atomic rmsd of only 0.06Å, demonstrating reproducibility of this final refinement procedure.

The superposition of the average and final structures is shown in Figure 5.6. It is evident that only minor structural deviations have occurred. The atomic rmsd between the two structures is only 0.32 Å. The covalent geometry throughout the average structure is distorted from the initial refinements as a result of taking the atomic average. The final structure has been refined into an acceptable covalent geometry defined by the CHARMM force field. Furthermore, the pattern of the carbonyl orientations have not been modified.

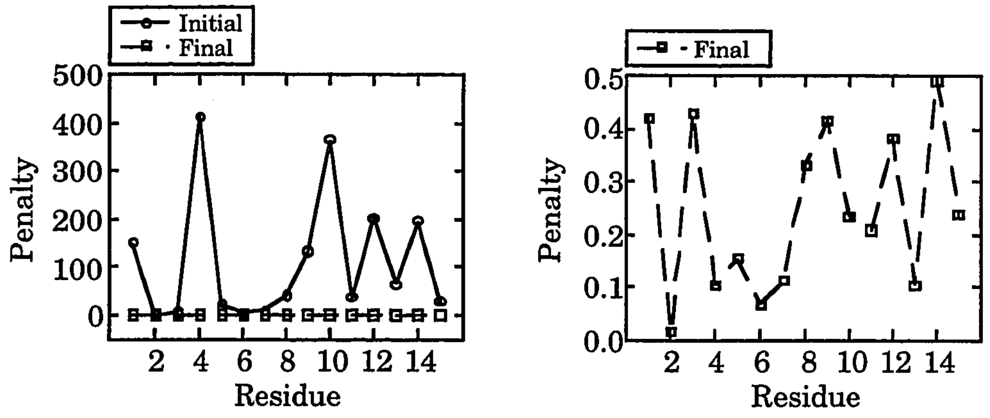


Figure 5.5: Penalty Per Residue. The penalty for all SSNMR interactions is calculated on a per residue basis and plotted for both the initial and final structures. The initial structure is g2328 and the final structure is the final refined structure. The left hand plot shows the penalties for both structures using the same penalty scale. The initial structure has a much higher penalty for many residues. The right hand plot is the final penalty alone plotted on a much smaller penalty scale. The final structure falls within experimental error and has a significantly reduced per residue penalty.

5.4 Characteristics of the Final Structure

The final structure is shown alone in Figure 5.7. This structure represents all of the experimentally derived conformational constraints used in the refinement procedure, the intramolecular hydrogen bonds and the CHARMM energy. The atomic coordinates in CHARMM PDB format for this structure as a dimer are listed in Appendix A.2. The calculated and observed values of the constraints used during the refinement procedure are listed in Appendix A.3.1. The final structure was analyzed using Procheck (Laskowski *et al.*, 1993), the output of which is listed in Appendix

A.3.2. The refinement procedure includes not only the constraints within the monomer, but also the intermolecular hydrogen bonds necessary to form a head to head dimer, thus forming the proper cation channel structure. These constraints are included as part of the CHARMM energy. The dimer is shown in Figure 5.8.

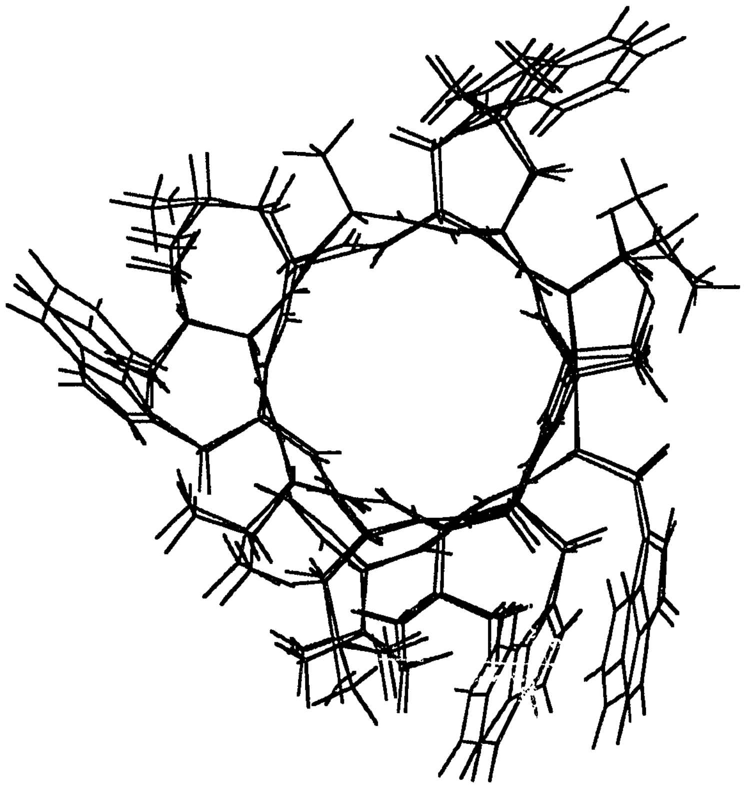


Figure 5.6: Average and Final Refined Structures. Atom moves alone were used during the refinement procedure to produce the final structure from the average of the initial refinements. Minor modifications in the structure were all that were required to relieve undesired penalties caused by taking an average of the forty refined structures. The average structure is shown in dark gray and the final structure in black.

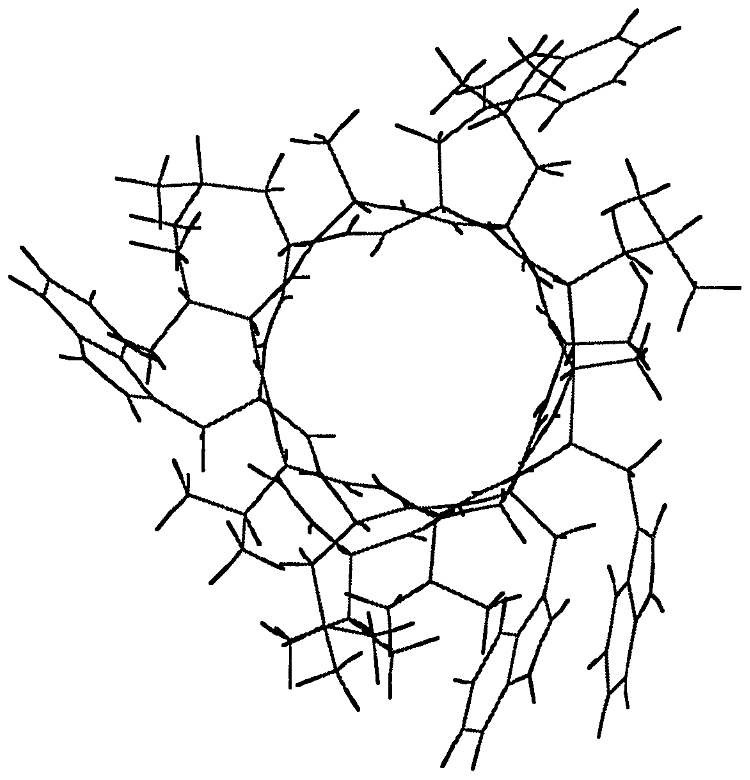


Figure 5.7: Final Structure. The final structure is determined by refining the atomic average from the forty initial refined structures. As a result, it represents the final solution comprising all of the conformational space defined by the four base initial structures. This final structure meets the imposed experimental constraints to within the limits of experimental error, the imposed intramolecular hydrogen bond distances and has a reasonable all atom CHARMM energy.

Table 5.2 shows the g2328 initial structure and global final structure torsion angles. The torsion angles are calculated for the g2328 structure prior to minimization as described in Chapter 4 so as to show the experimentally determined torsion angles. Even though large changes in the final structure (ϕ, ψ) torsion angles have occurred, the two structures show the same pattern of sign alternation, designating the conserved β-helix motif. As mentioned in Chapter 4, refinement relaxes the planarity of the peptide linkage. The final structure shows three deviations larger than 10° and an average deviation of 6°.

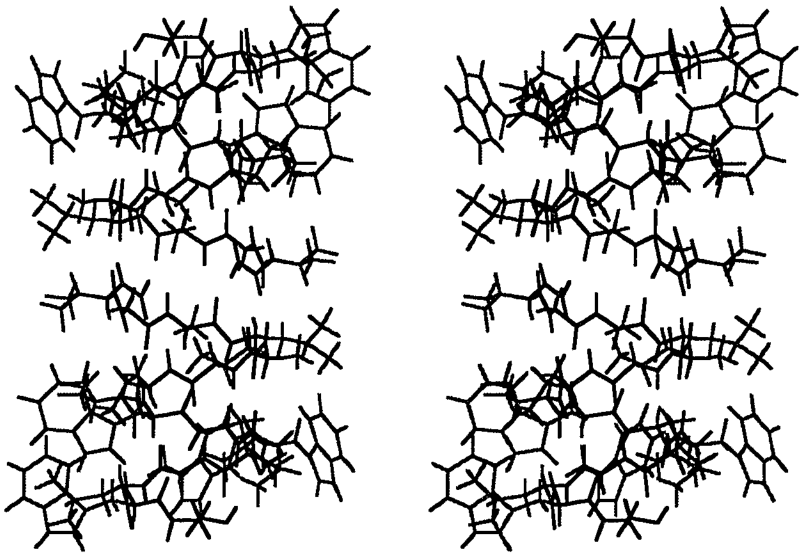


Figure 5.8: Final Structure as a Head to Head Dimer. gA in hydrated lipid bilayers forms a monovalent cation channel composed of two head to head monomers. The backbone is shown in black and the sidechains in dark gray. The refinement procedure refines the intermolecular hydrogen bonds and the position of the dimerization by using an image of the upper monomer rotated 180° about X, an axis perpendicular to the channel axis. The dimer shown here is positioned so that the N-terminus formyl groups are in front. This structure has an average residues per turn of 6.5 and an average pitch of 4.3 Å. The channel length, measured along Z from the Trp_{15}-C_1 in the XY plane from the upper monomer to the Trp_{15}-C_1 in the XY plane from the lower monomer, is 22.8 Å.

Table 5.2: Initial and Final Structure Torsion Angles. The torsion angles for the initial g2328 structure are before minimization to represent the initial structure as determined from experimental constraints. The final structure torsion angles indicate that the structural motif is conserved.

Initial g2328 Structure

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3
Val_1	-138.59	141.38	-179.85	-178.02	—	—
Gly_2	126.62	-116.65	179.63	—	—	—
Ala_3	-129.71	152.79	-179.98	—	—	—
Leu_4	120.10	-102.84	-179.94	-153.97	160.15	—
Ala_5	-141.98	150.95	179.57	—	—	—
Val_6	122.76	-116.39	-179.98	55.00	—	—
Val_7	-132.50	150.21	179.89	-145.03	—	—
Val_8	119.07	-103.65	-179.66	55.06	—	—
Trp_9	-134.54	152.43	179.82	-72.00	-96.89	-179.98
Leu_{10}	107.54	-98.28	179.90	-58.04	-11.97	—
Trp_{11}	-133.53	146.19	179.89	-69.97	-80.90	-179.89
Leu_{12}	114.09	-100.72	-179.76	-173.04	-61.01	—
Trp_{13}	-133.60	153.00	179.88	-62.98	-89.93	-179.88
Leu_{14}	109.93	-100.95	-179.97	-172.96	-62.97	—
Trp_{15}	-136.10	154.60	179.80	-58.01	-96.00	179.51

Final Structure

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3
Val_1	-107.66	120.58	171.28	177.44	—	—
Gly_2	151.13	-129.35	176.09	—	—	—
Ala_3	-114.83	143.93	-174.09	—	—	—
Leu_4	121.58	-135.93	-173.31	-157.28	151.92	—
Ala_5	-116.08	124.83	-178.79	—	—	—
Val_6	146.70	-118.17	176.16	58.90	—	—
Val_7	-120.19	125.72	164.60	-151.41	—	—
Val_8	151.68	-120.34	177.77	60.42	—	—
Trp_9	-110.55	128.31	-173.03	-74.17	-81.81	171.03
Leu_{10}	129.24	-128.14	176.29	-72.61	-70.72	—
Trp_{11}	-108.76	151.52	168.20	-70.63	-90.81	-173.86
Leu_{12}	114.43	-119.08	174.60	-176.82	-59.47	—
Trp_{13}	-99.98	153.38	167.11	-63.75	-85.09	175.97
Leu_{14}	111.18	-115.44	178.48	-176.07	-71.77	—
Trp_{15}	-108.46	127.38	-177.86	-60.64	-90.06	175.96

The sidechain torsion angles are worth noting. The Leu_{10}, Leu_{12} and Leu_{14} sidechains in the initial structures are minimized before refinement as mentioned in Chapter 4. The final structure therefore reflects the refinement of these minimized sidechain orientations. The refinement has little effect on any of the sidechain orientations except for Leu_{10}, and this is primarily exhibited in χ_2. The change in Leu_{10} is therefore due to the minimization. The Leu_{10} (χ_1, χ_2) torsion angles in the minimized initial structures are (-79.35, -51.79) for g2328, (-71.47, -74.66) for g2424, (-66.40, -48.20) for g2727, and (-57.08, 86.25) for g2823, and are (-73°, -71°) in the final structure.

5.4.1 Peptide Plane Orientations

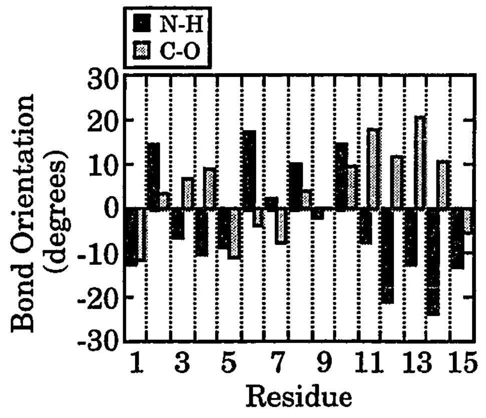


Figure 5.9: Final Structure Bond Orientations. The N-H and C-O bond orientations for the final structure show that the carbonyl orientations are nearly the same as those found during the initial refinements. Positive orientations designate carbonyls pointing into the channel. The latter section of the helix has the carbonyls pointing into the channel axis, thus in a good position for cation solvation. For many sites, the C-O orientation for residue i is not opposite the N-H orientation of residue i+1, designating a non-planar peptide linkage.

The peptide plane orientations in the final structure are important for the understanding of the function of this peptide as a cation channel. As previously discussed, the backbone carbonyl oxygens are thought to be involved in solvation of the cation, thus facilitating its passage through the channel. It is interesting to note that the carbonyl oxygens in the final structure are directed primarily into the channel, as shown in Figure 5.9. Of the four that point away from the channel, the Val_1, Ala_5, Val_6 and Val_7 carbonyls (the terminating Trp_{15} is not considered, as previously discussed), the largest is Val_1 at 11.5°. Of the residues that point into the channel, Trp_{13} is the largest at 20.6°.

Both the N-H and the C-O orientations are worth consideration. While the C-O groups are involved in solvating the cation, the N-H groups are directly controlled by an experimental constraint. The force of the constraint for the N-H group on residue i is transferred through the peptide linkage to the C-O group on residue i-1. There are no direct experimental observations controlling the orientation of the carbonyls, except for the $^{13}C_1$ chemical shifts for Gly_2 and Leu_{10}. The N-C_1 bond has an imposed experimental constraint, however, and is linked through a single bond angle to the C-O bond, thus affecting its orientation. The existing experimental constraints are sufficient to control the orientation of the carbonyl groups, even to the degree of introducing significant non-planarity to the peptide linkages.

5.4.2 Helical Parameters

Helical parameters are readily determined by observing the placement of the C_α atoms within the coordinate system. Plots of the C_α atoms as a function of X, Y and Z are shown in Figure 5.10.

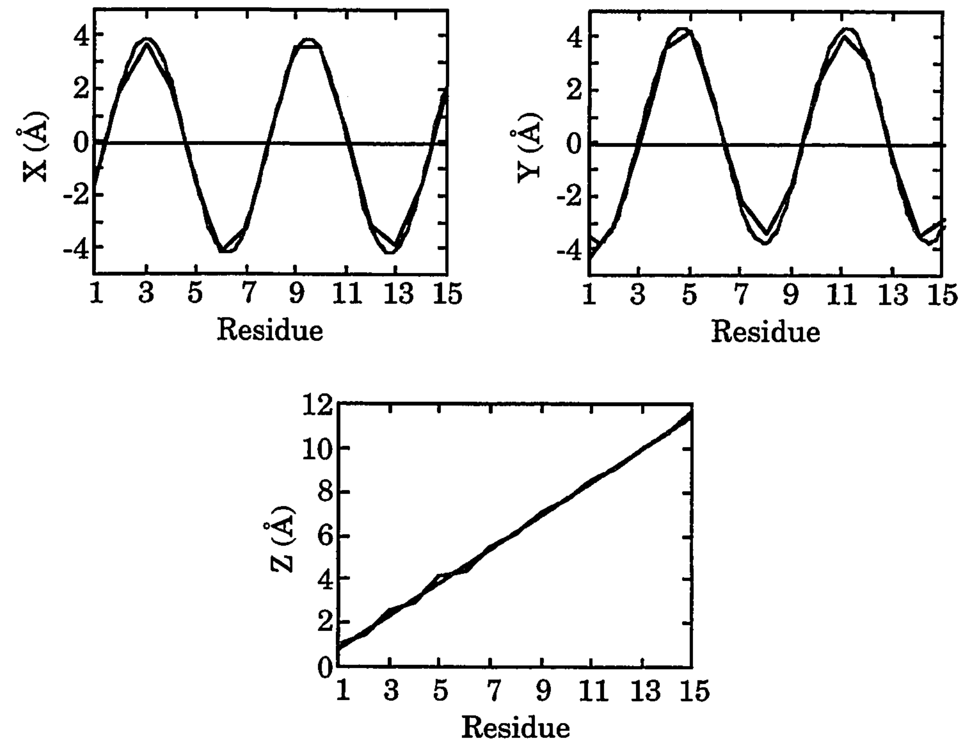


Figure 5.10: Helical Parameters as a Function of C_α Location. The C_α coordinates in X, Y and Z plotted against the C_α residue numbers yields information concerning the helical parameters, such as residues per turn and helical pitch. Plots in X and Y both give 6.5 as the residues per turn when fit to a sine wave. Using 6.5 as the residues per turn, the plot in Z gives the helical pitch as 4.9 Å.

5.4.2.1 Residues per turn. The helix axis is parallel to Z. Since gA forms a regular beta helix, the C_α locations as a function of both X and Y have a regular oscillating pattern. These plots can be fit to a sine wave and used to determine the residues per turn (rpt) of the helix. The form of the equation used to fit the plots in X and Y is

$$y = m_1 + m_2 \sin(m_3 x + m_4), \tag{5-1}$$

where m_x are the variable parameters used to fit the plotted curve. For the plot of C_α vs. X, these parameters are m_1=-0.16, m_2=4.00, m_3=-55.52 and m_4=313.01. For the plot of C_α vs. Y, these parameters are m_1=0.26, m_2=-4.03, m_3=-55.29 and m_4=221.39. m_1 is used to shift the line along the vertical axis. Since the helix is centered on the Z axis, small m_1 values are all that are required. m_2 scales the amplitude of the lines. m_3 serves to define the number of degrees per residue. m_4 controls the phase shift between the two lines. Since X and Y are 90° phase shifted, this is reflected in m_4 for the X and Y plots ($\Delta m_4 = 91.62$). The fits to the plotted curves are shown in Figure 5.10. The equations can be used directly to solve the rpt by using the absolute value of m_3. A full turn of the helix is 360°. Dividing this number by m_3 from either the X or Y plot gives 6.5 rpt. It was mentioned in Chapter 1 that the rpt for this structure was thought to be 6.3 based on early models of the channel structure. For 15 residues, 6.3 rpt gives 2.38 turns of the helix, while 6.5 rpt gives 2.31. A difference of 0.07 turns (0.5 residues) is minor but significant.

5.4.2.2 Helical pitch. The helical pitch, the distance between the turns of the helix, can also be calculated from the C_α locations by plotting C_α as a function of Z. Fitting a straight line to this plot produces the equation

$$y = -0.64 + 0.76x. \qquad (5\text{-}3)$$

The slope of the line indicates the change in distance in Å as a function of residue number. Taking 6.5 as the number of residues in a single turn of the helix and multiplying by the slope of the line, 0.76, gives 4.9 Å as the helical pitch.

5.5 Discussion

5.5.1 Structural Comparison

The structure of gA has been determined by solution NMR studies in

SDS micelles (Bystrov *et al.*, 1987; Lomize *et al.*, 1992). Given here is a structural comparison of this structure (Arseniev) with the one presented in this dissertation (Ketchem). The two structures are shown superimposed in Figure 5.11. The torsion angles for the Ketchem structure are given previously in Table 5.2 and the torsion angles for the Arseniev structure are given in Table 5.3.

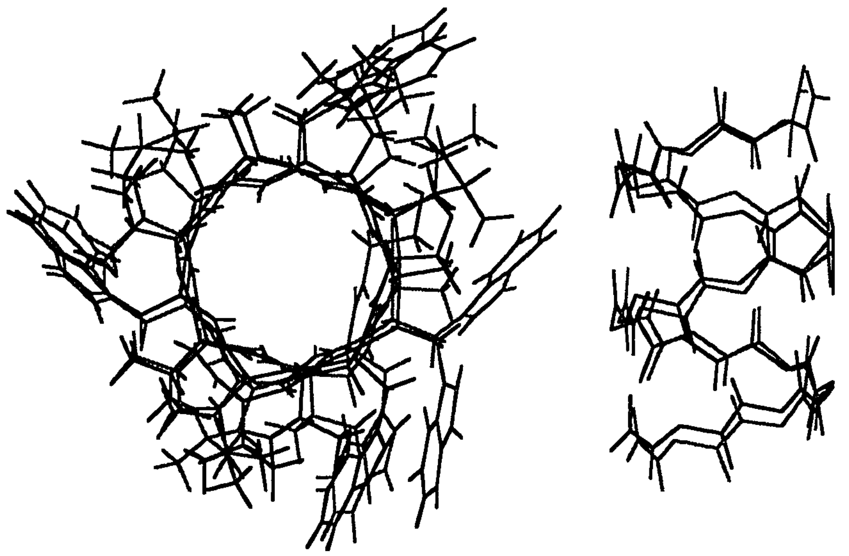


Figure 5.11: Ketchem and Arseniev Structures. The structures were superimposed as previously described. The Ketchem structure is black and the Arseniev structure is dark gray. Both the top and side views show that the backbone structures have similar structural folds. Many of the sidechain orientations differ, though, especially Trp_9. In the Ketchem structure the Trp_9 and Trp_{15} planes stack, while Trp_9 and Trp_{15} in the Arseniev structure point in opposite directions.

Table 5.3: Arseniev Structure Torsion Angles. The (ϕ, ψ) torsion angles alternate in sign, as do those in the Ketchem structure, indicating a similar structural motif. The peptide linkages are constrained to be planar. The sidechain orientations between the two structures are similar except for primarily Trp_9, Leu_{10}, Leu_{12} and Leu_{14}.

Residue	ϕ	ψ	ω	χ_1	χ_2	χ_3
Val_1	-126.92	106.98	-179.85	-175.48	—	—
Gly_2	158.29	-98.92	179.69	—	—	—
Ala_3	-140.35	130.53	-179.97	—	—	—
Leu_4	139.48	-134.05	-179.93	172.16	100.68	—
Ala_5	-107.81	122.86	179.58	—	—	—
Val_6	137.86	-115.73	-179.96	58.67	—	—
Val_7	-110.82	127.81	179.84	-167.39	—	—
Val_8	137.51	-138.84	-179.67	69.42	—	—
Trp_9	-110.34	138.92	179.77	167.26	88.79	-179.95
Leu_{10}	135.76	-97.39	179.92	-175.58	167.38	—
Trp_{11}	-141.44	157.85	179.82	-64.69	-53.21	-179.95
Leu_{12}	110.32	-105.64	-179.71	48.77	73.51	—
Trp_{13}	-132.24	156.35	179.83	-67.50	-90.28	-179.88
Leu_{14}	107.30	-93.89	-179.96	63.08	89.08	—
Trp_{15}	-132.97	136.04	179.70	-68.77	-92.42	179.56

Immediately evident in the Arseniev structure is that the (ϕ, ψ) torsion angles exhibit a β-helix pattern shown in the sign alternation. Also noted is that the peptide plane linkages are fixed near 180°. Plots of the backbone torsion angles in Figure 5.12 show that the backbones of these two experimental structures are qualitatively similar. The all atom atomic rmsd is 2.3 Å. The average deviation in ϕ between the two structures is only 14°, but at the C-terminus of the peptide the deviation in the odd site residues is much larger: 30° for the last three odd residues. The odd site residues show large deviations in ϕ but not ψ, indicating that the difference between the backbone structures occurs in alternating peptide planes. This is verified by the deviations in ψ. The average deviation for the entire structure is only 12°, but the last three even residues have an average ψ deviation of 22°.

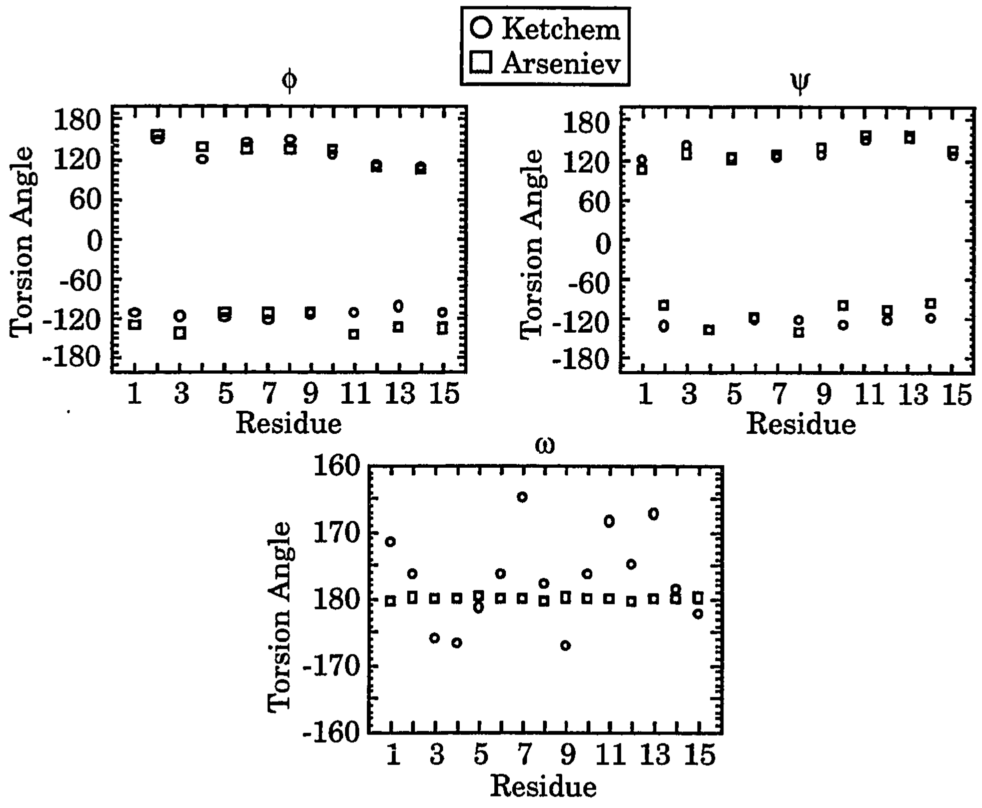


Figure 5.12: Ketchem and Arseniev Backbone Torsion Angles. The backbone torsion angles are qualitatively similar for the two structures, indicating that at this level they represent the same structural motif. A more detailed inspection shows that they differ substantially.

Deviations occurring in (ψ_i, ϕ_{i+1}) pairs indicates changes in peptide plane normal orientations with respect to the channel axis. The deviations in ψ_i are similar to those in ϕ_{i+1} resulting in minimal distortion to the helical parameters. This results in quite different orientations for the carbonyl orientations near the C-terminus. The structure determined in bilayers using SSNMR constraints showed most carbonyl orientations pointing into the channel, as previously shown in Figure 5.9. The Arseniev structure has a pattern of carbonyl orientations that alternates somewhat like g2823. This may be due to the planar peptide linkages used in the Arseniev structure since relaxing the linkage will allow for changes in the carbonyl orientations without severely affecting the rest of the structure. The difference between the alternating carbonyl orientation pattern in the Arseniev structure and the all in carbonyl orientation pattern in the Ketchem structure for the end of the helix accounts for the large differences in the backbone (ϕ, ψ) torsion angles.

The sidechain orientations between the two structures are also similar, as seen in Figure 5.11. The χ_1 values for both Val_1 and Val_7 are comparable, even though these sidechains show large amplitude local motions and only the dominant rotameric state is considered. The major differences are in Trp_9, Leu_{10}, Leu_{12} and Leu_{14}. In the Ketchem structure the Trp_9 and Trp_{15} residues stack. The Arseniev structure changes the Trp_9 orientation by approximately 120° for χ_1 and approximately 180° for χ_2. Although the Trp_9 indole is still available for interacting with the lipid head groups, the energetically favored stacking is lost. The difference in leucine sidechain conformations is probably due to the difference in the bilayer versus micellar surface.

The calculated penalty from the SSNMR observables for the Arseniev structure represent another way to compare these structures. The penalty values are shown in Table 5.4. The majority of the high penalty is from the C-^{2}H sidechain data, while the backbone data shows only modest differences.

Table 5.4: Ketchem and Arseniev SSNMR Penalties. The calculated SSNMR penalties are used to compare these two structures. The Arseniev structure was determined in SDS, which has the effect of distorting the structure in comparison to bilayers. It therefore does not fit well the observed SSNMR data.

Penalty Type	Ketchem	Arseniev
^{15}N Chemical Shift	1.2	41.9
^{13}C Chemical Shift	0.1	1.8
^{15}N Indole Chemical Shift	0.2	17.0
^{15}N-^{13}C$_1$ Dipolar Splitting	1.7	168.4
^{15}N-^{1}H Dipolar Splitting	1.7	24.6
^{15}N-^{1}H Indole Dipolar Splitting	0.6	3.3
$C_{\alpha,\beta,\ldots}$-^{2}H Quadrupolar Splitting	8.9	10155.4
Total Penalty	14.4	10412.4

These observations indicate that the protein fold has been adequately assessed by observations in SDS micelles but details of the sidechains and backbone structures are different in a bilayer environment.

5.5.2 Functional Aspects

The final structure represents a time averaged structure due to the fact that the experimental observations are time averaged. Molecular dynamics occur in natural environments and the structure should be viewed in this light. An understanding of both the structure and dynamics of the system under study is essential to a full understanding of its function. The structure has many important features for function, such as the Tryptophan and backbone carbonyl orientations. It has been shown (Hu *et al.*, 1993) that the tryptophan residues play an important role in both stabilizing the channel through electrostatic interactions at the bilayer surface and providing a net dipole moment for the channel that affects the cation conductance. The backbone carbonyl groups serve to solvate the cation during transport. The final structure supports this in that the carbonyl orientations point predominantly into the channel. Peptide plane librational motions occur on the time scale of cation transport (North and Cross, 1993; North and Cross, 1995), suggesting that the motions of the peptide planes are correlated with cation transport. These results suggest that the backbone carbonyl groups are essential for cation transport.

5.5.3 Final Remarks

A novel method for structure determination in the solid state has been presented. SSNMR has been demonstrated by this work to be a useful method by which protein structures in membrane systems can be solved. The final gA structure has been determined and found to be well within the limits of the imposed SSNMR constraints. The techniques described here are readily applicable to other membrane systems, the primary challenge being the sample preparation for the necessary experimental observations. Once the observations are made, however, the structure can be determined using the method outlined in this dissertation.

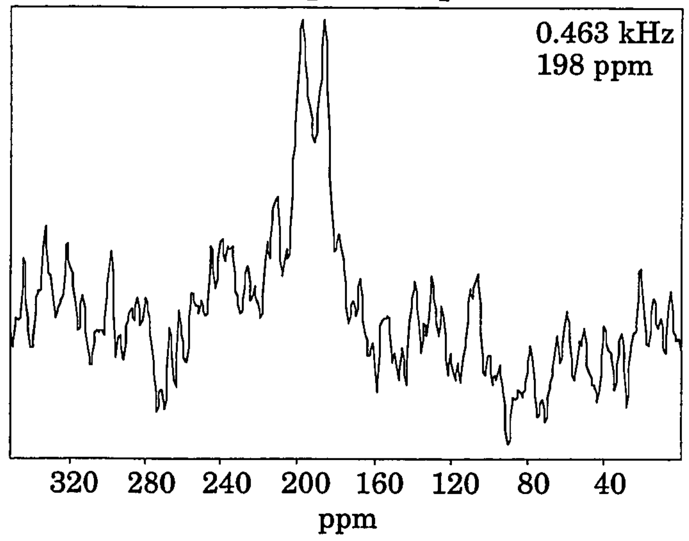
APPENDICES

A.1 Experimental SSNMR Data

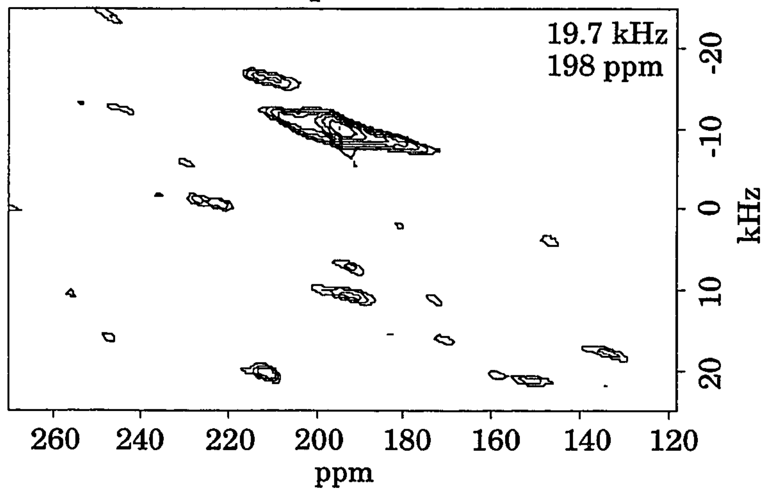
A.1.1 Sample Data Plots

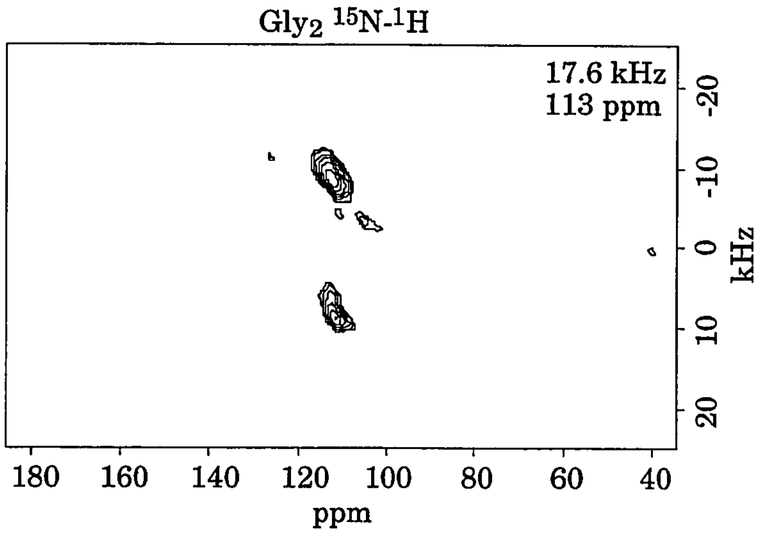
The structural constraints used in the determination and refinement of the final structure were obtained from experimental SSNMR spectra such as those shown here. The ^{15}N-$^{13}C_1$ dipolar splittings were obtained using a one dimensional CPECHO experiment as described in Chapter 2.3.2. The residue designation in the spectra title denotes the residue containing the ^{15}N label. The ^{15}N-^{1}H dipolar splittings were obtained using a two dimensional SLOCF experiment as described in Chapter 2.3.3. All spectra were chemical shift referenced using a saturated $^{15}NH_4NO_3$ solution. The observed dipolar splittings are listed, as well as the observed value of the ^{15}N chemical shift.

Val$_1$ ^{15}N-$^{13}C_1$

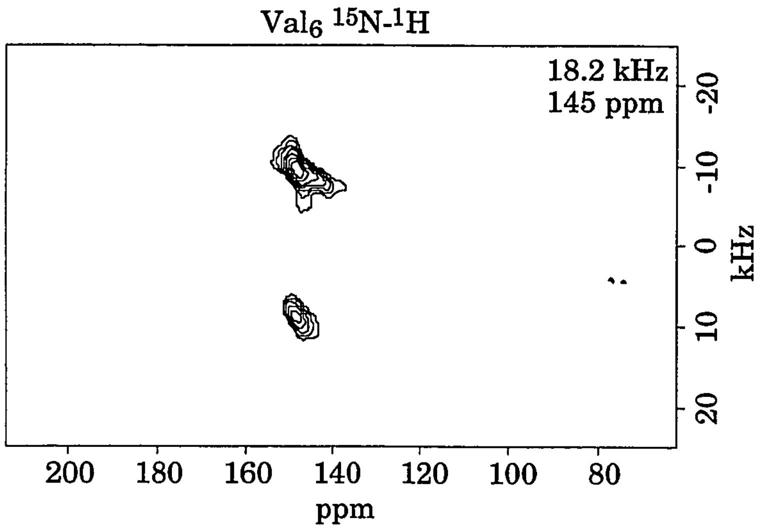


Val$_1$ ^{15}N-^{1}H



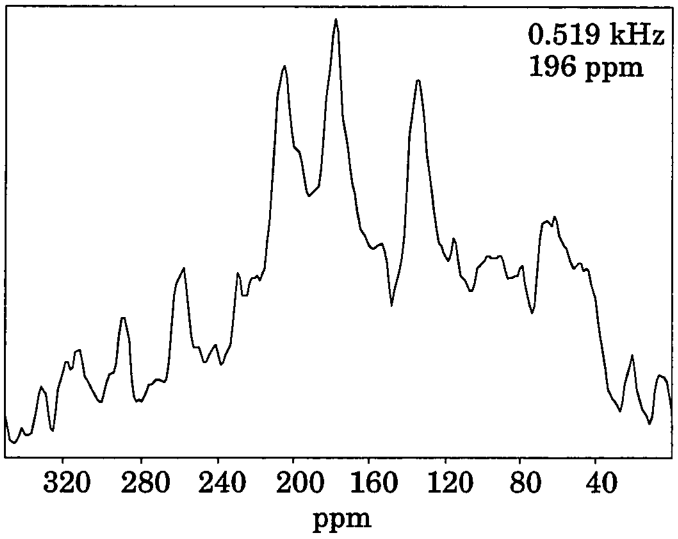


Gly2 15N-1H
17.6 kHz
113 ppm
-20
-10
0
10
20
kHz
180
160
140
120
100
80
60
40
ppm

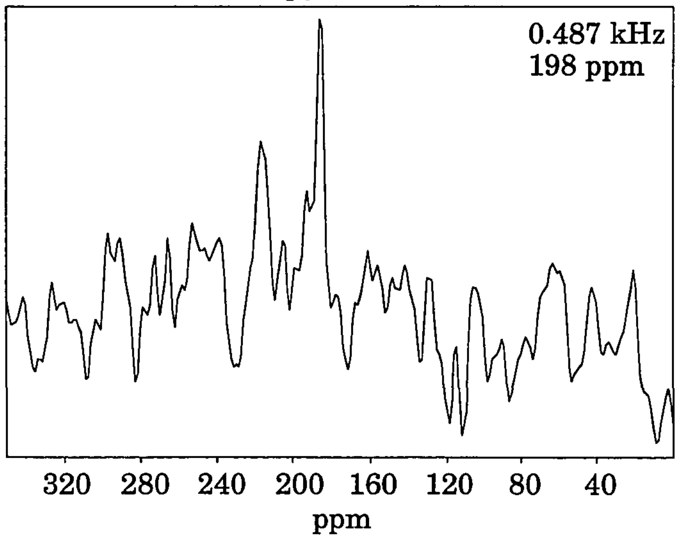


Val6 15N-1H
18.2 kHz
145 ppm
-20
-10
0
10
20
kHz
200
180
160
140
120
100
80
ppm

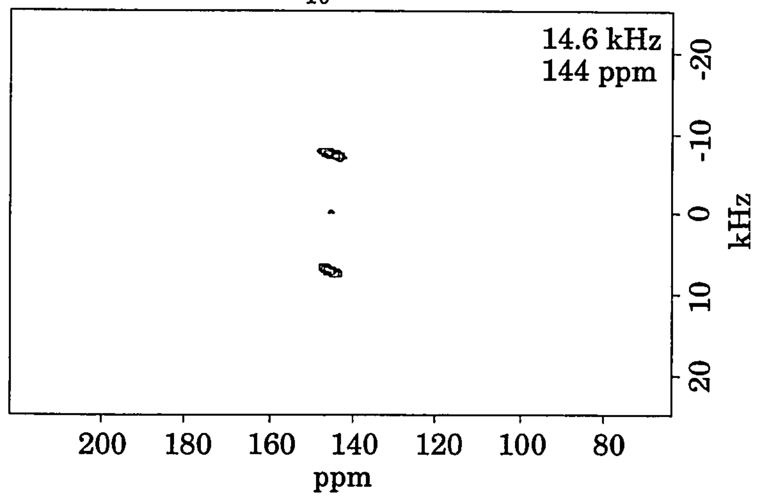
Val7 ^{15}N-$^{13}C_1$



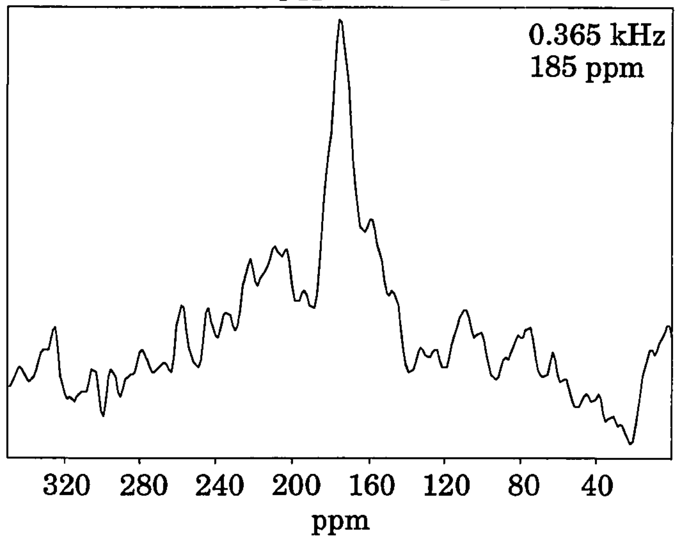
Trp9 ^{15}N-$^{13}C_1$



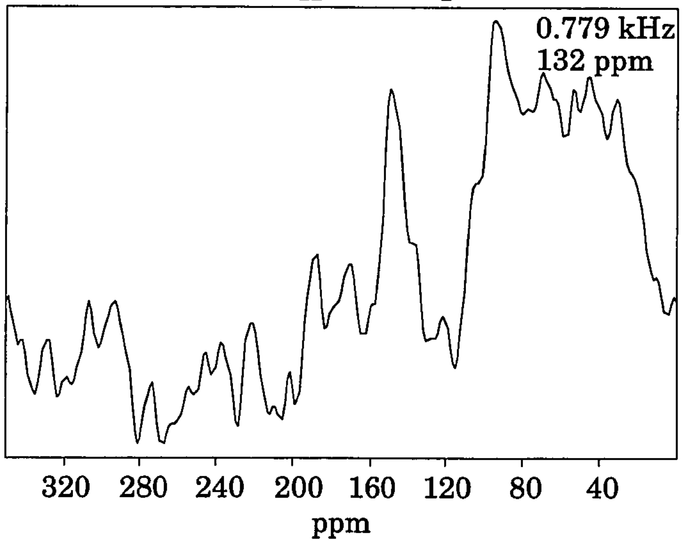
Leu$_{10}$ ^{15}N-^{1}H



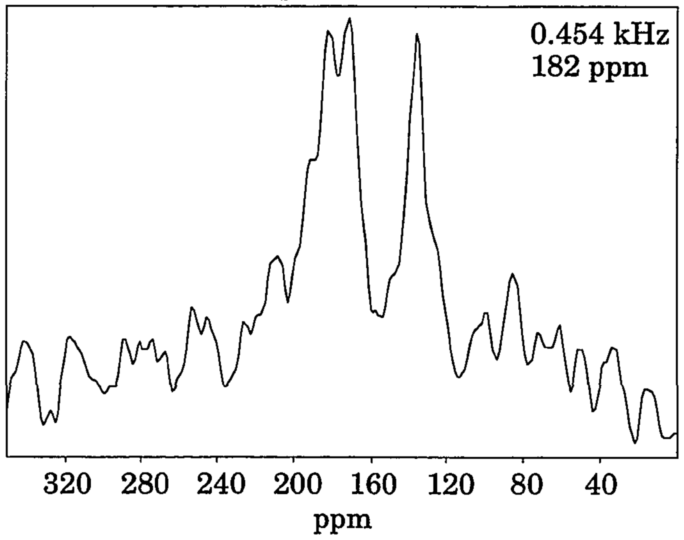
Trp$_{11}$ ^{15}N-^{13}C$_1$



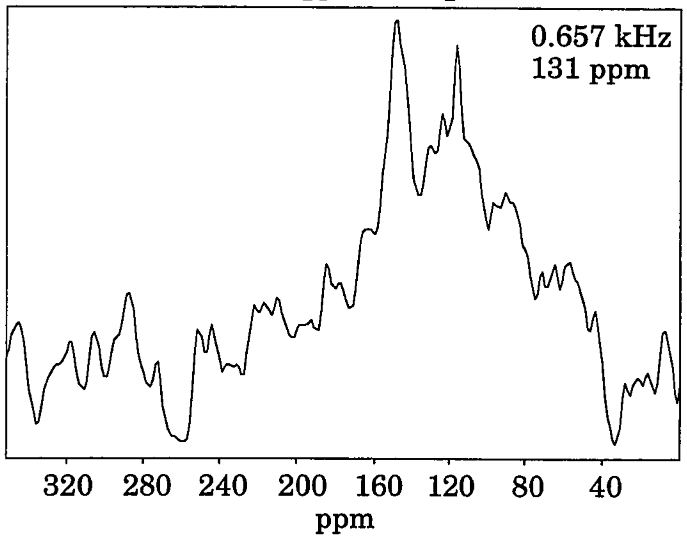
Leu$_{12}$ ^{15}N-^{13}C$_1$



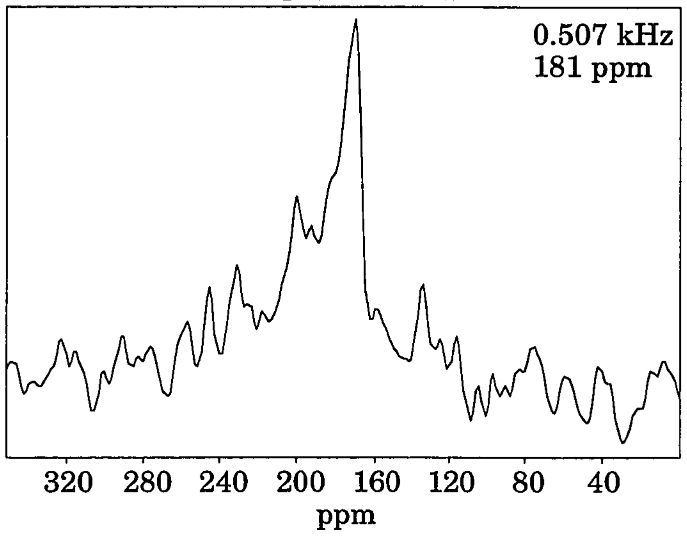
Trp$_{13}$ ^{15}N-^{13}C$_1$



Leu$_{14}$ ^{15}N-^{13}C$_1$



Trp$_{15}$ ^{15}N-^{13}C$_1$



A.1.2 Data Tables

All of the data used in the refinement procedure is listed here, along with the experimental errors used to define the penalty function.

^{15}N Chemical Shift

Residue	σ_{xx} (ppm)	σ_{yy} (ppm)	σ_{zz} (ppm)	α (°)	β (°)	Observed (ppm)	Error (ppm)
Val_1	39.0	63.0	213.0	27.0	106.0	198.0	5.0
Gly_2	28.0	49.0	197.0	0.0	98.0	113.0	5.0
Ala_3	37.0	63.0	206.0	0.0	104.0	198.0	5.0
Leu_4	35.0	64.0	201.0	0.0	105.0	145.0	5.0
Ala_5	38.0	67.0	207.0	0.0	104.0	198.0	5.0
Val_6	31.0	58.0	201.0	0.0	105.0	145.0	5.0
Val_7	37.0	60.0	203.0	0.0	104.0	196.0	5.0
Val_8	28.0	55.0	201.0	0.0	105.0	145.0	5.0
Trp_9	37.0	64.0	204.0	0.0	104.0	198.0	5.0
Leu_{10}	38.0	68.0	204.0	0.0	105.0	144.0	5.0
Trp_{11}	36.0	63.0	194.0	0.0	106.0	185.0	5.0
Leu_{12}	38.0	66.0	196.0	0.0	105.0	132.0	5.0
Trp_{13}	37.0	60.0	195.0	0.0	106.0	182.0	5.0
Leu_{14}	35.0	61.0	195.0	0.0	105.0	131.0	5.0
Trp_{15}	35.0	64.0	198.0	0.0	106.0	181.0	5.0

^{13}C Chemical Shift

Residue	σ_{xx} (ppm)	σ_{yy} (ppm)	σ_{zz} (ppm)	α (°)	β (°)	Observed (ppm)	Error (ppm)
Gly_2	173.0	91.0	243.0	0.0	34.0	175.0	5.0
Leu_{10}	182.0	91.0	247.0	0.0	35.0	180.0	5.0

^{15}N-^{13}C Dipolar Splitting

Residue	$\nu_{\|\|}$ (kHz)	Observed (kHz)	Error (kHz)
Val_1	1.271	0.463	0.1
Gly_2	1.271	0.910	0.1
Ala_3	1.271	0.670	0.1
Leu_4	1.271	0.820	0.1
Ala_5	1.271	0.572	0.1
Val_6	1.271	0.626	0.1
Val_7	1.271	0.519	0.1
Val_8	1.271	0.702	0.1
Trp_9	1.271	0.487	0.1
Trp_{11}	1.271	0.365	0.1
Leu_{12}	1.271	0.779	0.1
Trp_{13}	1.271	0.454	0.1
Leu_{14}	1.271	0.657	0.1
Trp_{15}	1.271	0.507	0.1

^{15}N-^{1}H Dipolar Splitting

Residue	$\nu_{\|\|}$ (kHz)	Observed (kHz)	Error (kHz)
Val_1	11.335	19.7	2.0
Gly_2	11.335	17.6	2.0
Ala_3	11.335	21.8	2.0
Leu_4	11.335	17.2	2.0
Ala_5	11.335	20.7	2.0
Val_6	11.335	18.2	2.0
Val_7	11.335	22.0	2.0
Val_8	11.335	17.8	2.0
Trp_9	11.335	20.7	2.0
Leu_{10}	11.335	14.6	2.0
Trp_{11}	11.335	20.9	2.0
Leu_{12}	11.335	16.2	2.0
Trp_{13}	11.335	21.1	2.0
Leu_{14}	11.335	14.3	2.0
Trp_{15}	11.335	21.5	2.0

Val_1 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	205.0	5.0
C_β-H_β	153.0	133.4	5.0
C_β-$C_{\gamma 1}$	48.0	9.0	5.0
C_β-$C_{\gamma 2}$	48.0	31.3	5.0

Gly_2 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-$H_{\alpha 1}$	160.0	111.6	5.0
C_α-$H_{\alpha 2}$	160.0	192.2	5.0

Ala_3 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	195.4	5.0
C_α-C_β	51.0	32.8	5.0

Leu_4 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	191.0	5.0
C_α-$H_{\beta 1}$	153.0	121.0	5.0
C_α-$H_{\beta 2}$	153.0	51.2	5.0
C_γ-H_γ	138.0	8.3	5.0
C_γ-$H_{\delta 1}$	130.0	29.0	5.0
C_γ-$H_{\delta 1}$	130.0	8.3	5.0

Ala_5 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	188.9	5.0
C_α-C_β	51.0	35.7	5.0

Val_6 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	187.0	5.0
C_β-H_β	153.0	187.0	5.0
C_β-$C_{\gamma 1}$	48.0	3.3	5.0
C_β-$C_{\gamma 2}$	48.0	26.1	5.0

Val_7 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	200.0	5.0
C_β-H_β	153.0	70.9	5.0
C_β-$C_{\gamma 1}$	48.0	41.8	5.0
C_β-$C_{\gamma 2}$	48.0	27.0	5.0

Val_8 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	174.0	5.0
C_β-H_β	153.0	174.0	5.0
C_β-$C_{\gamma 1}$	48.0	8.1	5.0
C_β-$C_{\gamma 2}$	48.0	27.3	5.0

Trp_9 C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
$C_{\delta 1}$-$H_{\delta 1}$	160.0	46.0	5.0
C_α-$H_{\beta 1}$	180.0	87.0	5.0
C_α-$H_{\beta 2}$	158.0	155.0	5.0
C_γ-$H_{\delta 1}$	171.0	102.0	5.0
C_γ-$H_{\delta 1}$	180.0	85.0	5.0

Leu$_{10}$ C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	196.0	5.0
C_α-$H_{\beta1}$	153.0	37.5	5.0
C_α-$H_{\beta2}$	153.0	110.5	5.0
C_γ-H_γ	153.0	49.1	5.0
C_γ-$H_{\delta1}$	137.0	11.5	5.0
C_γ-$H_{\delta1}$	137.0	31.4	5.0

Trp$_{11}$ C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
$C_{\delta1}$-$H_{\delta1}$	146.0	77.0	5.0
C_α-$H_{\beta1}$	178.0	43.0	5.0
C_α-$H_{\beta2}$	144.0	192.0	5.0
C_γ-$H_{\delta1}$	163.0	99.0	5.0
C_γ-$H_{\delta1}$	178.0	39.0	5.0

Leu$_{12}$ C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	207.0	5.0
C_α-$H_{\beta1}$	153.0	170.6	5.0
C_α-$H_{\beta2}$	153.0	48.5	5.0
C_γ-H_γ	115.0	76.3	5.0
C_γ-$H_{\delta1}$	108.0	43.2	5.0
C_γ-$H_{\delta1}$	108.0	6.4	5.0

Trp$_{13}$ C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
$C_{\delta1}$-$H_{\delta1}$	154.0	108.0	5.0
C_α-$H_{\beta1}$	179.0	32.0	5.0
C_α-$H_{\beta2}$	152.0	204.0	5.0
C_γ-$H_{\delta1}$	167.0	81.0	5.0
C_γ-$H_{\delta1}$	179.0	28.0	5.0

Leu_{14} C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
C_α-H_α	160.0	199.0	5.0
C_α-$H_{\beta1}$	153.0	163.2	5.0
C_α-$H_{\beta2}$	153.0	60.4	5.0
C_γ-H_γ	115.0	79.1	5.0
C_γ-$H_{\delta1}$	101.0	38.4	5.0
C_γ-$H_{\delta1}$	101.0	5.5	5.0

Trp_{15} C-^{2}H Quadrupolar Splitting

Bond	QCC (kHz)	Observed (kHz)	Error (kHz)
$C_{\delta1}$-$H_{\delta1}$	165.0	123.0	5.0
C_α-$H_{\beta1}$	181.0	4.0	5.0
C_α-$H_{\beta2}$	164.0	198.0	5.0
C_γ-$H_{\delta1}$	173.0	59.0	5.0
C_γ-$H_{\delta1}$	181.0	1.0	5.0

Trp Indole ^{15}N Chemical Shift

Residue	σ_{xx} (ppm)	σ_{yy} (ppm)	σ_{zz} (ppm)	α (°)	β (°)	Observed (ppm)	Error (ppm)
Trp_9	43.0	114.0	167.0	90.0	25.0	145.0	5.0
Trp_{11}	43.0	104.0	150.0	90.0	25.0	144.0	5.0
Trp_{13}	45.0	117.0	165.0	90.0	25.0	144.0	5.0
Trp_{15}	43.0	120.0	167.0	90.0	25.0	139.0	5.0

^{15}N-^{1}H Dipolar Splitting

Residue	$\nu_{\parallel}$ (kHz)	Observed (kHz)	Error (kHz)
Trp_9	10.7	13.2	2.0
Trp_{11}	10.3	11.1	2.0
Trp_{13}	10.6	10.1	2.0
Trp_{15}	10.9	7.7	2.0

A.2 Atomic Coordinates of the Final Gramicidin Structure

The final structural solution is represented here as atomic coordinates in CHARMM PDB format in units of angstroms. Both upper and lower monomers are shown. The residue numbering scheme starts with the N-terminus formyl blocking group as 1 and ends with the C-terminus blocking group as 17. Normally, the first Valine residue is termed 1, so be aware of this numbering change.

```
REMARK   1 Structure refined with:
REMARK   1 TORC (TOtal Refinement of Constraints) v5.4
REMARK   1 Wed Sep 13 00:05:31 1995
REMARK   2 Randal R. Ketchem
REMARK   2 Institute of Molecular Biophysics   904.644.1309 (voice)
REMARK   2 Florida State University            904.644.1366 (FAX)
REMARK   2 Tallahassee, FL 32306-3015          rrk@magnet.fsu.edu (email)
ATOM      1  HA  CHO   1      -4.124  -2.176   0.153  0.00  0.00      MONO
ATOM      2  C   CHO   1      -3.382  -2.570   0.855  0.00  0.00      MONO
ATOM      3  O   CHO   1      -3.395  -2.314   2.055  0.00  0.00      MONO
ATOM      4  N   VAL   2      -2.604  -3.494   0.244  0.00  0.00      MONO
ATOM      5  HN  VAL   2      -2.679  -3.730  -0.726  0.00  0.00      MONO
ATOM      6  CA  VAL   2      -1.668  -4.304   1.036  0.00  0.00      MONO
ATOM      7  HA  VAL   2      -1.953  -4.130   2.066  0.00  0.00      MONO
ATOM      8  CB  VAL   2      -1.699  -5.796   0.721  0.00  0.00      MONO
ATOM      9  HB  VAL   2      -1.201  -6.107  -0.231  0.00  0.00      MONO
ATOM     10  CG1 VAL   2      -0.640  -6.609   1.544  0.00  0.00      MONO
ATOM     11 HG11 VAL   2      -0.736  -7.693   1.307  0.00  0.00      MONO
ATOM     12 HG12 VAL   2       0.400  -6.303   1.302  0.00  0.00      MONO
ATOM     13 HG13 VAL   2      -0.797  -6.464   2.625  0.00  0.00      MONO
ATOM     14  CG2 VAL   2      -3.105  -6.352   0.970  0.00  0.00      MONO
ATOM     15 HG21 VAL   2      -3.097  -7.458   0.891  0.00  0.00      MONO
ATOM     16 HG22 VAL   2      -3.448  -6.063   1.990  0.00  0.00      MONO
ATOM     17 HG23 VAL   2      -3.811  -5.941   0.219  0.00  0.00      MONO
ATOM     18  C   VAL   2      -0.282  -3.838   0.751  0.00  0.00      MONO
ATOM     19  O   VAL   2       0.139  -3.929  -0.401  0.00  0.00      MONO
ATOM     20  N   GLY   3       0.473  -3.378   1.770  0.00  0.00      MONO
ATOM     21  HN  GLY   3       0.104  -3.220   2.697  0.00  0.00      MONO
ATOM     22  CA  GLY   3       1.888  -3.113   1.494  0.00  0.00      MONO
ATOM     23  HA1 GLY   3       2.463  -4.012   1.660  0.00  0.00      MONO
ATOM     24  HA2 GLY   3       1.951  -2.735   0.513  0.00  0.00      MONO
ATOM     25  C   GLY   3       2.467  -2.042   2.344  0.00  0.00      MONO
ATOM     26  O   GLY   3       2.334  -2.130   3.557  0.00  0.00      MONO
ATOM     27  N   ALA   4       3.158  -1.008   1.786  0.00  0.00      MONO
ATOM     28  HN  ALA   4       3.297  -0.983   0.790  0.00  0.00      MONO
ATOM     29  CA  ALA   4       3.672   0.112   2.587  0.00  0.00      MONO
ATOM     30  HA  ALA   4       3.362  -0.098   3.583  0.00  0.00      MONO
ATOM     31  CB  ALA   4       5.186   0.355   2.416  0.00  0.00      MONO
ATOM     32  HB1 ALA   4       5.501   1.192   3.080  0.00  0.00      MONO
ATOM     33  HB2 ALA   4       5.752  -0.548   2.723  0.00  0.00      MONO
ATOM     34  HB3 ALA   4       5.436   0.619   1.374  0.00  0.00      MONO
ATOM     35  C   ALA   4       3.011   1.413   2.234  0.00  0.00      MONO
ATOM     36  O   ALA   4       2.761   1.614   1.057  0.00  0.00      MONO
ATOM     37  N   LEU   5       2.775   2.301   3.243  0.00  0.00      MONO
ATOM    38  HN  LEU     5       3.101   2.062   4.171  0.00  0.00      MONO
ATOM    39  CA  LEU     5       2.001   3.549   2.960  0.00  0.00      MONO
ATOM    40  HA  LEU     5       1.611   3.493   1.967  0.00  0.00      MONO
ATOM    41  CB  LEU     5       2.752   4.940   3.075  0.00  0.00      MONO
ATOM    42  HB1 LEU     5       3.366   4.810   4.001  0.00  0.00      MONO
ATOM    43  HB2 LEU     5       3.533   5.080   2.308  0.00  0.00      MONO
ATOM    44  CG  LEU     5       1.932   6.271   3.322  0.00  0.00      MONO
ATOM    45  HG  LEU     5       1.109   5.927   3.995  0.00  0.00      MONO
ATOM    46  CD1 LEU     5       2.719   7.345   4.056  0.00  0.00      MONO
ATOM    47 HD11 LEU     5       2.039   8.047   4.593  0.00  0.00      MONO
ATOM    48 HD12 LEU     5       3.391   6.860   4.773  0.00  0.00      MONO
ATOM    49 HD13 LEU     5       3.342   7.946   3.353  0.00  0.00      MONO
ATOM    50  CD2 LEU     5       0.932   6.992   2.384  0.00  0.00      MONO
ATOM    51 HD21 LEU     5       0.373   7.778   2.931  0.00  0.00      MONO
ATOM    52 HD22 LEU     5       1.478   7.476   1.547  0.00  0.00      MONO
ATOM    53 HD23 LEU     5       0.209   6.267   1.953  0.00  0.00      MONO
ATOM    54  C   LEU     5       0.786   3.584   3.849  0.00  0.00      MONO
ATOM    55  O   LEU     5       0.913   3.371   5.051  0.00  0.00      MONO
ATOM    56  N   ALA     6      -0.392   3.951   3.276  0.00  0.00      MONO
ATOM    57  HN  ALA     6      -0.477   4.126   2.294  0.00  0.00      MONO
ATOM    58  CA  ALA     6      -1.578   4.147   4.115  0.00  0.00      MONO
ATOM    59  HA  ALA     6      -1.266   3.896   5.105  0.00  0.00      MONO
ATOM    60  CB  ALA     6      -2.223   5.549   3.988  0.00  0.00      MONO
ATOM    61  HB1 ALA     6      -3.141   5.614   4.619  0.00  0.00      MONO
ATOM    62  HB2 ALA     6      -1.508   6.328   4.316  0.00  0.00      MONO
ATOM    63  HB3 ALA     6      -2.518   5.758   2.938  0.00  0.00      MONO
ATOM    64  C   ALA     6      -2.612   3.136   3.740  0.00  0.00      MONO
ATOM    65  O   ALA     6      -2.991   3.052   2.578  0.00  0.00      MONO
ATOM    66  N   VAL     7      -3.119   2.332   4.695  0.00  0.00      MONO
ATOM    67  HN  VAL     7      -2.737   2.326   5.627  0.00  0.00      MONO
ATOM    68  CA  VAL     7      -4.117   1.312   4.348  0.00  0.00      MONO
ATOM    69  HA  VAL     7      -3.926   0.944   3.350  0.00  0.00      MONO
ATOM    70  CB  VAL     7      -5.606   1.727   4.435  0.00  0.00      MONO
ATOM    71  HB  VAL     7      -5.859   2.024   5.481  0.00  0.00      MONO
ATOM    72  CG1 VAL     7      -5.912   2.922   3.513  0.00  0.00      MONO
ATOM    73 HG11 VAL     7      -6.991   3.164   3.545  0.00  0.00      MONO
ATOM    74 HG12 VAL     7      -5.352   3.824   3.830  0.00  0.00      MONO
ATOM    75 HG13 VAL     7      -5.632   2.670   2.468  0.00  0.00      MONO
ATOM    76  CG2 VAL     7      -6.519   0.565   3.988  0.00  0.00      MONO
ATOM    77 HG21 VAL     7      -7.553   0.931   3.816  0.00  0.00      MONO
ATOM    78 HG22 VAL     7      -6.156   0.122   3.037  0.00  0.00      MONO
ATOM    79 HG23 VAL     7      -6.568  -0.225   4.769  0.00  0.00      MONO
ATOM    80  C   VAL     7      -3.887   0.144   5.260  0.00  0.00      MONO
ATOM    81  O   VAL     7      -4.004   0.289   6.473  0.00  0.00      MONO
ATOM    82  N   VAL     8      -3.551  -1.026   4.675  0.00  0.00      MONO
ATOM    83  HN  VAL     8      -3.462  -1.169   3.684  0.00  0.00      MONO
ATOM    84  CA  VAL     8      -3.234  -2.170   5.525  0.00  0.00      MONO
ATOM    85  HA  VAL     8      -3.224  -1.815   6.538  0.00  0.00      MONO
ATOM    86  CB  VAL     8      -4.216  -3.343   5.368  0.00  0.00      MONO
ATOM    87  HB  VAL     8      -3.866  -4.016   4.548  0.00  0.00      MONO
ATOM    88  CG1 VAL     8      -4.300  -4.151   6.678  0.00  0.00      MONO
ATOM    89 HG11 VAL     8      -5.049  -4.962   6.569  0.00  0.00      MONO
ATOM    90 HG12 VAL     8      -3.323  -4.609   6.933  0.00  0.00      MONO
ATOM    91 HG13 VAL     8      -4.626  -3.497   7.512  0.00  0.00      MONO
ATOM    92  CG2 VAL     8      -5.617  -2.842   4.951  0.00  0.00      MONO
ATOM    93 HG21 VAL     8      -6.353  -3.675   4.960  0.00  0.00      MONO
ATOM    94 HG22 VAL     8      -5.963  -2.065   5.662  0.00  0.00      MONO
ATOM    95 HG23 VAL     8      -5.594  -2.411   3.925  0.00  0.00      MONO
ATOM    96  C   VAL     8      -1.817  -2.596   5.214  0.00  0.00      MONO
ATOM    97  O   VAL     8      -1.443  -2.854   4.065  0.00  0.00      MONO
ATOM    98  N   VAL     9      -0.898  -2.704   6.187  0.00  0.00      MONO
ATOM    99  HN  VAL     9      -1.121  -2.392   7.117  0.00  0.00      MONO
ATOM    100  CA  VAL     9       0.363  -3.400   6.031  0.00  0.00       MONO
ATOM    101  HA  VAL     9       0.778  -3.201   5.052  0.00  0.00       MONO
ATOM    102  CB  VAL     9       0.246  -4.908   6.294  0.00  0.00       MONO
ATOM    103  HB  VAL     9      -0.128  -5.148   7.311  0.00  0.00       MONO
ATOM    104  CG1 VAL     9      -0.747  -5.565   5.310  0.00  0.00       MONO
ATOM    105 HG11 VAL     9      -0.709  -6.673   5.416  0.00  0.00       MONO
ATOM    106 HG12 VAL     9      -1.792  -5.248   5.495  0.00  0.00       MONO
ATOM    107 HG13 VAL     9      -0.465  -5.296   4.272  0.00  0.00       MONO
ATOM    108  CG2 VAL     9       1.582  -5.555   5.966  0.00  0.00       MONO
ATOM    109 HG21 VAL     9       1.489  -6.661   5.976  0.00  0.00       MONO
ATOM    110 HG22 VAL     9       1.944  -5.236   4.964  0.00  0.00       MONO
ATOM    111 HG23 VAL     9       2.344  -5.274   6.729  0.00  0.00       MONO
ATOM    112  C   VAL     9       1.416  -2.801   6.946  0.00  0.00       MONO
ATOM    113  O   VAL     9       1.257  -2.784   8.163  0.00  0.00       MONO
ATOM    114  N   TRP    10       2.518  -2.299   6.349  0.00  0.00       MONO
ATOM    115  HN  TRP    10       2.641  -2.266   5.358  0.00  0.00       MONO
ATOM    116  CA  TRP    10       3.599  -1.648   7.077  0.00  0.00       MONO
ATOM    117  HA  TRP    10       3.406  -1.744   8.133  0.00  0.00       MONO
ATOM    118  CB  TRP    10       5.001  -2.282   6.803  0.00  0.00       MONO
ATOM    119  HB1 TRP    10       5.133  -2.386   5.707  0.00  0.00       MONO
ATOM    120  HB2 TRP    10       5.786  -1.598   7.197  0.00  0.00       MONO
ATOM    121  CG  TRP    10       5.235  -3.632   7.476  0.00  0.00       MONO
ATOM    122  CD2 TRP    10       5.018  -4.922   6.879  0.00  0.00       MONO
ATOM    123  CD1 TRP    10       5.616  -3.869   8.769  0.00  0.00       MONO
ATOM    124  HD1 TRP    10       5.857  -3.113   9.504  0.00  0.00       MONO
ATOM    125  NE1 TRP    10       5.527  -5.217   9.053  0.00  0.00       MONO
ATOM    126  HE1 TRP    10       5.846  -5.612   9.886  0.00  0.00       MONO
ATOM    127  CE2 TRP    10       5.153  -5.883   7.913  0.00  0.00       MONO
ATOM    128  CE3 TRP    10       4.690  -5.297   5.580  0.00  0.00       MONO
ATOM    129  HE3 TRP    10       4.555  -4.584   4.783  0.00  0.00       MONO
ATOM    130  CZ2 TRP    10       4.917  -7.231   7.676  0.00  0.00       MONO
ATOM    131  HZ2 TRP    10       4.979  -7.956   8.470  0.00  0.00       MONO
ATOM    132  CZ3 TRP    10       4.456  -6.660   5.347  0.00  0.00       MONO
ATOM    133  HZ3 TRP    10       4.066  -7.003   4.399  0.00  0.00       MONO
ATOM    134  CH2 TRP    10       4.547  -7.612   6.375  0.00  0.00       MONO
ATOM    135  HH2 TRP    10       4.322  -8.629   6.095  0.00  0.00       MONO
ATOM    136  C   TRP    10       3.641  -0.146   6.781  0.00  0.00       MONO
ATOM    137  O   TRP    10       3.651   0.332   5.640  0.00  0.00       MONO
ATOM    138  N   LEU    11       3.660   0.694   7.829  0.00  0.00       MONO
ATOM    139  HN  LEU    11       3.476   0.259   8.718  0.00  0.00       MONO
ATOM    140  CA  LEU    11       3.546   2.127   7.600  0.00  0.00       MONO
ATOM    141  HA  LEU    11       3.192   2.264   6.592  0.00  0.00       MONO
ATOM    142  CB  LEU    11       4.889   2.919   7.410  0.00  0.00       MONO
ATOM    143  HB1 LEU    11       5.540   2.402   6.668  0.00  0.00       MONO
ATOM    144  HB2 LEU    11       4.583   3.980   7.251  0.00  0.00       MONO
ATOM    145  CG  LEU    11       5.867   3.211   8.595  0.00  0.00       MONO
ATOM    146  HG  LEU    11       6.704   3.781   8.104  0.00  0.00       MONO
ATOM    147  CD1 LEU    11       5.235   4.251   9.534  0.00  0.00       MONO
ATOM    148 HD11 LEU    11       5.953   4.536  10.325  0.00  0.00       MONO
ATOM    149 HD12 LEU    11       4.957   5.164   8.976  0.00  0.00       MONO
ATOM    150 HD13 LEU    11       4.327   3.852  10.019  0.00  0.00       MONO
ATOM    151  CD2 LEU    11       6.586   2.072   9.331  0.00  0.00       MONO
ATOM    152 HD21 LEU    11       6.569   2.219  10.425  0.00  0.00       MONO
ATOM    153 HD22 LEU    11       6.168   1.086   9.057  0.00  0.00       MONO
ATOM    154 HD23 LEU    11       7.655   2.085   9.037  0.00  0.00       MONO
ATOM    155  C   LEU    11       2.475   2.727   8.469  0.00  0.00       MONO
ATOM    156  O   LEU    11       2.399   2.520   9.676  0.00  0.00       MONO
ATOM    157  N   TRP    12       1.565   3.469   7.824  0.00  0.00       MONO
ATOM    158  HN  TRP    12       1.558   3.620   6.833  0.00  0.00       MONO
ATOM    159  CA  TRP    12       0.443   4.035   8.523  0.00  0.00       MONO
ATOM    160  HA  TRP    12       0.556   3.878   9.591  0.00  0.00       MONO
ATOM    161  CB  TRP    12       0.386   5.581   8.344  0.00  0.00       MONO
ATOM    162  HB1 TRP    12        0.442   5.810   7.259  0.00  0.00      MONO
ATOM    163  HB2 TRP    12       -0.584   5.961   8.716  0.00  0.00      MONO
ATOM    164  CG  TRP    12        1.489   6.375   9.080  0.00  0.00      MONO
ATOM    165  CD2 TRP    12        2.760   6.756   8.504  0.00  0.00      MONO
ATOM    166  CD1 TRP    12        1.491   6.916  10.344  0.00  0.00      MONO
ATOM    167  HD1 TRP    12        0.803   6.734  11.159  0.00  0.00      MONO
ATOM    168  NE1 TRP    12        2.641   7.666  10.548  0.00  0.00      MONO
ATOM    169  HE1 TRP    12        2.958   8.096  11.368  0.00  0.00      MONO
ATOM    170  CE2 TRP    12        3.433   7.566   9.440  0.00  0.00      MONO
ATOM    171  CE3 TRP    12        3.346   6.474   7.280  0.00  0.00      MONO
ATOM    172  HE3 TRP    12        2.827   5.856   6.563  0.00  0.00      MONO
ATOM    173  CZ2 TRP    12        4.686   8.110   9.165  0.00  0.00      MONO
ATOM    174  HZ2 TRP    12        5.196   8.747   9.874  0.00  0.00      MONO
ATOM    175  CZ3 TRP    12        4.607   7.039   6.987  0.00  0.00      MONO
ATOM    176  HZ3 TRP    12        4.902   6.986   5.949  0.00  0.00      MONO
ATOM    177  CH2 TRP    12        5.256   7.865   7.905  0.00  0.00      MONO
ATOM    178  HH2 TRP    12        6.191   8.337   7.634  0.00  0.00      MONO
ATOM    179  C   TRP    12       -0.879   3.350   8.163  0.00  0.00      MONO
ATOM    180  O   TRP    12       -1.177   2.845   7.079  0.00  0.00      MONO
ATOM    181  N   LEU    13       -1.776   3.391   9.142  0.00  0.00      MONO
ATOM    182  HN  LEU    13       -1.561   3.784  10.028  0.00  0.00      MONO
ATOM    183  CA  LEU    13       -3.159   3.056   9.029  0.00  0.00      MONO
ATOM    184  HA  LEU    13       -3.414   2.847   7.995  0.00  0.00      MONO
ATOM    185  CB  LEU    13       -3.783   4.359   9.571  0.00  0.00      MONO
ATOM    186  HB1 LEU    13       -3.331   4.452  10.592  0.00  0.00      MONO
ATOM    187  HB2 LEU    13       -3.343   5.258   9.080  0.00  0.00      MONO
ATOM    188  CG  LEU    13       -5.287   4.568   9.758  0.00  0.00      MONO
ATOM    189  HG  LEU    13       -5.393   5.654   9.998  0.00  0.00      MONO
ATOM    190  CD1 LEU    13       -6.349   4.486   8.643  0.00  0.00      MONO
ATOM    191 HD11 LEU    13       -6.306   3.485   8.167  0.00  0.00      MONO
ATOM    192 HD12 LEU    13       -7.371   4.682   9.022  0.00  0.00      MONO
ATOM    193 HD13 LEU    13       -6.117   5.224   7.845  0.00  0.00      MONO
ATOM    194  CD2 LEU    13       -5.922   3.525  10.686  0.00  0.00      MONO
ATOM    195 HD21 LEU    13       -6.981   3.780  10.900  0.00  0.00      MONO
ATOM    196 HD22 LEU    13       -5.909   2.515  10.221  0.00  0.00      MONO
ATOM    197 HD23 LEU    13       -5.369   3.470  11.649  0.00  0.00      MONO
ATOM    198  C   LEU    13       -3.359   1.811   9.900  0.00  0.00      MONO
ATOM    199  O   LEU    13       -3.093   1.822  11.102  0.00  0.00      MONO
ATOM    200  N   TRP    14       -3.787   0.700   9.284  0.00  0.00      MONO
ATOM    201  HN  TRP    14       -4.009   0.670   8.311  0.00  0.00      MONO
ATOM    202  CA  TRP    14       -3.897  -0.560   9.994  0.00  0.00      MONO
ATOM    203  HA  TRP    14       -3.884  -0.379  11.061  0.00  0.00      MONO
ATOM    204  CB  TRP    14       -5.232  -1.302   9.675  0.00  0.00      MONO
ATOM    205  HB1 TRP    14       -5.289  -1.495   8.581  0.00  0.00      MONO
ATOM    206  HB2 TRP    14       -5.244  -2.281  10.199  0.00  0.00      MONO
ATOM    207  CG  TRP    14       -6.485  -0.542  10.131  0.00  0.00      MONO
ATOM    208  CD2 TRP    14       -7.243   0.361   9.305  0.00  0.00      MONO
ATOM    209  CD1 TRP    14       -7.092  -0.533  11.356  0.00  0.00      MONO
ATOM    210  HD1 TRP    14       -6.761  -1.086  12.224  0.00  0.00      MONO
ATOM    211  NE1 TRP    14       -8.147   0.357  11.369  0.00  0.00      MONO
ATOM    212  HE1 TRP    14       -8.740   0.409  12.143  0.00  0.00      MONO
ATOM    213  CE2 TRP    14       -8.255   0.922  10.115  0.00  0.00      MONO
ATOM    214  CE3 TRP    14       -7.099   0.727   7.967  0.00  0.00      MONO
ATOM    215  HE3 TRP    14       -6.273   0.296   7.422  0.00  0.00      MONO
ATOM    216  CZ2 TRP    14       -9.129   1.868   9.599  0.00  0.00      MONO
ATOM    217  HZ2 TRP    14       -9.937   2.327  10.157  0.00  0.00      MONO
ATOM    218  CZ3 TRP    14       -7.986   1.677   7.447  0.00  0.00      MONO
ATOM    219  HZ3 TRP    14       -7.917   1.957   6.409  0.00  0.00      MONO
ATOM    220  CH2 TRP    14       -8.981   2.245   8.257  0.00  0.00      MONO
ATOM    221  HH2 TRP    14       -9.643   3.016   7.884  0.00  0.00      MONO
ATOM    222  C   TRP    14       -2.698  -1.465   9.707  0.00  0.00      MONO
ATOM    223  O   TRP    14       -2.085  -1.468   8.639  0.00  0.00      MONO
ATOM     224  N    LEU     15          -2.402   -2.347   10.665  0.00  0.00         MONO
ATOM     225  HN   LEU     15          -2.900   -2.309   11.522  0.00  0.00         MONO
ATOM     226  CA   LEU     15          -1.530   -3.489   10.642  0.00  0.00         MONO
ATOM     227  HA   LEU     15          -1.212   -3.674    9.620  0.00  0.00         MONO
ATOM     228  CB   LEU     15          -2.571   -4.535   11.149  0.00  0.00         MONO
ATOM     229  HB1  LEU     15          -2.913   -4.181   12.151  0.00  0.00         MONO
ATOM     230  HB2  LEU     15          -3.353   -4.520   10.355  0.00  0.00         MONO
ATOM     231  CG   LEU     15          -2.286   -6.014   11.452  0.00  0.00         MONO
ATOM     232  HG   LEU     15          -3.295   -6.431   11.680  0.00  0.00         MONO
ATOM     233  CD1  LEU     15          -2.015   -7.066   10.341  0.00  0.00         MONO
ATOM     234 HD11  LEU     15          -1.493   -7.962   10.735  0.00  0.00         MONO
ATOM     235 HD12  LEU     15          -2.970   -7.412    9.897  0.00  0.00         MONO
ATOM     236 HD13  LEU     15          -1.410   -6.614    9.527  0.00  0.00         MONO
ATOM     237  CD2  LEU     15          -1.093   -6.241   12.372  0.00  0.00         MONO
ATOM     238 HD21  LEU     15          -0.991   -7.315   12.628  0.00  0.00         MONO
ATOM     239 HD22  LEU     15          -0.151   -5.922   11.884  0.00  0.00         MONO
ATOM     240 HD23  LEU     15          -1.222   -5.675   13.318  0.00  0.00         MONO
ATOM     241  C    LEU     15          -0.312   -3.188   11.544  0.00  0.00         MONO
ATOM     242  O    LEU     15          -0.439   -2.975   12.749  0.00  0.00         MONO
ATOM     243  N    TRP     16           0.898   -3.114   10.942  0.00  0.00         MONO
ATOM     244  HN   TRP     16           1.001   -3.332    9.965  0.00  0.00         MONO
ATOM     245  CA   TRP     16           2.136   -2.803   11.670  0.00  0.00         MONO
ATOM     246  HA   TRP     16           1.931   -2.829   12.729  0.00  0.00         MONO
ATOM     247  CB   TRP     16           3.255   -3.849   11.423  0.00  0.00         MONO
ATOM     248  HB1  TRP     16           3.490   -3.865   10.339  0.00  0.00         MONO
ATOM     249  HB2  TRP     16           4.178   -3.558   11.973  0.00  0.00         MONO
ATOM     250  CG   TRP     16           2.915   -5.269   11.851  0.00  0.00         MONO
ATOM     251  CD2  TRP     16           2.274   -6.241   11.007  0.00  0.00         MONO
ATOM     252  CD1  TRP     16           3.115   -5.886   13.053  0.00  0.00         MONO
ATOM     253  HD1  TRP     16           3.541   -5.424   13.937  0.00  0.00         MONO
ATOM     254  NE1  TRP     16           2.577   -7.154   13.045  0.00  0.00         MONO
ATOM     255  HE1  TRP     16           2.716   -7.791   13.765  0.00  0.00         MONO
ATOM     256  CE2  TRP     16           2.067   -7.400   11.791  0.00  0.00         MONO
ATOM     257  CE3  TRP     16           1.862   -6.191    9.675  0.00  0.00         MONO
ATOM     258  HE3  TRP     16           2.007   -5.314    9.060  0.00  0.00         MONO
ATOM     259  CZ2  TRP     16           1.443   -8.520   11.260  0.00  0.00         MONO
ATOM     260  HZ2  TRP     16           1.238   -9.380   11.881  0.00  0.00         MONO
ATOM     261  CZ3  TRP     16           1.239   -7.327    9.146  0.00  0.00         MONO
ATOM     262  HZ3  TRP     16           0.849   -7.305    8.136  0.00  0.00         MONO
ATOM     263  CH2  TRP     16           1.024   -8.476    9.923  0.00  0.00         MONO
ATOM     264  HH2  TRP     16           0.517   -9.314    9.462  0.00  0.00         MONO
ATOM     265  C    TRP     16           2.715   -1.411   11.387  0.00  0.00         MONO
ATOM     266  O    TRP     16           2.987   -1.002   10.261  0.00  0.00         MONO
ATOM     267  N    EAM     17           2.973   -0.651   12.470  0.00  0.00         MONO
ATOM     268  HN1  EAM     17           2.697   -0.964   13.371  0.00  0.00         MONO
ATOM     269  CA   EAM     17           3.476    0.697   12.387  0.00  0.00         MONO
ATOM     270  HA1  EAM     17           3.057    1.215   11.493  0.00  0.00         MONO
ATOM     271  HA2  EAM     17           3.132    1.263   13.278  0.00  0.00         MONO
ATOM     272  CB   EAM     17           5.007    0.780   12.340  0.00  0.00         MONO
ATOM     273  HB1  EAM     17           5.456    0.140   13.132  0.00  0.00         MONO
ATOM     274  HB2  EAM     17           5.355    0.410   11.350  0.00  0.00         MONO
ATOM     275  OH1  EAM     17           5.416    2.128   12.541  0.00  0.00         MONO
ATOM     276  HO1  EAM     17           5.076    2.639   11.791  0.00  0.00         MONO
ATOM     277  HA   CHO      1          -4.124    2.176   -0.153  0.00  0.00         XROT
ATOM     278  C    CHO      1          -3.382    2.570   -0.855  0.00  0.00         XROT
ATOM     279  O    CHO      1          -3.395    2.314   -2.055  0.00  0.00         XROT
ATOM     280  N    VAL      2          -2.604    3.494   -0.244  0.00  0.00         XROT
ATOM     281  HN   VAL      2          -2.679    3.730    0.726  0.00  0.00         XROT
ATOM     282  CA   VAL      2          -1.668    4.304   -1.036  0.00  0.00         XROT
ATOM     283  HA   VAL      2          -1.953    4.130   -2.066  0.00  0.00         XROT
ATOM     284  CB   VAL      2          -1.699    5.796   -0.721  0.00  0.00         XROT
ATOM     285  HB   VAL      2          -1.201    6.107    0.231  0.00  0.00         XROT
ATOM     286  CG1 VAL     2      -0.640   6.609  -1.544  0.00  0.00         XROT
ATOM     287 HG11 VAL     2      -0.736   7.693  -1.307  0.00  0.00         XROT
ATOM     288 HG12 VAL     2       0.400   6.303  -1.302  0.00  0.00         XROT
ATOM     289 HG13 VAL     2      -0.797   6.464  -2.625  0.00  0.00         XROT
ATOM     290  CG2 VAL     2      -3.105   6.352  -0.970  0.00  0.00         XROT
ATOM     291 HG21 VAL     2      -3.097   7.458  -0.891  0.00  0.00         XROT
ATOM     292 HG22 VAL     2      -3.448   6.063  -1.990  0.00  0.00         XROT
ATOM     293 HG23 VAL     2      -3.811   5.941  -0.219  0.00  0.00         XROT
ATOM     294  C   VAL     2      -0.282   3.838  -0.751  0.00  0.00         XROT
ATOM     295  O   VAL     2       0.139   3.929   0.401  0.00  0.00         XROT
ATOM     296  N   GLY     3       0.473   3.378  -1.770  0.00  0.00         XROT
ATOM     297  HN  GLY     3       0.104   3.220  -2.697  0.00  0.00         XROT
ATOM     298  CA  GLY     3       1.888   3.113  -1.494  0.00  0.00         XROT
ATOM     299  HA1 GLY     3       2.463   4.012  -1.660  0.00  0.00         XROT
ATOM     300  HA2 GLY     3       1.951   2.735  -0.513  0.00  0.00         XROT
ATOM     301  C   GLY     3       2.467   2.042  -2.344  0.00  0.00         XROT
ATOM     302  O   GLY     3       2.334   2.130  -3.557  0.00  0.00         XROT
ATOM     303  N   ALA     4       3.158   1.008  -1.786  0.00  0.00         XROT
ATOM     304  HN  ALA     4       3.297   0.983  -0.790  0.00  0.00         XROT
ATOM     305  CA  ALA     4       3.672  -0.112  -2.587  0.00  0.00         XROT
ATOM     306  HA  ALA     4       3.362   0.098  -3.583  0.00  0.00         XROT
ATOM     307  CB  ALA     4       5.186  -0.355  -2.416  0.00  0.00         XROT
ATOM     308  HB1 ALA     4       5.501  -1.192  -3.080  0.00  0.00         XROT
ATOM     309  HB2 ALA     4       5.752   0.548  -2.723  0.00  0.00         XROT
ATOM     310  HB3 ALA     4       5.436  -0.619  -1.374  0.00  0.00         XROT
ATOM     311  C   ALA     4       3.011  -1.413  -2.234  0.00  0.00         XROT
ATOM     312  O   ALA     4       2.761  -1.614  -1.057  0.00  0.00         XROT
ATOM     313  N   LEU     5       2.775  -2.301  -3.243  0.00  0.00         XROT
ATOM     314  HN  LEU     5       3.101  -2.062  -4.171  0.00  0.00         XROT
ATOM     315  CA  LEU     5       2.001  -3.549  -2.960  0.00  0.00         XROT
ATOM     316  HA  LEU     5       1.611  -3.493  -1.967  0.00  0.00         XROT
ATOM     317  CB  LEU     5       2.752  -4.940  -3.075  0.00  0.00         XROT
ATOM     318  HB1 LEU     5       3.366  -4.810  -4.001  0.00  0.00         XROT
ATOM     319  HB2 LEU     5       3.533  -5.080  -2.308  0.00  0.00         XROT
ATOM     320  CG  LEU     5       1.932  -6.271  -3.322  0.00  0.00         XROT
ATOM     321  HG  LEU     5       1.109  -5.927  -3.995  0.00  0.00         XROT
ATOM     322  CD1 LEU     5       2.719  -7.345  -4.056  0.00  0.00         XROT
ATOM     323 HD11 LEU     5       2.039  -8.047  -4.593  0.00  0.00         XROT
ATOM     324 HD12 LEU     5       3.391  -6.860  -4.773  0.00  0.00         XROT
ATOM     325 HD13 LEU     5       3.342  -7.946  -3.353  0.00  0.00         XROT
ATOM     326  CD2 LEU     5       0.932  -6.992  -2.384  0.00  0.00         XROT
ATOM     327 HD21 LEU     5       0.373  -7.778  -2.931  0.00  0.00         XROT
ATOM     328 HD22 LEU     5       1.478  -7.476  -1.547  0.00  0.00         XROT
ATOM     329 HD23 LEU     5       0.209  -6.267  -1.953  0.00  0.00         XROT
ATOM     330  C   LEU     5       0.786  -3.584  -3.849  0.00  0.00         XROT
ATOM     331  O   LEU     5       0.913  -3.371  -5.051  0.00  0.00         XROT
ATOM     332  N   ALA     6      -0.392  -3.951  -3.276  0.00  0.00         XROT
ATOM     333  HN  ALA     6      -0.477  -4.126  -2.294  0.00  0.00         XROT
ATOM     334  CA  ALA     6      -1.578  -4.147  -4.115  0.00  0.00         XROT
ATOM     335  HA  ALA     6      -1.266  -3.896  -5.105  0.00  0.00         XROT
ATOM     336  CB  ALA     6      -2.223  -5.549  -3.988  0.00  0.00         XROT
ATOM     337  HB1 ALA     6      -3.141  -5.614  -4.619  0.00  0.00         XROT
ATOM     338  HB2 ALA     6      -1.508  -6.328  -4.316  0.00  0.00         XROT
ATOM     339  HB3 ALA     6      -2.518  -5.758  -2.938  0.00  0.00         XROT
ATOM     340  C   ALA     6      -2.612  -3.136  -3.740  0.00  0.00         XROT
ATOM     341  O   ALA     6      -2.991  -3.052  -2.578  0.00  0.00         XROT
ATOM     342  N   VAL     7      -3.119  -2.332  -4.695  0.00  0.00         XROT
ATOM     343  HN  VAL     7      -2.737  -2.326  -5.627  0.00  0.00         XROT
ATOM     344  CA  VAL     7      -4.117  -1.312  -4.348  0.00  0.00         XROT
ATOM     345  HA  VAL     7      -3.926  -0.944  -3.350  0.00  0.00         XROT
ATOM     346  CB  VAL     7      -5.606  -1.727  -4.435  0.00  0.00         XROT
ATOM     347  HB  VAL     7      -5.859  -2.024  -5.481  0.00  0.00         XROT
ATOM    348  CG1 VAL     7        -5.912  -2.922  -3.513  0.00  0.00         XROT
ATOM    349 HG11 VAL     7        -6.991  -3.164  -3.545  0.00  0.00         XROT
ATOM    350 HG12 VAL     7        -5.352  -3.824  -3.830  0.00  0.00         XROT
ATOM    351 HG13 VAL     7        -5.632  -2.670  -2.468  0.00  0.00         XROT
ATOM    352  CG2 VAL     7        -6.519  -0.565  -3.988  0.00  0.00         XROT
ATOM    353 HG21 VAL     7        -7.553  -0.931  -3.816  0.00  0.00         XROT
ATOM    354 HG22 VAL     7        -6.156  -0.122  -3.037  0.00  0.00         XROT
ATOM    355 HG23 VAL     7        -6.568   0.225  -4.769  0.00  0.00         XROT
ATOM    356  C   VAL     7        -3.887  -0.144  -5.260  0.00  0.00         XROT
ATOM    357  O   VAL     7        -4.004  -0.289  -6.473  0.00  0.00         XROT
ATOM    358  N   VAL     8        -3.551   1.026  -4.675  0.00  0.00         XROT
ATOM    359  HN  VAL     8        -3.462   1.169  -3.684  0.00  0.00         XROT
ATOM    360  CA  VAL     8        -3.234   2.170  -5.525  0.00  0.00         XROT
ATOM    361  HA  VAL     8        -3.224   1.815  -6.538  0.00  0.00         XROT
ATOM    362  CB  VAL     8        -4.216   3.343  -5.368  0.00  0.00         XROT
ATOM    363  HB  VAL     8        -3.866   4.016  -4.548  0.00  0.00         XROT
ATOM    364  CG1 VAL     8        -4.300   4.151  -6.678  0.00  0.00         XROT
ATOM    365 HG11 VAL     8        -5.049   4.962  -6.569  0.00  0.00         XROT
ATOM    366 HG12 VAL     8        -3.323   4.609  -6.933  0.00  0.00         XROT
ATOM    367 HG13 VAL     8        -4.626   3.497  -7.512  0.00  0.00         XROT
ATOM    368  CG2 VAL     8        -5.617   2.842  -4.951  0.00  0.00         XROT
ATOM    369 HG21 VAL     8        -6.353   3.675  -4.960  0.00  0.00         XROT
ATOM    370 HG22 VAL     8        -5.963   2.065  -5.662  0.00  0.00         XROT
ATOM    371 HG23 VAL     8        -5.594   2.411  -3.925  0.00  0.00         XROT
ATOM    372  C   VAL     8        -1.817   2.596  -5.214  0.00  0.00         XROT
ATOM    373  O   VAL     8        -1.443   2.854  -4.065  0.00  0.00         XROT
ATOM    374  N   VAL     9        -0.898   2.704  -6.187  0.00  0.00         XROT
ATOM    375  HN  VAL     9        -1.121   2.392  -7.117  0.00  0.00         XROT
ATOM    376  CA  VAL     9         0.363   3.400  -6.031  0.00  0.00         XROT
ATOM    377  HA  VAL     9         0.778   3.201  -5.052  0.00  0.00         XROT
ATOM    378  CB  VAL     9         0.246   4.908  -6.294  0.00  0.00         XROT
ATOM    379  HB  VAL     9        -0.128   5.148  -7.311  0.00  0.00         XROT
ATOM    380  CG1 VAL     9        -0.747   5.565  -5.310  0.00  0.00         XROT
ATOM    381 HG11 VAL     9        -0.709   6.673  -5.416  0.00  0.00         XROT
ATOM    382 HG12 VAL     9        -1.792   5.248  -5.495  0.00  0.00         XROT
ATOM    383 HG13 VAL     9        -0.465   5.296  -4.272  0.00  0.00         XROT
ATOM    384  CG2 VAL     9         1.582   5.555  -5.966  0.00  0.00         XROT
ATOM    385 HG21 VAL     9         1.489   6.661  -5.976  0.00  0.00         XROT
ATOM    386 HG22 VAL     9         1.944   5.236  -4.964  0.00  0.00         XROT
ATOM    387 HG23 VAL     9         2.344   5.274  -6.729  0.00  0.00         XROT
ATOM    388  C   VAL     9         1.416   2.801  -6.946  0.00  0.00         XROT
ATOM    389  O   VAL     9         1.257   2.784  -8.163  0.00  0.00         XROT
ATOM    390  N   TRP    10         2.518   2.299  -6.349  0.00  0.00         XROT
ATOM    391  HN  TRP    10         2.641   2.266  -5.358  0.00  0.00         XROT
ATOM    392  CA  TRP    10         3.599   1.648  -7.077  0.00  0.00         XROT
ATOM    393  HA  TRP    10         3.406   1.744  -8.133  0.00  0.00         XROT
ATOM    394  CB  TRP    10         5.001   2.282  -6.803  0.00  0.00         XROT
ATOM    395  HB1 TRP    10         5.133   2.386  -5.707  0.00  0.00         XROT
ATOM    396  HB2 TRP    10         5.786   1.598  -7.197  0.00  0.00         XROT
ATOM    397  CG  TRP    10         5.235   3.632  -7.476  0.00  0.00         XROT
ATOM    398  CD2 TRP    10         5.018   4.922  -6.879  0.00  0.00         XROT
ATOM    399  CD1 TRP    10         5.616   3.869  -8.769  0.00  0.00         XROT
ATOM    400  HD1 TRP    10         5.857   3.113  -9.504  0.00  0.00         XROT
ATOM    401  NE1 TRP    10         5.527   5.217  -9.053  0.00  0.00         XROT
ATOM    402  HE1 TRP    10         5.846   5.612  -9.886  0.00  0.00         XROT
ATOM    403  CE2 TRP    10         5.153   5.883  -7.913  0.00  0.00         XROT
ATOM    404  CE3 TRP    10         4.690   5.297  -5.580  0.00  0.00         XROT
ATOM    405  HE3 TRP    10         4.555   4.584  -4.783  0.00  0.00         XROT
ATOM    406  CZ2 TRP    10         4.917   7.231  -7.676  0.00  0.00         XROT
ATOM    407  HZ2 TRP    10         4.979   7.956  -8.470  0.00  0.00         XROT
ATOM    408  CZ3 TRP    10         4.456   6.660  -5.347  0.00  0.00         XROT
ATOM    409  HZ3 TRP    10         4.066   7.003  -4.399  0.00  0.00         XROT
ATOM    410  CH2 TRP    10       4.547   7.612  -6.375  0.00  0.00      XROT
ATOM    411  HH2 TRP    10       4.322   8.629  -6.095  0.00  0.00      XROT
ATOM    412  C   TRP    10       3.641   0.146  -6.781  0.00  0.00      XROT
ATOM    413  O   TRP    10       3.651  -0.332  -5.640  0.00  0.00      XROT
ATOM    414  N   LEU    11       3.660  -0.694  -7.829  0.00  0.00      XROT
ATOM    415  HN  LEU    11       3.476  -0.259  -8.718  0.00  0.00      XROT
ATOM    416  CA  LEU    11       3.546  -2.127  -7.600  0.00  0.00      XROT
ATOM    417  HA  LEU    11       3.192  -2.264  -6.592  0.00  0.00      XROT
ATOM    418  CB  LEU    11       4.889  -2.919  -7.410  0.00  0.00      XROT
ATOM    419  HB1 LEU    11       5.540  -2.402  -6.668  0.00  0.00      XROT
ATOM    420  HB2 LEU    11       4.583  -3.980  -7.251  0.00  0.00      XROT
ATOM    421  CG  LEU    11       5.867  -3.211  -8.595  0.00  0.00      XROT
ATOM    422  HG  LEU    11       6.704  -3.781  -8.104  0.00  0.00      XROT
ATOM    423  CD1 LEU    11       5.235  -4.251  -9.534  0.00  0.00      XROT
ATOM    424 HD11 LEU    11       5.953  -4.536 -10.325  0.00  0.00      XROT
ATOM    425 HD12 LEU    11       4.957  -5.164  -8.976  0.00  0.00      XROT
ATOM    426 HD13 LEU    11       4.327  -3.852 -10.019  0.00  0.00      XROT
ATOM    427  CD2 LEU    11       6.586  -2.072  -9.331  0.00  0.00      XROT
ATOM    428 HD21 LEU    11       6.569  -2.219 -10.425  0.00  0.00      XROT
ATOM    429 HD22 LEU    11       6.168  -1.086  -9.057  0.00  0.00      XROT
ATOM    430 HD23 LEU    11       7.655  -2.085  -9.037  0.00  0.00      XROT
ATOM    431  C   LEU    11       2.475  -2.727  -8.469  0.00  0.00      XROT
ATOM    432  O   LEU    11       2.399  -2.520  -9.676  0.00  0.00      XROT
ATOM    433  N   TRP    12       1.565  -3.469  -7.824  0.00  0.00      XROT
ATOM    434  HN  TRP    12       1.558  -3.620  -6.833  0.00  0.00      XROT
ATOM    435  CA  TRP    12       0.443  -4.035  -8.523  0.00  0.00      XROT
ATOM    436  HA  TRP    12       0.556  -3.878  -9.591  0.00  0.00      XROT
ATOM    437  CB  TRP    12       0.386  -5.581  -8.344  0.00  0.00      XROT
ATOM    438  HB1 TRP    12       0.442  -5.810  -7.259  0.00  0.00      XROT
ATOM    439  HB2 TRP    12      -0.584  -5.961  -8.716  0.00  0.00      XROT
ATOM    440  CG  TRP    12       1.489  -6.375  -9.080  0.00  0.00      XROT
ATOM    441  CD2 TRP    12       2.760  -6.756  -8.504  0.00  0.00      XROT
ATOM    442  CD1 TRP    12       1.491  -6.916 -10.344  0.00  0.00      XROT
ATOM    443  HD1 TRP    12       0.803  -6.734 -11.159  0.00  0.00      XROT
ATOM    444  NE1 TRP    12       2.641  -7.666 -10.548  0.00  0.00      XROT
ATOM    445  HE1 TRP    12       2.958  -8.096 -11.368  0.00  0.00      XROT
ATOM    446  CE2 TRP    12       3.433  -7.566  -9.440  0.00  0.00      XROT
ATOM    447  CE3 TRP    12       3.346  -6.474  -7.280  0.00  0.00      XROT
ATOM    448  HE3 TRP    12       2.827  -5.856  -6.563  0.00  0.00      XROT
ATOM    449  CZ2 TRP    12       4.686  -8.110  -9.165  0.00  0.00      XROT
ATOM    450  HZ2 TRP    12       5.196  -8.747  -9.874  0.00  0.00      XROT
ATOM    451  CZ3 TRP    12       4.607  -7.039  -6.987  0.00  0.00      XROT
ATOM    452  HZ3 TRP    12       4.902  -6.986  -5.949  0.00  0.00      XROT
ATOM    453  CH2 TRP    12       5.256  -7.865  -7.905  0.00  0.00      XROT
ATOM    454  HH2 TRP    12       6.191  -8.337  -7.634  0.00  0.00      XROT
ATOM    455  C   TRP    12      -0.879  -3.350  -8.163  0.00  0.00      XROT
ATOM    456  O   TRP    12      -1.177  -2.845  -7.079  0.00  0.00      XROT
ATOM    457  N   LEU    13      -1.776  -3.391  -9.142  0.00  0.00      XROT
ATOM    458  HN  LEU    13      -1.561  -3.784 -10.028  0.00  0.00      XROT
ATOM    459  CA  LEU    13      -3.159  -3.056  -9.029  0.00  0.00      XROT
ATOM    460  HA  LEU    13      -3.414  -2.847  -7.995  0.00  0.00      XROT
ATOM    461  CB  LEU    13      -3.783  -4.359  -9.571  0.00  0.00      XROT
ATOM    462  HB1 LEU    13      -3.331  -4.452 -10.592  0.00  0.00      XROT
ATOM    463  HB2 LEU    13      -3.343  -5.258  -9.080  0.00  0.00      XROT
ATOM    464  CG  LEU    13      -5.287  -4.568  -9.758  0.00  0.00      XROT
ATOM    465  HG  LEU    13      -5.393  -5.654  -9.998  0.00  0.00      XROT
ATOM    466  CD1 LEU    13      -6.349  -4.486  -8.643  0.00  0.00      XROT
ATOM    467 HD11 LEU    13      -6.306  -3.485  -8.167  0.00  0.00      XROT
ATOM    468 HD12 LEU    13      -7.371  -4.682  -9.022  0.00  0.00      XROT
ATOM    469 HD13 LEU    13      -6.117  -5.224  -7.845  0.00  0.00      XROT
ATOM    470  CD2 LEU    13      -5.922  -3.525 -10.686  0.00  0.00      XROT
ATOM    471 HD21 LEU    13      -6.981  -3.780 -10.900  0.00  0.00      XROT
ATOM    472 HD22 LEU    13       -5.909  -2.515 -10.221  0.00   0.00        XROT
ATOM    473 HD23 LEU    13       -5.369  -3.470 -11.649  0.00   0.00        XROT
ATOM    474  C   LEU    13       -3.359  -1.811  -9.900  0.00   0.00        XROT
ATOM    475  O   LEU    13       -3.093  -1.822 -11.102  0.00   0.00        XROT
ATOM    476  N   TRP    14       -3.787  -0.700  -9.284  0.00   0.00        XROT
ATOM    477  HN  TRP    14       -4.009  -0.670  -8.311  0.00   0.00        XROT
ATOM    478  CA  TRP    14       -3.897   0.560  -9.994  0.00   0.00        XROT
ATOM    479  HA  TRP    14       -3.884   0.379 -11.061  0.00   0.00        XROT
ATOM    480  CB  TRP    14       -5.232   1.302  -9.675  0.00   0.00        XROT
ATOM    481  HB1 TRP    14       -5.289   1.495  -8.581  0.00   0.00        XROT
ATOM    482  HB2 TRP    14       -5.244   2.281 -10.199  0.00   0.00        XROT
ATOM    483  CG  TRP    14       -6.485   0.542 -10.131  0.00   0.00        XROT
ATOM    484  CD2 TRP    14       -7.243  -0.361  -9.305  0.00   0.00        XROT
ATOM    485  CD1 TRP    14       -7.092   0.533 -11.356  0.00   0.00        XROT
ATOM    486  HD1 TRP    14       -6.761   1.086 -12.224  0.00   0.00        XROT
ATOM    487  NE1 TRP    14       -8.147  -0.357 -11.369  0.00   0.00        XROT
ATOM    488  HE1 TRP    14       -8.740  -0.409 -12.143  0.00   0.00        XROT
ATOM    489  CE2 TRP    14       -8.255  -0.922 -10.115  0.00   0.00        XROT
ATOM    490  CE3 TRP    14       -7.099  -0.727  -7.967  0.00   0.00        XROT
ATOM    491  HE3 TRP    14       -6.273  -0.296  -7.422  0.00   0.00        XROT
ATOM    492  CZ2 TRP    14       -9.129  -1.868  -9.599  0.00   0.00        XROT
ATOM    493  HZ2 TRP    14       -9.937  -2.327 -10.157  0.00   0.00        XROT
ATOM    494  CZ3 TRP    14       -7.986  -1.677  -7.447  0.00   0.00        XROT
ATOM    495  HZ3 TRP    14       -7.917  -1.957  -6.409  0.00   0.00        XROT
ATOM    496  CH2 TRP    14       -8.981  -2.245  -8.257  0.00   0.00        XROT
ATOM    497  HH2 TRP    14       -9.643  -3.016  -7.884  0.00   0.00        XROT
ATOM    498  C   TRP    14       -2.698   1.465  -9.707  0.00   0.00        XROT
ATOM    499  O   TRP    14       -2.085   1.468  -8.639  0.00   0.00        XROT
ATOM    500  N   LEU    15       -2.402   2.347 -10.665  0.00   0.00        XROT
ATOM    501  HN  LEU    15       -2.900   2.309 -11.522  0.00   0.00        XROT
ATOM    502  CA  LEU    15       -1.530   3.489 -10.642  0.00   0.00        XROT
ATOM    503  HA  LEU    15       -1.212   3.674  -9.620  0.00   0.00        XROT
ATOM    504  CB  LEU    15       -2.571   4.535 -11.149  0.00   0.00        XROT
ATOM    505  HB1 LEU    15       -2.913   4.181 -12.151  0.00   0.00        XROT
ATOM    506  HB2 LEU    15       -3.353   4.520 -10.355  0.00   0.00        XROT
ATOM    507  CG  LEU    15       -2.286   6.014 -11.452  0.00   0.00        XROT
ATOM    508  HG  LEU    15       -3.295   6.431 -11.680  0.00   0.00        XROT
ATOM    509  CD1 LEU    15       -2.015   7.066 -10.341  0.00   0.00        XROT
ATOM    510 HD11 LEU    15       -1.493   7.962 -10.735  0.00   0.00        XROT
ATOM    511 HD12 LEU    15       -2.970   7.412  -9.897  0.00   0.00        XROT
ATOM    512 HD13 LEU    15       -1.410   6.614  -9.527  0.00   0.00        XROT
ATOM    513  CD2 LEU    15       -1.093   6.241 -12.372  0.00   0.00        XROT
ATOM    514 HD21 LEU    15       -0.991   7.315 -12.628  0.00   0.00        XROT
ATOM    515 HD22 LEU    15       -0.151   5.922 -11.884  0.00   0.00        XROT
ATOM    516 HD23 LEU    15       -1.222   5.675 -13.318  0.00   0.00        XROT
ATOM    517  C   LEU    15       -0.312   3.188 -11.544  0.00   0.00        XROT
ATOM    518  O   LEU    15       -0.439   2.975 -12.749  0.00   0.00        XROT
ATOM    519  N   TRP    16        0.898   3.114 -10.942  0.00   0.00        XROT
ATOM    520  HN  TRP    16        1.001   3.332  -9.965  0.00   0.00        XROT
ATOM    521  CA  TRP    16        2.136   2.803 -11.670  0.00   0.00        XROT
ATOM    522  HA  TRP    16        1.931   2.829 -12.729  0.00   0.00        XROT
ATOM    523  CB  TRP    16        3.255   3.849 -11.423  0.00   0.00        XROT
ATOM    524  HB1 TRP    16        3.490   3.865 -10.339  0.00   0.00        XROT
ATOM    525  HB2 TRP    16        4.178   3.558 -11.973  0.00   0.00        XROT
ATOM    526  CG  TRP    16        2.915   5.269 -11.851  0.00   0.00        XROT
ATOM    527  CD2 TRP    16        2.274   6.241 -11.007  0.00   0.00        XROT
ATOM    528  CD1 TRP    16        3.115   5.886 -13.053  0.00   0.00        XROT
ATOM    529  HD1 TRP    16        3.541   5.424 -13.937  0.00   0.00        XROT
ATOM    530  NE1 TRP    16        2.577   7.154 -13.045  0.00   0.00        XROT
ATOM    531  HE1 TRP    16        2.716   7.791 -13.765  0.00   0.00        XROT
ATOM    532  CE2 TRP    16        2.067   7.400 -11.791  0.00   0.00        XROT
ATOM    533  CE3 TRP    16        1.862   6.191  -9.675  0.00   0.00        XROT
ATOM    534  HE3 TRP    16         2.007   5.314  -9.060  0.00  0.00        XROT
ATOM    535  CZ2 TRP    16         1.443   8.520 -11.260  0.00  0.00        XROT
ATOM    536  HZ2 TRP    16         1.238   9.380 -11.881  0.00  0.00        XROT
ATOM    537  CZ3 TRP    16         1.239   7.327  -9.146  0.00  0.00        XROT
ATOM    538  HZ3 TRP    16         0.849   7.305  -8.136  0.00  0.00        XROT
ATOM    539  CH2 TRP    16         1.024   8.476  -9.923  0.00  0.00        XROT
ATOM    540  HH2 TRP    16         0.517   9.314  -9.462  0.00  0.00        XROT
ATOM    541  C   TRP    16         2.715   1.411 -11.387  0.00  0.00        XROT
ATOM    542  O   TRP    16         2.987   1.002 -10.261  0.00  0.00        XROT
ATOM    543  N   EAM    17         2.973   0.651 -12.470  0.00  0.00        XROT
ATOM    544  HN1 EAM    17         2.697   0.964 -13.371  0.00  0.00        XROT
ATOM    545  CA  EAM    17         3.476  -0.697 -12.387  0.00  0.00        XROT
ATOM    546  HA1 EAM    17         3.057  -1.215 -11.493  0.00  0.00        XROT
ATOM    547  HA2 EAM    17         3.132  -1.263 -13.278  0.00  0.00        XROT
ATOM    548  CB  EAM    17         5.007  -0.780 -12.340  0.00  0.00        XROT
ATOM    549  HB1 EAM    17         5.456  -0.140 -13.132  0.00  0.00        XROT
ATOM    550  HB2 EAM    17         5.355  -0.410 -11.350  0.00  0.00        XROT
ATOM    551  OH1 EAM    17         5.416  -2.128 -12.541  0.00  0.00        XROT
ATOM    552  HO1 EAM    17         5.076  -2.639 -11.791  0.00  0.00        XROT
END
```

A.3 Analysis of the Final Structure

A.3.1 Torc Output

The experimental observables used in the determination and refinement of the final structure are compared here to the observables calculated from the final structure coordinates. The individual and final penalties are also shown. The residue numbering scheme follows that shown in Appendix A.2. An asterisk next to a value in the calc-obs column indicates a value outside of the range of the defined experimental error. The first column names the interaction: ncs for ^{15}N chemical shift, ccs for $^{13}C_1$ chemical shift, nc for ^{15}N-$^{13}C_1$ dipolar splitting, nh for ^{15}N-^{1}H dipolar splitting, ih for the Trp indole ^{15}N-^{1}H dipolar splitting, dis for the defined intramolecular hydrogen bond distances, cd for C-^{2}H quadrupolar splittings, and e for the CHARMM energy.

```
                      calc        obs      calc-obs   expError
                    ---------  ---------  ---------  ---------
ncs VAL   2  =      197.8874   198.0000    -0.1126     5.0000
ncs GLY   3  =      113.0915   113.0000     0.0915     5.0000
ncs ALA   4  =      199.0521   198.0000     1.0521     5.0000
ncs LEU   5  =      146.3203   145.0000     1.3203     5.0000
ncs ALA   6  =      197.4664   198.0000    -0.5336     5.0000
ncs VAL   7  =      145.6016   145.0000     0.6016     5.0000
ncs VAL   8  =      195.8142   196.0000    -0.1858     5.0000
ncs VAL   9  =      144.8537   145.0000    -0.1463     5.0000
ncs TRP  10  =      196.4232   198.0000    -1.5768     5.0000
ncs LEU  11  =      143.3842   144.0000    -0.6158     5.0000
ncs TRP  12  =      184.2977   185.0000    -0.7023     5.0000
ncs LEU  13  =      133.3673   132.0000     1.3673     5.0000
ncs TRP  14  =      182.5719   182.0000     0.5719     5.0000
ncs LEU  15  =      131.8295   131.0000     0.8295     5.0000
ncs TRP  16  =      183.1432   181.0000     2.1432     5.0000

ccs GLY   3  =      174.7820   175.0000    -0.2180     5.0000
ccs LEU  11  =      179.8535   180.0000    -0.1465     5.0000

ics TRP  10  =      145.5269   145.0000     0.5269     5.0000
ics TRP  12  =      143.2984   144.0000    -0.7016     5.0000
ics TRP  14  =      144.8905   144.0000     0.8905     5.0000
ics TRP  16  =      139.8278   139.0000     0.8278     5.0000
```

```
nc  VAL   2  =    0.4948    0.4630    0.0318    0.1000 angle =  63.2
nc  GLY   3  =    0.9044    0.9100   -0.0056    0.1000 angle = 139.1
nc  ALA   4  =    0.6319    0.6700   -0.0381    0.1000 angle =  65.8
nc  LEU   5  =    0.8139    0.8200   -0.0061    0.1000 angle = 137.7
nc  ALA   6  =    0.5932    0.5720    0.0212    0.1000 angle =  65.1
nc  VAL   7  =    0.6448    0.6260    0.0188    0.1000 angle = 135.1
nc  VAL   8  =    0.5556    0.5190    0.0366    0.1000 angle =  64.3
nc  VAL   9  =    0.7320    0.7020    0.0300    0.1000 angle = 136.5
nc  TRP  10  =    0.5259    0.4870    0.0389    0.1000 angle =  63.8
nc  TRP  12  =    0.3882    0.3650    0.0232    0.1000 angle =  61.2
nc  LEU  13  =    0.8001    0.7790    0.0211    0.1000 angle = 137.5
nc  TRP  14  =    0.4649    0.4540    0.0109    0.1000 angle =  62.6
nc  LEU  15  =    0.6905    0.6570    0.0335    0.1000 angle = 135.8
nc  TRP  16  =    0.5168    0.5070    0.0098    0.1000 angle =  63.6

nh  VAL   2  =   20.5873   19.7000    0.8873    2.0000 angle = 165.7
nh  GLY   3  =   17.2948   17.6000   -0.3052    2.0000 angle =  23.4
nh  ALA   4  =   22.0076   21.8000    0.2076    2.0000 angle = 172.0
nh  LEU   5  =   17.2604   17.2000    0.0604    2.0000 angle =  23.5
nh  ALA   6  =   21.3853   20.7000    0.6853    2.0000 angle = 168.8
nh  VAL   7  =   17.7835   18.2000   -0.4165    2.0000 angle =  22.3
nh  VAL   8  =   21.7106   22.0000   -0.2894    2.0000 angle = 170.3
nh  VAL   9  =   17.7314   17.8000   -0.0686    2.0000 angle =  22.4
nh  TRP  10  =   22.1168   20.7000    1.4168    2.0000 angle = 172.7
nh  LEU  11  =   15.1844   14.6000    0.5844    2.0000 angle =  28.0
nh  TRP  12  =   21.9058   20.9000    1.0058    2.0000 angle = 171.4
nh  LEU  13  =   15.7517   16.2000   -0.4483    2.0000 angle =  26.8
nh  TRP  14  =   20.9581   21.1000   -0.1419    2.0000 angle = 167.0
nh  LEU  15  =   14.0455   14.3000   -0.2545    2.0000 angle =  30.2
nh  TRP  16  =   20.7191   21.5000   -0.7809    2.0000 angle = 166.1

ih  TRP  10  =   12.7142   13.2000   -0.4858    2.0000 angle =  31.3
ih  TRP  12  =   11.3878   11.1000    0.2878    2.0000 angle =  33.1
ih  TRP  14  =    9.3925   10.1000   -0.7075    2.0000 angle =  37.5
ih  TRP  16  =    7.0661    7.7000   -0.6339    2.0000 angle =  42.2

dis 1O-8HN   =    1.9919    1.9600    0.0319    0.3000
dis 3HN-8O   =    2.0967    1.9600    0.1367    0.3000
dis 3O-10HN  =    1.8321    1.9600   -0.1279    0.3000
dis 5HN-10O  =    2.3358    1.9600    0.3758*   0.3000
dis 5O-12HN  =    1.9110    1.9600   -0.0490    0.3000
dis 7HN-12O  =    2.1941    1.9600    0.2341    0.3000
dis 7O-14HN  =    1.8772    1.9600   -0.0828    0.3000
dis 9HN-14O  =    2.0247    1.9600    0.0647    0.3000
dis 9O-16HN  =    1.9015    1.9600   -0.0585    0.3000
dis 11HN-16O =    2.0517    1.9600    0.0917    0.3000
dis 1O-8N    =    2.9233    2.9100    0.0133    0.3000
dis 3N-8O    =    3.0342    2.9100    0.1242    0.3000
dis 3O-10N   =    2.8039    2.9100   -0.1061    0.3000
dis 5N-10O   =    3.2238    2.9100    0.3138*   0.3000
dis 5O-12N   =    2.8506    2.9100   -0.0594    0.3000
dis 7N-12O   =    3.1174    2.9100    0.2074    0.3000
dis 7O-14N   =    2.8495    2.9100   -0.0605    0.3000
dis 9N-14O   =    2.9914    2.9100    0.0814    0.3000
dis 9O-16N   =    2.8215    2.9100   -0.0885    0.3000
dis 11N-16O  =    3.0402    2.9100    0.1302    0.3000
```

```
cd  VAL   2  =  205.8170  205.0000   0.8170   5.0000 angle =  17.9
cd  VAL   2  =  134.7717  133.4000   1.3717   5.0000 angle = 148.4
cd  VAL   2  =    6.2539    9.0000  -2.7461   5.0000 angle =  58.3
cd  VAL   2  =   33.1533   31.3000   1.8533   5.0000 angle =  80.7
cd  GLY   3  =  111.5186  111.6000  -0.0814   5.0000 angle =  81.2
cd  GLY   3  =  192.4566  192.2000   0.2566   5.0000 angle = 158.7
cd  ALA   4  =  195.5523  195.4000   0.1523   5.0000 angle =  20.6
cd  ALA   4  =   36.8416   32.8000   4.0416   5.0000 angle =  96.4
cd  LEU   5  =  191.1429  191.0000   0.1429   5.0000 angle = 158.4
cd  LEU   5  =  120.9985  121.0000  -0.0015   5.0000 angle =  34.2
cd  LEU   5  =   51.5136   51.2000   0.3136   5.0000 angle = 134.0
cd  LEU   5  =    9.0690    8.3000   0.7690   5.0000 angle =  53.0
cd  LEU   5  =   29.4882   29.0000   0.4882   5.0000 angle =  61.2
cd  LEU   5  =    9.8639    8.3000   1.5639   5.0000 angle = 127.3
cd  ALA   6  =  189.3186  188.9000   0.4186   5.0000 angle =  22.0
cd  ALA   6  =   37.4792   35.7000   1.7792   5.0000 angle =  94.7
cd  VAL   7  =  187.0947  187.0000   0.0947   5.0000 angle = 157.5
cd  VAL   7  =  187.2339  187.0000   0.2339   5.0000 angle =  20.5
cd  VAL   7  =    2.7416    3.3000  -0.5584   5.0000 angle = 126.8
cd  VAL   7  =   26.9335   26.1000   0.8335   5.0000 angle = 106.8
cd  VAL   8  =  200.5815  200.0000   0.5815   5.0000 angle =  19.3
cd  VAL   8  =   70.9837   70.9000   0.0837   5.0000 angle = 137.3
cd  VAL   8  =   42.0157   41.8000   0.2157   5.0000 angle =  31.8
cd  VAL   8  =   28.1458   27.0000   1.1458   5.0000 angle = 105.6
cd  VAL   9  =  174.7187  174.0000   0.7187   5.0000 angle = 154.8
cd  VAL   9  =  174.3695  174.0000   0.3695   5.0000 angle =  23.6
cd  VAL   9  =    7.8195    8.1000  -0.2805   5.0000 angle = 129.6
cd  VAL   9  =   30.9835   27.3000   3.6835   5.0000 angle = 102.4
cd  TRP  10  =   46.0145   46.0000   0.0145   5.0000 angle =  47.2
cd  TRP  10  =   86.7751   87.0000  -0.2249   5.0000 angle = 137.7
cd  TRP  10  =  155.0605  155.0000   0.0605   5.0000 angle = 151.3
cd  TRP  10  =  102.3379  102.0000   0.3379   5.0000 angle = 105.0
cd  TRP  10  =   84.9548   85.0000  -0.0452   5.0000 angle =  42.5
cd  LEU  11  =  195.3748  196.0000  -0.6252   5.0000 angle = 159.4
cd  LEU  11  =   38.0792   37.5000   0.5792   5.0000 angle = 131.8
cd  LEU  11  =  107.7402  110.5000  -2.7598   5.0000 angle =  98.2
cd  LEU  11  =   49.2236   49.1000   0.1236   5.0000 angle = 115.9
cd  LEU  11  =   12.2494   11.5000   0.7494   5.0000 angle =  52.4
cd  LEU  11  =   31.9523   31.4000   0.5523   5.0000 angle =  61.4
cd  TRP  12  =   76.6526   77.0000  -0.3474   5.0000 angle =  41.2
cd  TRP  12  =   43.1451   43.0000   0.1451   5.0000 angle = 131.6
cd  TRP  12  =  191.0149  192.0000  -0.9851   5.0000 angle = 163.9
cd  TRP  12  =   99.2901   99.0000   0.2901   5.0000 angle = 104.5
cd  TRP  12  =   38.7762   39.0000  -0.2238   5.0000 angle =  49.0
cd  LEU  13  =  206.8554  207.0000  -0.1446   5.0000 angle = 162.3
cd  LEU  13  =  171.1408  170.6000   0.5408   5.0000 angle =  24.3
cd  LEU  13  =   48.0770   48.5000  -0.4230   5.0000 angle = 116.1
cd  LEU  13  =   74.2755   76.3000  -2.0245   5.0000 angle =  77.6
cd  LEU  13  =   46.0466   43.2000   2.8466   5.0000 angle = 136.3
cd  LEU  13  =    7.8910    6.4000   1.4910   5.0000 angle =  52.8
cd  TRP  14  =  107.8030  108.0000  -0.1970   5.0000 angle =  36.6
cd  TRP  14  =   31.6939   32.0000  -0.3061   5.0000 angle = 120.3
cd  TRP  14  =  203.5633  204.0000  -0.4367   5.0000 angle = 164.5
cd  TRP  14  =   80.6487   81.0000  -0.3513   5.0000 angle = 110.2
cd  TRP  14  =   27.6303   28.0000  -0.3697   5.0000 angle =  59.0
cd  LEU  15  =  198.7786  199.0000  -0.2214   5.0000 angle = 160.2
cd  LEU  15  =  162.5842  163.2000  -0.6158   5.0000 angle =  26.2
cd  LEU  15  =   60.0580   60.4000  -0.3420   5.0000 angle = 135.4
cd  LEU  15  =   75.4329   79.1000  -3.6671   5.0000 angle =  78.2
cd  LEU  15  =   40.3482   38.4000   1.9482   5.0000 angle = 135.6
cd  LEU  15  =    7.1969    5.5000   1.6969   5.0000 angle =  52.8
cd  TRP  16  =  122.8873  123.0000  -0.1127   5.0000 angle =  35.4
cd  TRP  16  =     4.0191    4.0000     0.0191     5.0000 angle = 124.7
cd  TRP  16  =   198.0052  198.0000     0.0052     5.0000 angle = 158.9
cd  TRP  16  =    59.0675   59.0000     0.0675     5.0000 angle = 115.2
cd  TRP  16  =     1.0022    1.0000     0.0022     5.0000 angle =  54.9

       penalty     lambda    penalty
NCS     0.2882     3.0000     0.8646
CCS     0.0014     3.0000     0.0041
ICS     0.0450     3.0000     0.1349
NC      0.4703     3.0000     1.4110
NH      0.7394     3.0000     2.2181
IH      0.1526     3.0000     0.4579
DIS     2.5794     3.0000     7.7381
CD      2.0118     3.0000     6.0355
E     370.2383     1.0000   370.2383
                          ----------
total                       389.1025
```

A.3.2 Procheck Output

The final structure was analyzed using Procheck (Laskowski *et al.*, 1993). Procheck is a suite of programs for assessing the overall quality of a protein structure and for giving an indication of its local, residue-by-residue reliability. The programs are intended as a tool for highlighting regions of a structure that might need closer examination, and for indicating whether the structure is more or less reliable than others solved at the same resolution.

Many of the results are in need of clarification due to the alternating stereochemistry of the amino acids in this peptide. The secondary structure was identified as an extended strand, participating in a beta ladder, which can be identified as a beta helix. The residues all map into the core β region of a Ramachandran map (Ramachandran and Sassiekharan, 1968), but the Procheck analysis does not plot the D amino acids separately, resulting in the D amino acids being plotted in non-allowed regions.

As mentioned in Chapter 4, the ω torsion angle after refinement is not 180°. Procheck allows 5.8° standard deviation from 180°, but several of the peptide linkages in gA are beyond this limit due to the imposed experimental constraints. For example, the peptide linkage between Val_7 and Val_8 is 164.6°.

The only bad contact reported is between C_1 on Trp_{15} and N on the ethanolamine, but these atoms are directly bonded. This bad contact report can be ignored since Procheck does not recognize the ethanolamine blocking group.

The covalent geometry is very reasonable except for three instances in which the N-C_α-C_β bond angle is large. The values for the covalent analysis are taken from small molecule data (Engh and Huber, 1991). The values in gA are highly influenced by the experimental constraints, most likely the C_α-^2H quadrupolar splittings influences the N-C_α-C_β bond angles. Keeping these justifications in mind, the final structure passes well the Procheck analysis.

```
Residue-by-residue listing for avgref_iupac
-------------------------------------------

This listing highlights the residues in the structure which may need investigation.
The ideal values and standard deviations against which the structure has been compared are shown in the following table:

                                   <------------------------------ I D E A L   V A L U E S ------------------------------>

                                   Chi-1 dihedral        Proline Phi   Helix  Chi-3    Chi-3 Disulph Omega  H-bond Chirality
                                   g(-) trans g(+) Chi-2  phi   helix  psi   rt-hand  lf-hand bond dihedral  en.   C-alpha
                                   -------------------------------------------------------------------------------------
     Ideal value                   64.1 183.6 -66.7 177.4 -65.4 -65.3 -39.4   96.8    -85.8    2.0  180.0   -2.0     33.9
     Standard deviation            15.7  16.8  15.0  18.5  11.2  11.9  11.3   14.8     10.7    0.1    5.8    0.8      3.5
                                   -------------------------------------------------------------------------------------

In the listing below, properties that deviate from these values are highlighted by asterisks and plus-signs. Each asterisk
represents one standard deviation, and each plus-sign represents half a standard deviation. So, a highlight such as +***, indicates
that the value of the parameter is between 3.5 and 4.0 standard deviations from the ideal value shown above.
Where the deviation is greater than 4.5 standard deviations its numerical value is shown; for example, *5.5*.

The final column gives the maximum deviation in each row, while the maximum column deviations are shown at the end of the listing.
Also at the end are the keys to the codes used for the secondary structure and Ramachandran plot assignments.

.....................................................................................................................................
 Residue     Kabsch Region
---------    Sander   of
No.  Type Seq  sec  Ramch.  Chi-1 dihedral    Chi-2 Proline Phi  Helix  Chi-3    Chi-3 Disulph Omega  H-bond Chirality   Bad      Max
  Chain   no. struc  plot   g(-) trans g(+)   trans   phi  helix  psi  rt-hand  lf-hand  bond dihedral  en.   C-alpha  contacts   dev
-------------------------------------------------------------------------------------------------------------------------------------
    1A VAL   1           -     -  177.4    -      -      -     -     -     -        -       -    171.3     -      34.3       -
                                                                                                  +*                                +*
    2A GLY   2   E       -     -     -     -      -      -     -     -     -        -       -    176.1   -2.0       -        -
    3A ALA   3   E       B     -     -     -      -      -     -     -     -        -       -    185.9     -      34.8       -
                                                                                                   *                                 *
    4A LEU   4   E      XX     -  202.7    -    151.9    -     -     -     -        -       -    186.7   -1.4 D> -30.5       -
                       ****          *            *                                                *          *18.4*           *18.4*
    5A ALA   5   E       B     -     -     -      -      -     -     -     -        -       -    181.2     -      33.8       -
    6A VAL   6   E      XX   58.9    -     -      -      -     -     -     -        -       -    176.2   -1.8 D> -31.6       -
                       ****                                                                                   *18.7*           *18.7*
    7A VAL   7   E       B     -  208.6    -      -      -     -     -     -        -       -    164.6     -      32.1       -
                                     *                                                            +**                               +**
    8A VAL   8   E      XX   60.4    -     -      -      -     -     -     -        -       -    177.8   -2.3 D> -33.1       -
                       ****                                                                                   *19.1*           *19.1*
    9A TRP   9   E       B     -     -   -74.2    -      -     -     -     -        -       -    187.0   -3.0     32.5       -
                                                                                                   *       *                         *
   10A LEU  10   E      XX     -     -   -72.6    -      -     -     -     -        -       -    176.3   -2.2 D> -23.1       -
                       ****                                                                                   *16.3*           *16.3*
   11A TRP  11   E       B     -     -   -70.6    -      -     -     -     -        -       -    168.2   -2.5     31.6       -
                                                                                                  **                                **
   12A LEU  12   E      XX     -  183.2    -      -      -     -     -     -        -       -    174.6     -  D> -38.5       -
                       ****                                                                                   *20.7*           *20.7*
   13A TRP  13   E       B     -     -   -63.8    -      -     -     -     -        -       -    167.1   -2.9     32.8       -
                                                                                                  **       *                        **
   14A LEU  14   E      XX     -  183.9    -      -      -     -     -     -        -       -    178.5     -  D> -38.4       -
                       ****                                                                                   *20.7*           *20.7*
   15A TRP  15   E       -     -     -   -60.6    -      -     -     -     -        -       -      -     -2.8     33.1       1
                                                                                                                             *       *
-------------------------------------------------------------------------------------------------------------------------------------
Max deviations:        ****          *            *                                               +**      *    *20.7*       *   *20.7*
-------------------------------------------------------------------------------------------------------------------------------------
Mean values:                 59.7 191.2 -68.4 151.9    -     -     -     -        -       -    176.5   -2.3      5.0
                                                  *                                                            *8.3*            *8.3*
Standard deviations:          1.1  13.6   5.9   0.0    -     -     -     -        -       -      7.1    0.5     33.9

Numbers of values:              2     5     5     1    0     0     0     0        0       0       14      9       14         1

                                           Number of D-amino acids (labelled D>):      6

   KEY TO CODES:
   ------------

        Regions of the Ramachandran plot                         Secondary structure (extended Kabsch/Sander)
        --------------------------------                         --------------------------------------------

        A  - Core alpha                                          B - residue in isolated beta-bridge
        a  - Allowed alpha                                       E - extended strand, participates in beta-ladder
        ~a - Generous alpha                ** Generous           G - 3-helix (3/10 helix)
        B  - Core beta                                           H - 4-helix (alpha-helix)
        b  - Allowed beta                                        I - 5-helix (pi-helix)
        ~b - Generous beta                 ** Generous           S - bend
        L  - Core left-handed alpha                              T - hydrogen-bonded turn
        l  - Allowed left-handed alpha
        ~l - Generous left-handed alpha    ** Generous           e - extension of beta-strand
        p  - Allowed epsilon                                     g - extension of 3/10 helix
        ~p - Generous epsilon              ** Generous           h - extension of alpha-helix
        XX - Outside major areas         **** Disallowed
```

```
                       M A I N   C H A I N   B O N D   L E N G T H S   A N D   B O N D   A N G L E S

                  ..................................... Small molecule data .........................................
                  <---------- Bond lengths ---------->  <--------------------- Bond angles ---------------------->
                        C-N      C-O     CA-C    CA-CB     N-CA   C-N-CA   CA-C-N   CA-C-O  CB-CA-C   N-CA-C  N-CA-CB    O-C-N
                  ---------------------------------------------------------------------------------------------------------
Any                       -    1.231        -        -        -        -        -        -        -        -        -        -
                            ( 0.020)
Pro                   1.341        -        -        -    1.466   122.60   116.90        -        -   111.80   103.00   122.00
                   ( 0.016)                            ( 0.015) (  5.00) (  1.50)                   (  2.50) (  1.10) (  1.40)
Except Pro            1.329        -        -        -        -        -        -        -        -        -        -   123.00
                   ( 0.014)                                                                                           (  1.60)
Gly                       -        -    1.516        -    1.451   120.60   116.40   120.80        -   112.50        -        -
                                     ( 0.018)          ( 0.016) (  1.70) (  2.10) (  2.10)          (  2.90)
Except Gly                -        -    1.525        -        -        -        -   120.80        -        -        -        -
                                     ( 0.021)                                     (  1.70)
Ala                       -        -        -    1.521        -        -        -        -   110.50        -   110.40        -
                                              ( 0.033)                                     (  1.50)          (  1.50)
Ile,Thr,Val               -        -        -    1.540        -        -        -        -   109.10        -   111.50        -
                                              ( 0.027)                                     (  2.20)          (  1.70)
Except Gly,Pro            -        -        -        -    1.458   121.70   116.20        -        -   111.20        -        -
                                                       ( 0.019) (  1.80) (  2.00)                   (  2.80)
The rest                  -        -        -    1.530        -        -        -        -   110.10        -   110.50        -
                                              ( 0.020)                                     (  1.90)          (  1.70)

Note. The table above shows the mean values obtained from small molecule data by Engh & Huber (1991). The values shown in brackets
      are standard deviations

.........................................................................................................................................
  Residue
-------------   <---------- Bond lengths ---------->  <--------------------- Bond angles ---------------------->
No.  Type Seq                                                                                                                   Max
 Chain    no.        C-N      C-O     CA-C    CA-CB     N-CA   C-N-CA   CA-C-N   CA-C-O  CB-CA-C   N-CA-C  N-CA-CB    O-C-N            dev
-----------------------------------------------------------------------------------------------------------------------------------------
   1A FOR  1A          -    1.227        -        -        -        -        -        -        -        -        -   126.15
                                                                                                                           +*              +*
   2A VAL   1      1.354    1.230    1.490    1.525    1.469   119.94   118.87   118.33   106.58   108.49   114.52   122.76
                      +*                +*                                   *        *        *                 +*                        +*
   3A GLY   2      1.349    1.223    1.485        -    1.466   117.37   120.70   118.25        -   113.52        -   121.03
                       *                +*                          +*       **        *                                    *              **
   4A ALA   3      1.363    1.220    1.501    1.543    1.470   122.16   117.79   117.38   105.63   112.22   113.75   124.80
                      **                 *                                            **      ***                **        *             ***
   5A LEU   4      1.365    1.227    1.506    1.585    1.495   119.51   117.15   119.38   108.61   108.98   118.26   123.29
                     +**                        +**       +*        *                                         *4.6*                     *4.6*
   6A ALA   5      1.360    1.225    1.494    1.548    1.466   119.71   119.10   119.91   107.69   109.01   114.28   120.99
                      **                 *                           *        *                +*               +**        *             +**
   7A VAL   6      1.347    1.227    1.500    1.548    1.469   120.18   116.71   119.63   108.79   107.04   117.01   123.66
                       *                 *                                                              *       ***                      ***
   8A VAL   7      1.351    1.236    1.512    1.538    1.460   118.73   121.00   122.26   111.26   107.73   113.55   116.74
                      +*                                            +*       **                         *         *     +***            +***
   9A VAL   8      1.343    1.227    1.518    1.535    1.449   123.80   116.51   120.88   109.70   110.44   112.78   122.62
                                                                     *                                                                     *
  10A TRP   9      1.350    1.237    1.531    1.563    1.457   123.42   117.58   123.88   109.84   111.22   113.38   118.54
                      +*                         +*                                   +*                         +*      +**             +**
  11A LEU  10      1.343    1.227    1.504    1.571    1.456   119.46   115.24   123.04   118.53   110.95   116.65   121.67
                       *                         **                  *                 *     ****               +***                    ****
  12A TRP  11      1.340    1.232    1.532    1.557    1.438   120.93   113.31   126.77   112.66   112.53   111.31   119.84
                                                  *        *                  *     +***        *                         +*            +***
  13A LEU  12      1.328    1.231    1.533    1.543    1.428   126.06   116.99   121.29   115.70   105.79    99.54   121.67
                                                          +*        **                        +**       +*    *6.4*                     *6.4*
  14A TRP  13      1.340    1.231    1.529    1.560    1.450   121.28   115.47   123.67   110.55   111.27   112.20   120.68
                                                 +*                                   +*                          *        *              +*
  15A LEU  14      1.335    1.230    1.545    1.560    1.437   130.38   117.11   121.61   117.82   108.31    97.05   121.23
                                                 +*        *     *4.8*                       ****        *    *7.9*        *            *7.9*
  16A TRP  15      1.354    1.229    1.534    1.551    1.469   122.98   115.83   123.70   108.07   114.75   112.71   120.42
                      +*                          *                                   +*        *        *        *       +*              +*
  17A ETA  15A     1.348        -        -    1.534    1.441   123.22        -        -        -        -   113.62        -
                       *                                                                                         +*                       +*
-----------------------------------------------------------------------------------------------------------------------------------------
Max deviations:      +**                +*      +**       +*    *4.8*       **     +***     ****       +*    *7.9*     +***            *7.9*
-----------------------------------------------------------------------------------------------------------------------------------------


              A N A L Y S I S   O F   M A I N   C H A I N   B O N D   L E N G T H S   A N D   B O N D   A N G L E S

                                                  +-------------------+
                                                  |   BOND LENGTHS    |
                                                  +-------------------+
-----------------------------------------------------------------------------------------------------------------
                                              (Small molecule data)  Number of  Min      Max        Mean    Standard
Bond      X-PLOR labelling                         Mean    St. dev     values   value    value      value   deviation
-----------------------------------------------------------------------------------------------------------------
C-N       C-NH1           (except Pro)            1.329    0.014           16   1.328    1.365      1.348     0.010
                                                                                           +**          *
          C-N             (Pro)                   1.341    0.016            0     -        -          -         -
C-O       C-O                                     1.231    0.020           16   1.220    1.237      1.229     0.004
CA-C      CH1E-C          (except Gly)            1.525    0.021           14   1.490    1.545      1.516     0.017
                                                                                  +*
          CH2G*-C         (Gly)                   1.516    0.018            1   1.485    1.485      1.485     0.000
                                                                                  +*       +*         +*
CA-CB     CH1E-CH3E       (Ala)                   1.521    0.033            2   1.543    1.548      1.546     0.003
          CH1E-CH1E       (Ile,Thr,Val)           1.540    0.027            4   1.525    1.548      1.537     0.008
          CH1E-CH2E       (the rest)              1.530    0.020            9   1.534    1.585      1.558     0.014
                                                                                           +**          *
N-CA      NH1-CH1E        (except Gly,Pro)1.458    0.019           15   1.428    1.495      1.457     0.017
                                                                                  +*       +*
          NH1-CH2G*       (Gly)                   1.451    0.016            1   1.466    1.466      1.466     0.000
          N-CH1E          (Pro)                   1.466    0.015            0     -        -          -         -
-----------------------------------------------------------------------------------------------------------------
```

```
                                          +-----------------+
                                          |  BOND ANGLES    |
                                          +-----------------+
---------------------------------------------------------------------------------------------------------
                                          (Small molecule data)  Number of  Min      Max        Mean    Standard
Angle     X-PLOR labelling                    Mean   St. dev      values   value    value      value   deviation
---------------------------------------------------------------------------------------------------------
CA-C-N    CH1E-C-NH1         (except Gly,Pro)116.2      2.0          14    113.31   121.00     117.05      1.80
                                                                               *       **
          CH2G*-C-NH1        (Gly)           116.4      2.1           1    120.70   120.70     120.70      0.00
                                                                              **       **         **
          CH1E-C-N           (Pro)           116.9      1.5           0      -        -          -          -
O-C-N     O-C-NH1            (except Pro)    123.0      1.6          16    116.74   126.15     121.63      2.23
                                                                            +***       +*
          O-C-N              (Pro)           122.0      1.4           0      -        -          -          -
C-N-CA    C-NH1-CH1E         (except Gly,Pro)121.7      1.8          15    118.73   130.38     122.12      2.97
                                                                              +*     *4.8*
          C-NH1-CH2G*        (Gly)           120.6      1.7           1    117.37   117.37     117.37      0.00
                                                                              +*       +*         +*
          C-N-CH1E           (Pro)           122.6      5.0           0      -        -          -          -
CA-C-O    CH1E-C-O           (except Gly)    120.8      1.7          14    117.38   126.77     121.55      2.45
                                                                              **     +***
          CH2G*-C-O          (Gly)           120.8      2.1           1    118.25   118.25     118.25      0.00
                                                                               *        *          *
CB-CA-C   CH3E-CH1E-C        (Ala)           110.5      1.5           2    105.63   107.69     106.66      1.03
                                                                             ***       +*        +**
          CH1E-CH1E-C        (Ile,Thr,Val)   109.1      2.2           4    106.58   111.26     109.08      1.69
                                                                               *
          CH2E-CH1E-C        (the rest)      110.1      1.9           8    108.07   118.53     112.72      3.88
                                                                               *     ****          *
N-CA-C    NH1-CH1E-C         (except Gly,Pro)111.2      2.8          14    105.79   114.75     109.91      2.34
                                                                              +*        *
          NH1-CH2G*-C        (Gly)           112.5      2.9           1    113.52   113.52     113.52      0.00
          N-CH1E-C           (Pro)           111.8      2.5           0      -        -          -          -
N-CA-CB   NH1-CH1E-CH3E      (Ala)           110.4      1.5           2    113.75   114.28     114.01      0.26
                                                                              **      +**         **
          NH1-CH1E-CH1E      (Ile,Thr,Val)   111.5      1.7           4    112.78   117.01     114.46      1.59
                                                                                      ***         +*
          N-CH1E-CH2E        (Pro)           103.0      1.1           0      -        -          -          -
          NH1-CH1E-CH2E      (the rest)      110.5      1.7           9     97.05   118.26     110.52      6.87
                                                                            *7.9*    *4.6*
---------------------------------------------------------------------------------------------------------
```

The small molecule data used in the above analysis is from Engh & Huber (1991). The atom labelling follows that used in the X-PLOR dictionary, with some additional atoms (marked with an asterisk) as defined by Engh & Huber.

```
                    B A D   C O N T A C T S   L I S T I N G

             ...................................................................
              Residue              Residue
             ----------           ----------
             No.   Type           No.   Type                    Contact      Distance
                 Chain     Atom       Chain       Atom           type       (Angstroms)
             -------------------------------------------------------------------
      1.      15  A TRP     C   -->   15A A ETA    N            Main-Het          1.3

                  R A M A C H A N D R A N   P L O T   S T A T I S T I C S
          Residues in most favoured regions       [A,B,L]             6       50.0%
          Residues in additional allowed regions  [a,b,l,p]           0        0.0%
          Residues in generously allowed regions  [~a,~b,~l,~p]       0        0.0%
          Residues in disallowed regions          [XX]                6       50.0%
                                                                    ----     ------
          Number of non-glycine and non-proline residues             12      100.0%

          Number of end-residues (excl. Gly and Pro)                  2

          Number of glycine residues                                  1
          Number of proline residues                                  0
                                                                    ----
          Total number of residues                                   15
```

Based on the analysis of 118 structures of resolution of at least 2.0 Angstroms and R-factor no greater than 20%, a good quality model would be expected to have over 90% in the most favoured regions [E,H,L].

```
                  S T E R E O C H E M I S T R Y   O F   M A I N - C H A I N
                                                       Comparison values    No. of
                                        No. of   Parameter  Typical   Band    band widths
      Stereochemical parameter         data pts    value     value   width    from mean
      ------------------------         --------    -----     -----   -----    ---------
   a. %-tage residues in A, B, L          12        50.0      89.7    10.0       -4.0    WORSE
   b. Omega angle st dev                  14         7.1       6.0     3.0        0.4    Inside
   c. Bad contacts / 100 residues          1         6.7       1.0    10.0        0.6    Inside
   d. Zeta angle st dev                   14        33.9       3.1     1.6       19.3    WORSE
   e. H-bond energy st dev                 9         0.5       0.6     0.2       -0.2    Inside
   f. Overall G-factor                    15        -1.4       0.0     0.3       -4.7    WORSE

                  S T E R E O C H E M I S T R Y   O F   S I D E - C H A I N
                                                       Comparison values    No. of
                                        No. of   Parameter  Typical   Band    band widths
      Stereochemical parameter         data pts    value     value   width    from mean
      ------------------------         --------    -----     -----   -----    ---------
   a. Chi-1 gauche minus st dev            2         1.1       9.0     6.5       -1.2    BETTER
   b. Chi-1 trans st dev                   5        13.6      11.6     5.3        0.4    Inside
   c. Chi-1 gauche plus st dev             5         5.9      10.0     4.9       -0.8    Inside
   d. Chi-1 pooled st dev                 12        10.3      10.5     4.8        0.0    Inside
   e. Chi-2 trans st dev                   1         0.0      15.0     5.0       -3.0    BETTER

                  M O R R I S   E T   A L .   C L A S S I F I C A T I O N
                           Mean  St.dev                  Classification
   Parameter                m      s          1        2         3         4        Value     Class
   ---------               ----   ---     ---------------------------------------   -----     -----
   Phi-psi distribution     -      -      >75.0%   >65.0%    >55.0%    <55.0%       50.0        4
   Chi-1 st.dev.          18.2    6.2     <12.0    <18.2     <24.4     >24.4         6.2        1
   H-bond energy st dev    0.87   0.24    <0.63    <0.87     <1.11     >1.11         0.61       1
```

```
                          G - F A C T O R S
                                                             Average
Parameter                                        Score        Score
---------                                        -----        -----
Dihedral angles:-
     Phi-psi distribution                        -2.42
     Chi1-chi2 distribution                      -1.42
     Chi1 only                                   -1.41
     Chi3 & chi4                                  0.00
     Omega                                       -1.12
                                                ------
                                                              -1.65
                                                              =====
Main-chain covalent forces:-
     Main-chain bond lengths                     -0.10
     Main-chain bond angles                      -1.90
                                                ------
                                                              -1.14
                                                              =====


OVERALL AVERAGE                                               -1.40
                                                              =====

Ideally, scores should be above -0.5. Values below -1.0 may need investigation.
```

A.4 Program Source Code

A.4.1 CNFCS

CNFCS (CoNFormation with reduction by Chemical Shift) is used to calculate the (ϕ, ψ) torsion angles using the ^{15}N-^{1}H and ^{15}N-$^{13}C_1$ dipolar splittings, using the ^{15}N chemical shifts as filters.

```
/*
program CNFCS (CoNFormation with reduction by Chemical Shift)

Randal R. Ketchem
Quike Teng
Institute of Molecular Biophysics   904.644.1309 (voice)
Florida State University            904.644.1366 (FAX)
Tallahassee, FL 32306-3015          rrk@magnet.fsu.edu (email)
*/

/*
This program calculates (phi, psi) torsion angles from 15N-1H and 15N-13C
dipolar splittings. The possible bond components are reduced by
calculating the 15N chemical shift for each bond component pair for each
peptide plane and comparing the calculated chemical shift with the
observed chemical shift. The surviving bond components are then used to
calculate (phi, psi) torsion angles for the dipeptide.
*/

/*
Test data
21.8
0.67
37.0
63.0
206.0
198.0
0.0
104.0
17.2
0.82
35.0
64.0
201.0
145.0
0.0
105.0
*/

#include <stdio.h>
#include <math.h>

#define  X          0        /* Used as array element */
#define  Y          1        /* Used as array element */
#define  Z          2        /* Used as array element */
#define  PLANE1     0        /* Used as array element */
#define  PLANE2     1        /* Used as array element */
#define  HV         11.335   /* principle component for N-H */
#define  CV         1.271    /* principle component for N-C */
```

```
#define  DEG_PER_RAD 57.2958  /* Degrees per radians */
#define  COMP1       0        /* Used as array element: component one */
#define  COMP2       1        /* Used as array element: component two */
#define  COMP3       2        /* Used as array element: component three */
#define  COMP4       3        /* Used as array element: component four */
#define  PHI         0        /* Used as array element: Phi */
#define  PSI         1        /* Used as array element: Psi */
#define  BDEG        70.0     /* angle between l' and k */
#define  B1DEG       63.0     /* hnk bond angle */
#define  B2DEG       59.0     /* cnk bond angle */
#define  B1PRIMEDEG  57.0     /* angle between h' and l' bonds */
#define  B2PRIMEDEG  65.0     /* angle between c' and l' bonds */
#define  NCL         70.0     /* NCL bond angle */
#define  HNC         122.0    /* HNC bond angle */
#define  PPM_RANGE   20.0     /* PPM range used in comparing calc to obs */
#define  LN_LNGTH    256      /* Line length */

#define SIGN(a)   ((a) < 0.0 ? -1 : 1)

void enter_data(float hnu[2],
                float cnu[2],
                float sig1[4],
                float sig2[4],
                float alphad[2],
                float betad[2],
                char fname_in[])
{
   FILE  *workfile;

   if((workfile = fopen(fname_in, "r")) == NULL)
   {
      fprintf(stderr, "\n\t***File '%s' not found***\n", fname_in);
      fprintf(stderr, "\nThe file '%s' should be in the form:\n", fname_in);
      fprintf(stderr, "\n\t***DO NOT USE THE LABELS***");
      fprintf(stderr, "\n\t***USE ONLY THE NUMBERS!***\n");
      fprintf(stderr, "\nhnu\t\txxx.xxx");
      fprintf(stderr, "\ncnu\t\txxx.xxx");
      fprintf(stderr, "\nsigma11\t\txxx.xxx");
      fprintf(stderr, "\nsigma22\t\txxx.xxx");
      fprintf(stderr, "\nsigma33\t\txxx.xxx");
      fprintf(stderr, "\nsigma obs\txxx.xxx");
      fprintf(stderr, "\nalphad\t\txxx.xxx");
      fprintf(stderr, "\nbetad\t\txxx.xxx");
      fprintf(stderr, "\nhnu\t\txxx.xxx");
      fprintf(stderr, "\ncnu\t\txxx.xxx");
      fprintf(stderr, "\nsigma11\t\txxx.xxx");
      fprintf(stderr, "\nsigma22\t\txxx.xxx");
      fprintf(stderr, "\nsigma33\t\txxx.xxx");
      fprintf(stderr, "\nsigma obs\txxx.xxx");
      fprintf(stderr, "\nalphad\t\txxx.xxx");
      fprintf(stderr, "\nbetad\t\txxx.xxx\n\n");
      exit(1);
   }

   fscanf(workfile, "%f", &hnu[0]);
   fscanf(workfile, "%f", &cnu[0]);
   fscanf(workfile, "%f", &sig1[0]);
   fscanf(workfile, "%f", &sig1[1]);
   fscanf(workfile, "%f", &sig1[2]);
   fscanf(workfile, "%f", &sig1[3]);
   fscanf(workfile, "%f", &alphad[0]);
   fscanf(workfile, "%f", &betad[0]);
   fscanf(workfile, "%f", &hnu[1]);
   fscanf(workfile, "%f", &cnu[1]);
   fscanf(workfile, "%f", &sig2[0]);
   fscanf(workfile, "%f", &sig2[1]);
   fscanf(workfile, "%f", &sig2[2]);
   fscanf(workfile, "%f", &sig2[3]);
   fscanf(workfile, "%f", &alphad[1]);
   fscanf(workfile, "%f", &betad[1]);
}

void gnrcomp(float split,
             float bond[4],
             float parallel,
             int count[4],
             int compnum)
{
   bond[COMP1] = sqrt((1.0 + (split/parallel))/3.0);
   bond[COMP2] = -bond[COMP1];
   if ((split/parallel) < 1.0)
   {
      bond[COMP3] = sqrt((1.0 - (split/parallel))/3.0);
      bond[COMP4] = -bond[COMP3];
      count[compnum] = 4;
   }
   else
      count[compnum] = 2;
}

void simulat(float hnu[2],
             float cnu[2],
             int totl[4],
             float h1bond[4],
             float c1bond[4],
             float h2bond[4],
             float c2bond[4])
{
   gnrcomp(hnu[PLANE1], h1bond, HV, totl, COMP1);
   gnrcomp(cnu[PLANE1], c1bond, CV, totl, COMP2);
   gnrcomp(hnu[PLANE2], h2bond, HV, totl, COMP3);
   gnrcomp(cnu[PLANE2], c2bond, CV, totl, COMP4);
}

void write_head(float hnu[2],
                float cnu[2],
                float sig1[4],
                float sig2[4],
                float alphad[2],
                float betad[2],
                int totl[4],
                float h1bond[4],
                float c1bond[4],
                float h2bond[4],
                float c2bond[4],
                char fname_out[LN_LNGTH])
{
   FILE  *workfile;
   int   count;

   if ((workfiie = fopen(fname_out, "w")) == NULL)
   {
      printf("Could not open %s.\nData not saved.\n", fname_out);
      exit(1);
   }

   fprintf(workfile, "\n\t\t\t%s", fname_out);

   fprintf(workfile, "\n\n\n\t\t\tFirst Plane");
   fprintf(workfile, "\n\n\tNH\t\tNC");
   fprintf(workfile, "\n\t%6.3f\t\t%6.3f", hnu[PLANE1], cnu[PLANE1]);
   fprintf(workfile, "\n\n\tSigma11\t\tSigma22\t\tSigma33\n");
   fprintf(workfile, "\t%7.2f\t\t%7.2f\t\t%7.2f",
                     sig1[0], sig1[1], sig1[2]);
   fprintf(workfile, "\n\n\tSigma Obs\talphad\t\tbetad\n");
   fprintf(workfile, "\t%7.2f\t\t%7.2f\t\t%7.2f",
                     sig1[3], alphad[0], betad[0]);

   fprintf(workfile, "\n\n\n\t\t\tSecond Plane");
   fprintf(workfile, "\n\n\tNH\t\tNC");
   fprintf(workfile, "\n\t%6.3f\t\t%6.3f", hnu[PLANE2], cnu[PLANE2]);
   fprintf(workfile, "\n\n\tSigma11\t\tSigma22\t\tSigma33\n");
   fprintf(workfile, "\t%7.2f\t\t%7.2f\t\t%7.2f",
                     sig2[0], sig2[1], sig2[2]);
   fprintf(workfile, "\n\n\tSigma Obs\talphad\t\tbetad\n");
   fprintf(workfile, "\t%7.2f\t\t%7.2f\t\t%7.2f",
                     sig2[3], alphad[1], betad[1]);

   fprintf(workfile, "\n\n\n\tH1 Components\n");
   for (count = 0; count < totl[COMP1]; ++count)
      fprintf(workfile, "\n\t%8.4f", h1bond[count]);

   fprintf(workfile, "\n\n\n\tC1 Components\n");
   for (count = 0; count < totl[COMP2]; ++count)
      fprintf(workfile, "\n\t%8.4f", c1bond[count]);

   fprintf(workfile, "\n\n\n\tH2 Components\n");
   for (count = 0; count < totl[COMP3]; ++count)
      fprintf(workfile, "\n\t%8.4f", h2bond[count]);

   fprintf(workfile, "\n\n\n\tC2 Components\n");
   for (count = 0; count < totl[COMP4]; ++count)
      fprintf(workfile, "\n\t%8.4f", c2bond[count]);

   fprintf(workfile, "\n");

   fclose(workfile);
}

void convert_data(float alphad[2],
                  float betad[2])
{
   alphad[0] = alphad[0] / DEG_PER_RAD;
   alphad[1] = alphad[1] / DEG_PER_RAD;
   betad[0] = betad[0] / DEG_PER_RAD;
   betad[1] = betad[1] / DEG_PER_RAD;
}

void calculate_nh(float nh[3],
                  float hbond,
                  float cbond)
{
   float angleh, anglec, nhvector;

   angleh = acos(hbond);
   anglec = acos(cbond);
   nh[Z] = cos(angleh);
   nh[Y] = cos(angleh) * cos(anglec);
   nh[Y] = cos(HNC / DEG_PER_RAD) - nh[Y];
   nh[Y] = nh[Y] / sin(anglec);
   nh[X] = sqrt(1 - pow(nh[Y], 2) - pow(nh[Z] ,2));

   nhvector = sqrt(pow(nh[X], 2) + pow(nh[Y], 2) + pow(nh[Z], 2));

   if (nhvector == 0.0)
      printf("\n\nnh == 0. No normalization.\n");
   else
   {
      nh[X] = nh[X] / nhvector;
      nh[Y] = nh[Y] / nhvector;
      nh[Z] = nh[Z] / nhvector;
   }
}

void calculate_nc(float nc[3],
                  float cbond)
{
   float angle, ncvector;

   angle = acos(cbond);

   nc[X] = 0.0;
   nc[Y] = sin(angle);
   nc[Z] = cos(angle);

   ncvector = sqrt(pow(nc[X], 2) + pow(nc[Y], 2) + pow(nc[Z], 2));

   if (ncvector == 0.0)
      printf("\n\nnc == 0. No normalization.\n");
   else
   {
      nc[X] = nc[X] / ncvector;
      nc[Y] = nc[Y] / ncvector;
      nc[Z] = nc[Z] / ncvector;
   }
}

void calculate_theta(float hbond,
                     float cbond,
                     float theta[3])
{
   float nc[3], nh[3];
   float ncvec, nhvec;
   float y1, y2, y3;
   float yprime;
   float xci;

   calculate_nh(nh, hbond, cbond);
   calculate_nc(nc, cbond);

   ncvec = sqrt(pow(nc[X], 2) + pow(nc[Y], 2) + pow(nc[Z], 2));
   nhvec = sqrt(pow(nh[X], 2) + pow(nh[Y], 2) + pow(nh[Z], 2));

   theta[0] = acos(((nc[X]*nh[X]) + (nc[Y]*nh[Y]) + (nc[Z]*nh[Z])) /
              (ncvec*nhvec));
   xci = acos(nc[Y]/sqrt(pow(nc[X],2) + pow(nc[Y], 2))) * SIGN(nc[X]);
   theta[2] = acos(nc[Z]);
   y1 = (((nc[Y] * nh[Z]) - (nc[Z] * nh[Y])) / sin(theta[0]));
   y2 = (((nc[Z] * nh[X]) - (nc[X] * nh[Z])) / sin(theta[0]));
   y3 = (((nc[X] * nh[Y]) - (nc[Y] * nh[X])) / sin(theta[0]));
   yprime = (y1 * cos(xci)) - (y2 * sin(xci));

   theta[1] = acos(yprime) * SIGN(y3);
}

void transpose(float matrix1[3][3],
               float matrix2[3][3])
{
   int          row, col;

   for (row = 0; row < 3; ++row)
      for (col = 0; col < 3; ++col)
         matrix2[col][row] = matrix1[row][col];
}

void matrix_multiply(float matrix1[3][3],
                     float matrix2[3][3],
                     float result[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
         result[i][j] =  matrix1[i][0]*matrix2[0][j]
                       + matrix1[i][1]*matrix2[1][j]
                       + matrix1[i][2]*matrix2[2][j];
}

void rotate_pas_to_mf(float sigma_mf[3][3],
                      float sigma[4],
                      float alphad,
                      float betad)
{
   float        rbd[3][3];              /* R(betaD) */
   float        rad[3][3];              /* R(alphaD) */
   float        radt[3][3];             /* R(alphaD)Transpose */
   float        rbdt[3][3];             /* R(betaD)Transpose */
   float        sigma_pas[3][3];        /* principle axis system */
   float        result[3][3];           /* matrix used in calculations */
   float        result1[3][3];          /* matrix used in calculations */
   float        result2[3][3];          /* matrix used in calculations */

   rbd[0][0] = cos(betad);   /* R(betaD) matrix, [row][col] */
   rbd[0][1] = 0.0;
   rbd[0][2] = sin(betad);
   rbd[1][0] = 0.0;
   rbd[1][1] = 1.0;
   rbd[1][2] = 0.0;
   rbd[2][0] = -sin(betad);
   rbd[2][1] = 0.0;
   rbd[2][2] = cos(betad);

   transpose(rbd, rbdt);

   rad[0][0] = cos(alphad);  /* R(alphaD) matrix, [row][col] */
   rad[0][1] = sin(alphad);
   rad[0][2] = 0.0;
   rad[1][0] = -sin(alphad);
   rad[1][1] = cos(alphad);
```

```
    rad[1][2] = 0.0;
    rad[2][0] = 0.0;
    rad[2][1] = 0.0;
    rad[2][2] = 1.0;

    transpose(rad, radt);

    sigma_pas[0][0] = sigma[0];/* sigma_pas matrix, [row][col] */
    sigma_pas[0][1] = 0.0;
    sigma_pas[0][2] = 0.0;
    sigma_pas[1][0] = 0.0;
    sigma_pas[1][1] = sigma[1];
    sigma_pas[1][2] = 0.0;
    sigma_pas[2][0] = 0.0;
    sigma_pas[2][1] = 0.0;
    sigma_pas[2][2] = sigma[2];

    matrix_multiply(rbd, rad, result1);
    matrix_multiply(result1, sigma_pas, result2);
    matrix_multiply(result2, radt, result);
    matrix_multiply(result, rbdt, sigma_mf);
}

void rotate_mf_to_lab(float sigma_mf[3][3],
                      float theta[3],
                      float sigma_lab[3][3])
{
    float       rt3[3][3];      /* R(theta3) */
    float       rt2[3][3];      /* R(theta2) */
    float       rt2t[3][3];     /* R(theta2)T */
    float       rt3t[3][3];     /* R(theta3)T */
    float       result[3][3];   /* used during matrix multiplication */
    float       result1[3][3];  /* used during matrix multiplication */
    float       result2[3][3];  /* used during matrix multiplication */

    rt3[0][0] = cos(theta[2]);  /* R(theta3) matrix, [row][col] */
    rt3[0][1] = 0.0;
    rt3[0][2] = -sin(theta[2]);
    rt3[1][0] = 0.0;
    rt3[1][1] = 1.0;
    rt3[1][2] = 0.0;
    rt3[2][0] = sin(theta[2]);
    rt3[2][1] = 0.0;
    rt3[2][2] = cos(theta[2]);

    transpose(rt3, rt3t);

    rt2[0][0] = cos(theta[1]);  /* R(theta2) matrix, [row][col] */
    rt2[0][1] = sin(theta[1]);
    rt2[0][2] = 0.0;
    rt2[1][0] = -sin(theta[1]);
    rt2[1][1] = cos(theta[1]);
    rt2[1][2] = 0.0;
    rt2[2][0] = 0.0;
    rt2[2][1] = 0.0;
    rt2[2][2] = 1.0;

    transpose(rt2, rt2t);

    matrix_multiply(rt3, rt2, result1);
    matrix_multiply(result1, sigma_mf, result2);
    matrix_multiply(result2, rt2t, result);
```

```
   matrix_multiply(result, rt3t, sigma_lab);
}

int cscomput(float sigma[4],
             float alphad,
             float betad,
             float hbond,
             float cbond)
{
   float calc_cs;
   float theta[3];
   float sigma_mf[3][3];
   float sigma_lab[3][3];

   calculate_theta(hbond, cbond, theta);
   rotate_pas_to_mf(sigma_mf, sigma, alphad, betad);
   rotate_mf_to_lab(sigma_mf, theta, sigma_lab);

   calc_cs = sigma_lab[2][2];

   if((calc_cs < (sigma[3] + PPM_RANGE)) &&
      (calc_cs > (sigma[3] - PPM_RANGE)))
   {
      fprintf(stdout, "cs = %10.4f (%.4f obs)\n", calc_cs, sigma[3]);
      return(1);
   }
   else
      return(0);
}

void chemshift(int numok[2],
               int tot1[4],
               float plane1[2][16],
               float plane2[2][16],
               float sig1[4],
               float sig2[4],
               float alphad[2],
               float betad[2],
               float h1bond[4],
               float c1bond[4],
               float h2bond[4],
               float c2bond[4],
               char fname_out[LN_LNGTH])
{
   int   count1, count2, count, cs;
   FILE  *workfile;

   numok[0] = 0;
   for (count1 = 0; count1 < tot1[0]; ++count1)
      for (count2 = 0; count2 < tot1[1]; ++count2)
      {
         cs = cscomput(sig1, alphad[0], betad[0], h1bond[count1],
                       c1bond[count2]);
         if(cs)
         {
            plane1[0][numok[0]] = h1bond[count1];
            plane1[1][numok[0]] = c1bond[count2];
            ++numok[0];
         }
      }

   numok[1] = 0;
   for (count1 = 0; count1 < totl[2]; ++count1)
      for (count2 = 0; count2 < totl[3]; ++count2)
      {
         cs = cscomput(sig2, alphad[1], betad[1], h2bond[count1],
                     c2bond[count2]);
         if(cs)
         {
            plane2[0][numok[1]] = h2bond[count1];
            plane2[1][numok[1]] = c2bond[count2];
            ++numok[1];
         }
      }

   if ((workfile = fopen(fname_out, "a")) == NULL)
   {
      printf("Could not open %s.\nData not saved.\n", fname_out);
      exit(1);
   }

   fprintf(workfile, "\n\n\t\tFirst Plane Combination After CS\n");
   for (count = 0; count < numok[0]; ++count)
      fprintf(workfile, "\n\t%3i\t%8.4f\t%8.4f",
            count+1, plane1[0][count], plane1[1][count]);

   fprintf(workfile, "\n\n\n\t\tSecond Plane Combination After CS\n");
   for (count = 0; count < numok[1]; ++count)
      fprintf(workfile, "\n\t%3i\t%8.4f\t%8.4f",
            count+1, plane2[0][count], plane2[1][count]);

   fprintf(workfile, "\n");

   fclose(workfile);
}

float zcomput(float bond1,
              float bond2,
              float beta1,
              float beta2)
{
   float beta, result;

   beta = beta1 + beta2;
   result = (sin(beta2) * bond1 + sin(beta1) * bond2) / sin(beta);
   return(result);
}

void link(float first[2][16],
          float second[2][16],
          int numplane[2],
          int *numdip,
          float quartet[4][32],
          float lcomp[2][32],
          char fname_out[LN_LNGTH])
{
   float b1, b2, b1prime, b2prime, evalpls, evalmns;
   float lambda1, lambda2;
   int   count1, count2, count;
   FILE  *workfile;

   b1 = B1DEG / DEG_PER_RAD;
   b2 = B2DEG / DEG_PER_RAD;
   b1prime = B1PRIMEDEG / DEG_PER_RAD;
   b2prime = B2PRIMEDEG / DEG_PER_RAD;
   evalpls = 1.0 / (1.0 + cos(NCL / DEG_PER_RAD));
   evalmns = 1.0 / (1.0 - cos(NCL / DEG_PER_RAD));

   if ((workfile = fopen(fname_out, "a")) == NULL)
   {
      printf("Could not open %s.\nData not saved.\n", fname_out);
      exit(1);
   }

   fprintf(workfile, "\n\n\t\tResults For Dipeptide Combinations");
   fprintf(workfile, "\n\n\tH1\t\tC1\t\tH2\t\tC2\n");


   *numdip = 0;
   for (count1 = 0; count1 < numplane[0]; ++count1)
   {
      lambda1 = zcomput(first[0][count1], first[1][count1], b1, b2);
      for (count2 = 0; count2 < numplane[1]; ++count2)
      {
         lambda2 = zcomput(second[0][count2], second[1][count2],
                           b1prime, b2prime);
         quartet[0][*numdip] = first[0][count1];
         quartet[1][*numdip] = first[1][count1];
         quartet[2][*numdip] = second[0][count2];
         quartet[3][*numdip] = second[1][count2];
         lcomp[0][*numdip] = lambda1;
         lcomp[1][*numdip] = lambda2;
         fprintf(workfile, "\n%3i", *numdip+1);
         for (count = 0; count < 4; ++count)
            fprintf(workfile, "\t%8.4f", quartet[count][*numdip]);
         ++*numdip;
      }
   }
   fprintf(workfile, "\n\n");
   fclose(workfile);
}

void torcomp(float angle[4],
             float lcomp2,
             float lcomp1,
             float hcomp,
             int torsion,
             char fname_out[LN_LNGTH])
{
   float b, beta1, top1, bottoml, l;
   float toph, bottomh, hh, h;
   int   count;
   FILE  *workfile;

   b = BDEG / DEG_PER_RAD;
   switch (torsion)
   {
      case(PHI):
         beta1 = B1DEG/DEG_PER_RAD;
         break;
      case(PSI):
         beta1 = B1PRIMEDEG/DEG_PER_RAD;
         break;
   }

   topl = lcomp2 - lcomp1 * cos(b);
   bottoml = (sqrt(1.0 - pow(lcompl, 2))) * sin(b);
   l = acos(topl / bottoml);

   toph = hcomp - lcompl * cos(betal);
   bottomh = (sqrt(1.0 - pow(lcompl, 2))) * sin(betal);
   hh = toph / bottomh;
   if (torsion == PHI)
   {
      if (hh > 1.0)
         hh = 1.0;
      if (hh <= -1.0)
         hh = -1.0;
   }
   h = acos(hh);

   angle[0] = (l + h) * DEG_PER_RAD;
   angle[1] = -angle[0];
   angle[2] = (l - h) * DEG_PER_RAD;
   angle[3] = -angle[2];

   for (count = 0; count < 4; ++count)
   {
      if (angle[count] > 180.0)
         angle[count] = angle[count] - 360.0;
      if (angle[count] < -180.0)
         angle[count] = angle[count] + 360.0;
   }

   if ((workfile = fopen(fname_out, "a")) == NULL)
   {
      printf("Could not open %s.\nData not saved.\n", fname_out);
      exit(1);
   }

   switch(torsion)
   {
/* All possible solutions */
      case(PHI):
         fprintf(workfile, "\n");
         fprintf(workfile, "\t%7.2f", angle[0]);
         fprintf(workfile, "\t%7.2f", angle[0]);
         fprintf(workfile, "\t%7.2f", angle[1]);
         fprintf(workfile, "\t%7.2f", angle[1]);
         fprintf(workfile, "\t%7.2f", angle[2]);
         fprintf(workfile, "\t%7.2f", angle[2]);
         fprintf(workfile, "\t%7.2f", angle[3]);
         fprintf(workfile, "\t%7.2f", angle[3]);
         break;
      case(PSI):
         fprintf(workfile, "\n");
         fprintf(workfile, "\t%7.2f", angle[0]);
         fprintf(workfile, "\t%7.2f", angle[2]);
         fprintf(workfile, "\t%7.2f", angle[1]);
         fprintf(workfile, "\t%7.2f", angle[3]);
         fprintf(workfile, "\t%7.2f", angle[0]);
         fprintf(workfile, "\t%7.2f", angle[2]);
         fprintf(workfile, "\t%7.2f", angle[1]);
         fprintf(workfile, "\t%7.2f", angle[3]);
         fprintf(workfile, "\n\n");
   }

   fclose(workfile);
```

```
}

void main(int argc,
          char *argv[])
{
   float h1bond[4], c1bond[4], h2bond[4], c2bond[4];
   float hnu[2], cnu[2], alphad[2], betad[2], sig1[4], sig2[4];
   float pep1[2][16], pep2[2][16];
   float dipep[4][32], lbond[2][32], phiang[4], psiang[4];
   int   numplane[2], numdipep, oritot[4], count;
   char  fname_out[LN_LNGTH];
   FILE  *workfile;

   if (argc != 2)
   {
      fprintf(stderr, "Usage: %s filename\n", argv[0]);
      exit(2);
   }

   sprintf(fname_out, "%s_cnfcs", argv[1]);

   enter_data(hnu, cnu, sig1, sig2, alphad, betad, argv[1]);
   simulat(hnu, cnu, oritot, h1bond, c1bond, h2bond, c2bond);
   write_head(hnu, cnu, sig1, sig2, alphad, betad, oritot, h1bond, c1bond,
            h2bond, c2bond, fname_out);
   convert_data(alphad, betad);
   chemshift(numplane, oritot, pep1, pep2, sig1, sig2, alphad, betad,
            h1bond, c1bond, h2bond, c2bond, fname_out);
   link(pep1, pep2, numplane, &numdipep, dipep, lbond, fname_out);
   for (count = 0; count < numdipep; ++count)
   {
      if ((workfile = fopen(fname_out, "a")) == NULL)
      {
         printf("Could not open %s.\nData not saved.\n", fname_out);
         exit(1);
      }

      fprintf(workfile, "\nCombination # %3i", count+1);
      fprintf(workfile, "\n\nPair #\t 1\t 2\t 3\t 4\t 5\t 6\t 7\t 8");
      fclose(workfile);

      torcomp(phiang, lbond[1][count], lbond[0][count], dipep[0][count],
            PHI, fname_out);
      torcomp(psiang, lbond[0][count], lbond[1][count], dipep[2][count],
            PSI, fname_out);
   }
   printf("Output stored in '%s'\n", fname_out);
}
```

A.4.2 COORDS

COORDS (Calculate cOORDinateS) is used to calculate atomic coordinates for a diplane orientation and a single (ϕ, ψ) torsion angle pair, as output from CNFCS.

```
/* program coords.c */
/* to compile on UNIX machine: cc coords.c -lm -o coords */

/*
Randal R. Ketchem
Institute of Molecular Biophysics   904.644.1309 (voice)
Florida State University            904.644.1366 (FAX)
Tallahassee, FL 32306-3015          rrk@magnet.fsu.edu (email)
*/

/*
This program will take as input the direction cosines and (phi, psi)
torsion angles from the program cnfcs and will output the diplane atomic
coordinates as:

              oiml          hip1
              ||            |
              ||            |
              ciml   cai    nip1
              /\     /\     /\
             /  \   /  \   /  \
            /    \ /    \ /    \
           caiml  ni     ci     caip1
                  |      ||
                  |      ||
                  hi     oi
*/

#include <stdio.h>
#include <math.h>

#define CNH             122.0
#define X               0
#define Y               1
#define Z               2
#define PHI             0
#define PSI             1
#define NHTHETA1        0
#define NCTHETA1        1
#define NHTHETA2        2
#define NCTHETA2        3
#define RAD_PER_DEGREE  0.017453293

#define SIGN(a)    ((a) < 0.0 ? -1 : 1)

float caiml_ciml[3];  /* vector from caiml to ciml */
float ciml_oiml[3];   /* vector from ciml to oiml */
float ciml_ni[3];     /* vector from ciml to ni */
float ni_hi[3];       /* vector from ni to hi */
float ni_cai[3];      /* vector from ni to cai */
float cai_ci[3];      /* vector from cai to ci */
float ci_oi[3];       /* vector from ci to oi */
float ci_nip1[3];     /* vector from ci to nip1 */
float nip1_hip1[3];   /* vector from nip1 to hip1 */
float nip1_caip1[3];  /* vector from nip1 to caip1 */

void input_data(float theta[4],
                float torsion[2])
{
   fprintf(stdout, "Enter direction cosines (from cnfcs):\n");
   scanf("%f %f %f %f", &theta[NHTHETA1], &theta[NCTHETA1],
                        &theta[NHTHETA2], &theta[NCTHETA2]);
   theta[NHTHETA1] = acos(theta[NHTHETA1]);
   theta[NCTHETA1] = acos(theta[NCTHETA1]);
   theta[NHTHETA2] = acos(theta[NHTHETA2]);
   theta[NCTHETA2] = acos(theta[NCTHETA2]);

   fprintf(stdout, "Enter Phi (degrees): ");
   scanf("%f", &torsion[PHI]);
   torsion[PHI] = torsion[PHI] * RAD_PER_DEGREE;

   fprintf(stdout, "Enter Psi (degrees): ");
   scanf("%f", &torsion[PSI]);
   torsion[PSI] = torsion[PSI] * RAD_PER_DEGREE;

   fprintf(stdout, "\n");
}

void vectorize(float atom1[3],
               float atom2[3],
               float vector[3])
{
   vector[X] = atom2[X] - atom1[X];
   vector[Y] = atom2[Y] - atom1[Y];
   vector[Z] = atom2[Z] - atom1[Z];
}

void define_plane(void)
{
        /* c alpha sub i minus 1 */
   static float caim1[3] = {1.386651, 0.000000, 1.986606};
        /* c sub i minus 1 */
   static float cim1[3]  = {0.000000, 0.000000, 1.340000};
        /* o sub i minus 1 */
   static float oim1[3]  = {-1.028007, 0.000000, 2.033399};
        /* n sub i */
   static float ni[3]    = {0.000000, 0.000000, 0.000000};
        /* h sub i */
   static float hi[3]    = {0.868401, 0.000000, -0.542637};
        /* c alpha sub i */
   static float cai[3]   = {-1.242893, 0.000000, -0.746805};
        /* c sub i */
   static float ci[3]    = {-0.950955, 0.000000, -2.248695};
        /* o sub i */
   static float oi[3]    = {0.214264, 0.000000, -2.672800};
        /* n sub i plus 1 */
   static float nip1[3]  = {-2.035038, 0.000000, -3.036327};
        /* h sub i plus 1 */
   static float hip1[3]  = {-2.984474, 0.000000, -2.652730};
        /* c alpha sub i plus 1 */
   static float caip1[3] = {-1.908662, 0.000000, -4.480809};

   vectorize(caim1, cim1, caim1_cim1);
   vectorize(ciml, oiml, ciml_oiml);
   vectorize(ciml, ni, ciml_ni);
   vectorize(ni, hi, ni_hi);
   vectorize(ni, cai, ni_cai);
   vectorize(cai, ci, cai_ci);
   vectorize(ci, oi, ci_oi);
   vectorize(ci, nip1, ci_nip1);
   vectorize(nip1, hip1, nip1_hip1);
   vectorize(nip1, caip1, nip1_caip1);
}

void calculate_nc(float nc[3],
                  float theta[4])
{
   float ncvector;

   nc[X] = 0.0;
   nc[Y] = sin(theta[NCTHETA1]);
   nc[Z] = cos(theta[NCTHETA1]);

   ncvector = sqrt(pow(nc[X], 2) + pow(nc[Y], 2) + pow(nc[Z], 2));

   if (ncvector == 0.0)
      fprintf(stderr, "\n\nnc == 0.  No normalization.\n");
   else
   {
      nc[X] = nc[X]/ncvector;
      nc[Y] = nc[Y]/ncvector;
      nc[Z] = nc[Z]/ncvector;
   }
}

void calculate_nh(float nh[3],
                  float theta[4],
                  int count)
{
   float nhvector;

   nh[Z] = cos(theta[NHTHETA1]);

   nh[Y] = cos(theta[NHTHETA1])*cos(theta[NCTHETA1]);
   nh[Y] = cos(CNH*RAD_PER_DEGREE)-nh[Y];
   nh[Y] = nh[Y]/sin(theta[NCTHETA1]);

   switch(count)
   {
      case 0:
         nh[X] = sqrt(1-pow(nh[Y], 2)-pow(nh[Z], 2));
         break;
      case 1:
         nh[X] = -sqrt(1-pow(nh[Y], 2)-pow(nh[Z], 2));
         break;
   }

   nhvector = sqrt(pow(nh[X], 2) + pow(nh[Y], 2) + pow(nh[Z], 2));

   if (nhvector == 0.0)
      fprintf(stderr, "\n\nnh == 0.  No normalization.\n");
   else
   {
      nh[X] = nh[X]/nhvector;
      nh[Y] = nh[Y]/nhvector;
      nh[Z] = nh[Z]/nhvector;
   }
}

void calculate_y(float y[3],
                 float nc[3],
                 float nh[3])
{
   y[X] = ((nc[Y]*nh[Z]) - (nc[Z]*nh[Y])) / sin(CNH*RAD_PER_DEGREE);
   y[Y] = ((nc[X]*nh[Z]) - (nc[Z]*nh[X])) / sin(CNH*RAD_PER_DEGREE);
   y[Z] = ((nc[X]*nh[Y]) - (nc[Y]*nh[X])) / sin(CNH*RAD_PER_DEGREE);
}

void matrix_multiply(float matrix1[3][3],
                     float matrix2[3],
                     float result[3])
{
   int   i;

   for (i = 0; i < 3; i++)
   {
      result[i] = matrix1[i][0]*matrix2[0]
                + matrix1[i][1]*matrix2[1]
                + matrix1[i][2]*matrix2[2];
   }
}

void rotate_mf_to_lab(float alpha,
                      float beta,
                      float vector[3])
{
   float alpha_matrix[3][3];
   float beta_matrix[3][3];
   float result[3];

   alpha_matrix[0][0] = cos(alpha);
   alpha_matrix[0][1] = sin(alpha);
   alpha_matrix[0][2] = 0.0;
   alpha_matrix[1][0] = -sin(alpha);
   alpha_matrix[1][1] = cos(alpha);
   alpha_matrix[1][2] = 0.0;
   alpha_matrix[2][0] = 0.0;
   alpha_matrix[2][1] = 0.0;
   alpha_matrix[2][2] = 1.0;

   beta_matrix[0][0] = cos(beta);
   beta_matrix[0][1] = 0.0;
   beta_matrix[0][2] = -sin(beta);
   beta_matrix[1][0] = 0.0;
   beta_matrix[1][1] = 1.0;
   beta_matrix[1][2] = 0.0;
   beta_matrix[2][0] = sin(beta);
   beta_matrix[2][1] = 0.0;
   beta_matrix[2][2] = cos(beta);

   matrix_multiply(alpha_matrix, vector, result);
   matrix_multiply(beta_matrix, result, vector);
}

void rotate_torsion(float about_bond[3],
                    float angle,
                    float bond[3])
```

```
{
   float result[3];
   float euler_matrix[3][3];
   float euler[3];
   float ebeta[4];
   float angleprime;
   float eulermag;
   int   count;

   angleprime = angle - (180.0 * RAD_PER_DEGREE);

   euler[X] = about_bond[X];
   euler[Y] = about_bond[Y];
   euler[Z] = about_bond[Z];
   eulermag = sqrt(pow(euler[X], 2) + pow(euler[Y], 2) + pow(euler[Z], 2));

   if (eulermag == 0.0)
      fprintf(stderr, "\n\neulermag == 0.  No normalization.\n");
   else
   {
      euler[X] = euler[X]/eulermag;
      euler[Y] = euler[Y]/eulermag;
      euler[Z] = euler[Z]/eulermag;
   }

   ebeta[0] = cos(angleprime/2.0);
   ebeta[1] = euler[X] * sin(angleprime/2.0);
   ebeta[2] = euler[Y] * sin(angleprime/2.0);
   ebeta[3] = euler[Z] * sin(angleprime/2.0);

   euler_matrix[0][0] =   pow(ebeta[0], 2) + pow(ebeta[1], 2)
                        - pow(ebeta[2], 2) - pow(ebeta[3], 2);
   euler_matrix[0][1] = 2.0*((ebeta[1]*ebeta[2]) - (ebeta[0]*ebeta[3]));
   euler_matrix[0][2] = 2.0*((ebeta[1]*ebeta[3]) + (ebeta[0]*ebeta[2]));
   euler_matrix[1][0] = 2.0*((ebeta[1]*ebeta[2]) + (ebeta[0]*ebeta[3]));
   euler_matrix[1][1] =   pow(ebeta[0], 2) - pow(ebeta[1], 2)
                        + pow(ebeta[2], 2) - pow(ebeta[3], 2);
   euler_matrix[1][2] = 2.0*((ebeta[2]*ebeta[3]) - (ebeta[0]*ebeta[1]));
   euler_matrix[2][0] = 2.0*((ebeta[1]*ebeta[3]) - (ebeta[0]*ebeta[2]));
   euler_matrix[2][1] = 2.0*((ebeta[2]*ebeta[3]) + (ebeta[0]*ebeta[1]));
   euler_matrix[2][2] =   pow(ebeta[0], 2) - pow(ebeta[1], 2)
                        - pow(ebeta[2], 2) + pow(ebeta[3], 2);

   matrix_multiply(euler_matrix, bond, result);

   for (count = 0; count < 3; ++count)
      bond[count] = result[count];
}

float angleize(float a[3],
               float b[3],
               float c[3])
{
   float vec1[3], vec2[3], temp;

   vectorize(a, b, vec1);
   vectorize(a, c, vec2);

   temp = (vec1[X]*vec2[X]) + (vec1[Y]*vec2[Y]) + (vec1[Z]*vec2[Z]);
   temp = temp / (sqrt(pow(vec1[X],2) + pow(vec1[Y],2) + pow(vec1[Z],2)));
   temp = temp / (sqrt(pow(vec2[X],2) + pow(vec2[Y],2) + pow(vec2[Z],2)));
   return(temp);
```

```
}

void generate_coords(int count,
                     float theta[4],
                     float torsion[2])
{
   float caim1[3];       /* c alpha sub i minus 1 */
   float cim1[3];        /* c sub i minus 1 */
   float oim1[3];        /* o sub i minus 1 */
   float ni[3];          /* n sub i */
   float hi[3];          /* h sub i */
   float cai[3];         /* c alpha sub i */
   float ci[3];          /* c sub i */
   float oi[3];          /* o sub i */
   float nip1[3];        /* n sub i plus 1 */
   float hip1[3];        /* h sub i plus 1 */
   float caip1[3];       /* c alpha sub i plus 1 */
   float resultnh, resultnc, zero[3], thetanh, thetanc;
   FILE  *workfile;
   float theta0, theta1, theta2, theta3, torsion0, torsion1;
   char  headerX[1], headerY[1], headerZ[1];
   char  name[256];

   sprintf(headerX, "X");
   sprintf(headerY, "Y");
   sprintf(headerZ, "Z");

   ni[X] = 0.0;
   ni[Y] = 0.0;
   ni[Z] = 0.0;

   cim1[X] = -cim1_ni[X] + ni[X];
   cim1[Y] = -cim1_ni[Y] + ni[Y];
   cim1[Z] = -cim1_ni[Z] + ni[Z];

   oim1[X] = cim1_oim1[X] + cim1[X];
   oim1[Y] = cim1_oim1[Y] + cim1[Y];
   oim1[Z] = cim1_oim1[Z] + cim1[Z];

   caim1[X] = -caim1_cim1[X] + cim1[X];
   caim1[Y] = -caim1_cim1[Y] + cim1[Y];
   caim1[Z] = -caim1_cim1[Z] + cim1[Z];

   hi[X] = ni_hi[X] + ni[X];
   hi[Y] = ni_hi[Y] + ni[Y];
   hi[Z] = ni_hi[Z] + ni[Z];

   cai[X] = ni_cai[X] + ni[X];
   cai[Y] = ni_cai[Y] + ni[Y];
   cai[Z] = ni_cai[Z] + ni[Z];

   ci[X] = cai_ci[X] + cai[X];
   ci[Y] = cai_ci[Y] + cai[Y];
   ci[Z] = cai_ci[Z] + cai[Z];

   oi[X] = ci_oi[X] + ci[X];
   oi[Y] = ci_oi[Y] + ci[Y];
   oi[Z] = ci_oi[Z] + ci[Z];

   nip1[X] = ci_nip1[X] + ci[X];
   nip1[Y] = ci_nip1[Y] + ci[Y];
   nip1[Z] = ci_nip1[Z] + ci[Z];
```

```
hip1[X] = nip1_hip1[X] + nip1[X];
hip1[Y] = nip1_hip1[Y] + nip1[Y];
hip1[Z] = nip1_hip1[Z] + nip1[Z];

caip1[X] = nip1_caip1[X] + nip1[X];
caip1[Y] = nip1_caip1[Y] + nip1[Y];
caip1[Z] = nip1_caip1[Z] + nip1[Z];

theta0 = cos(theta[NHTHETA1]);
theta1 = cos(theta[NCTHETA1]);
theta2 = cos(theta[NHTHETA2]);
theta3 = cos(theta[NCTHETA2]);
torsion0 = torsion[PHI] / RAD_PER_DEGREE;
torsion1 = torsion[PSI] / RAD_PER_DEGREE;

zero[X] = nip1[X];
zero[Y] = nip1[Y];
zero[Z] = 0.0;
resultnh = angleize(nip1, hip1, zero);
resultnh = resultnh * SIGN(resultnh);
resultnc = angleize(nip1, ci, zero);
resultnc = resultnc * SIGN(resultnc);
thetanh = theta2 * SIGN(theta2);
thetanc = theta3 * SIGN(theta3);

switch(count)
{
   case 0:
      sprintf(name, "pos_coords");
      break;
   case 1:
      sprintf(name, "neg_coords");
      break;
}

if (((resultnh < (thetanh * 1.05)) && (resultnh > (thetanh * 0.95))) &&
    ((resultnc < (thetanc * 1.05)) && (resultnc > (thetanc * 0.95))))
{
   if ((workfile = fopen(name, "w")) == NULL)
   {
      fprintf(stderr, "\nCould not open the file %s.\n", name);
      fprintf(stderr, "Data not saved.\n");
      exit(1);
   }

   fprintf(workfile, "\n\t\t\t%s\n", name);
   fprintf(workfile, "\nInputed Values\n");
   fprintf(workfile, "\nFirst Plane\n");
   fprintf(workfile, "\tcos(ZNH) Bond Angle:\t%9.4f\n", theta0);
   fprintf(workfile, "\tcos(ZNC) Bond Angle:\t%9.4f\n", theta1);
   fprintf(workfile, "\tPhi:\t\t\t%9.4f\n", torsion0);
   fprintf(workfile, "\nSecond Plane\n");
   fprintf(workfile, "\tPsi:\t\t\t%9.4f\n", torsion1);
   fprintf(workfile, "\tcos(ZNH) Bond Angle:\t%9.4f\n", theta2);
   fprintf(workfile, "\tcos(ZNC) Bond Angle:\t%9.4f\n", theta3);
   fprintf(workfile, "\n%20s %14s %14s\n", headerX, headerY, headerZ);
   fprintf(workfile, "caim1      %14.9f %14.9f %14.9f\n",
                       caim1[X], caim1[Y], caim1[Z]);
   fprintf(workfile, "cim1       %14.9f %14.9f %14.9f\n",
                       cim1[X], cim1[Y], cim1[Z]);
   fprintf(workfile, "oim1       %14.9f %14.9f %14.9f\n",
                              oim1[X], oim1[Y], oim1[Z]);
         fprintf(workfile, "ni          %14.9f %14.9f %14.9f\n",
                              ni[X], ni[Y], ni[Z]);
         fprintf(workfile, "hi          %14.9f %14.9f %14.9f\n",
                              hi[X], hi[Y], hi[Z]);
         fprintf(workfile, "cai         %14.9f %14.9f %14.9f\n",
                              cai[X], cai[Y], cai[Z]);
         fprintf(workfile, "ci          %14.9f %14.9f %14.9f\n",
                              ci[X], ci[Y], ci[Z]);
         fprintf(workfile, "oi          %14.9f %14.9f %14.9f\n",
                              oi[X], oi[Y], oi[Z]);
         fprintf(workfile, "nip1        %14.9f %14.9f %14.9f\n",
                              nip1[X], nip1[Y], nip1[Z]);
         fprintf(workfile, "hip1        %14.9f %14.9f %14.9f\n",
                              hip1[X], hip1[Y], hip1[Z]);
         fprintf(workfile, "caip1       %14.9f %14.9f %14.9f\n",
                              caip1[X], caip1[Y], caip1[Z]);

         fclose(workfile);
         fprintf(stdout, "Final coordinates stored in the file: %s\n\n", name);
      }
   }

   void main(void)
   {
      float theta[4];
      float torsion[2];
      float nc[3];
      float nh[3];
      float y[3];
      float alpha, beta;
      float temp;
      char  name[256];
      int   count;

      input_data(theta, torsion);

      for(count = 0; count < 2; ++count)
      {
         define_plane();
         calculate_nc(nc, theta);
         calculate_nh(nh, theta, count);
         calculate_y(y, nc, nh);
         alpha = acos(y[X]) * SIGN(y[Z]);
         beta = theta[NCTHETA1];

         rotate_mf_to_lab(alpha, beta, caim1_cim1);
         rotate_mf_to_lab(alpha, beta, cim1_oim1);
         rotate_mf_to_lab(alpha, beta, cim1_ni);
         rotate_mf_to_lab(alpha, beta, ni_hi);
         rotate_mf_to_lab(alpha, beta, ni_cai);
         rotate_mf_to_lab(alpha, beta, cai_ci);
         rotate_mf_to_lab(alpha, beta, ci_oi);
         rotate_mf_to_lab(alpha, beta, ci_nip1);
         rotate_mf_to_lab(alpha, beta, nip1_hip1);
         rotate_mf_to_lab(alpha, beta, nip1_caip1);

         rotate_torsion(ni_cai, torsion[PHI], cai_ci);
         rotate_torsion(ni_cai, torsion[PHI], ci_oi);
         rotate_torsion(ni_cai, torsion[PHI], ci_nip1);
         rotate_torsion(ni_cai, torsion[PHI], nip1_hip1);
         rotate_torsion(ni_cai, torsion[PHI], nip1_caip1);
```

```
        rotate_torsion(cai_ci, torsion[PSI], ci_oi);
        rotate_torsion(cai_ci, torsion[PSI], ci_nip1);
        rotate_torsion(cai_ci, torsion[PSI], nip1_hip1);
        rotate_torsion(cai_ci, torsion[PSI], nip1_caip1);

        generate_coords(count, theta, torsion);
    }
}
```

A.4.3 TORC

TORC (TOtal Refinement of Constraints) is used to refine the protein structure so that all of the experimental constraints are met while also minimizing the CHARMM energy.

A.4.3.1 TORC source code. The following source code is actually a concatenation of several files that together make up the TORC program. The individual files, or modules, can be compiled into a self operating program outside of CHARMM using the Makefile provided in appendix A.4.3.2, but doing so will only allow structural modifications via torsion angle moves and will not include the CHARMM energy. The TORC code can also be compiled as a module within CHARMM, thus allowing for all structural modification moves and the inclusion of the CHARMM energy. Please contact me for instructions on using this code within CHARMM.

```
/*
File Torc.h
Header file containing global declarations.
*/

/*
program TORC (TOtal Refinement of Constraints)

Randal R. Ketchem
Institute of Molecular Biophysics   904.644.1309 (voice)
Florida State University            904.644.1366 (FAX)
Tallahassee, FL 32306-3015          rrk@magnet.fsu.edu (email)
*/

/*
Programming Conventions

Global definitions are declared starting with 'gd' and every following word
capitalized, as in 'gdNumLambda'.

Local definitions are declared starting with 'ld' and every following word
capitalized, as in 'ldNoPrint'.

The convention for definitions is not followed when the definition would be
better as a single word or macro, as in 'TRUE', 'THREE' or 'DSQR(a)'.

Global variables are declared starting with 'g' and every following word
capitalized, as in 'gNumRes' or 'gAtoms'.

Internal variables are declared starting with a lowercase and every following
```

```
word capitalized, as in 'atomTwoType' or 'resNum' or 'alpha'.

Functions are declared starting with an uppercase and every following word
capitalized, as in 'CalcNCPenalty' or OutputPDB'. This convention is not
followed for the function 'main'.
*/

/*
Version History
5.4    Added compensating peptide plane moves as a move option.
       Outputs C-O and N-H peptide plane orientations before and after
          refinement.
       Outputs an orientation file containing C-O and N-H peptide plane
          orientations for each site as a function of attempted moves.
       Added tunneling for peptide planes as a move option. This switches
          the orientation of the carbonyl with respect to the channel axis.
       Made compensating and tunneling moves possible during both torsion
          and atom refinement.
       Add torsion moves as an option during atom refinement.
       Changed Reads so that data input files define the atom names instead
          of having them hard wired.
       All move types defined by ratios given in control file.
5.3    Add a global translation to each atom move. This is to help the
          dimerization step so that the monomers line up. This will have no
          effect on the internal values, but will on the charmm energy.
       Changed the random number call from rand(), which has a range of 0 to
          (2^15)-1 and a period of 2^32, to random(), which has a range of
          (2^31)-1 and a period of approximately 16*((2^31)-1).
5.2    Add indole chemical shift as a penalty.
       Add indole N-H dipolar splitting as a penalty.
       Penalty function straight quadratic. Penalty zero only if calc = obs.
5.1    Trajectories saved in variables and printed when needed.
       Input number of cycles at each T and number of T cycles instead of
          calculating those values from the number of modifiable torsion
          angles in the limit file.
       New gaussian random number generator based on Box-Muller.
       Penalty data files are read when they exist, ignored otherwise.
5.0    Charmm and Non-Charmm now use same source files.
3.8    Add error to output and adjust output.
3.7    Remove B2 and replace with calculation of C-D quadrupole splittings.
       Calculate penalty as square of deviation from limit divided by square
          of error. Penalty set to zero if within experimental error.
3.6    Use individual nuParallel value for each dipole interaction.
3.5    Create x, y and z arrays for charmm after each successful rotation.
3.4    Output data as a function of successful rotations (trajectories).
       Fixed bug in calculation of carbon chemical shift.
3.3    Remove totLoop. Refine until numSuccess or penalty is 0.
       Input number of rotations per history output in control file.
       Can use comments in input files, except for the pdb file. Comment lines
          begin with "#". (Of course, only ATOM records are read from pdb
          files, so comments could certainly be used in a pdb file.)
       Input files use header identifiers instead of requiring a specific data
          order.
3.2    Fixed reading PDB files with a chain identifier.
       CalcResPT now shows all possible n-(n-2).
       CalcResPT calculates pitch based on dipeptide plane.
       If final penalty > lowest penalty, set totLoop=1 with lowest penalty
          coordinates.
2.0    The next version is so much of an upgrade that I skipped v2.0.
1.0    It's Alive! Doesn't align helix or read PDB files or many other nice
       things, but it moves a single turn around so that the hydrogen bonds
       match without deviating from the observed data, which convinced us
```

```
     that the method will work.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define X                0
#define Y                1
#define Z                2

#define ONE              0
#define TWO              1
#define THREE            2

#define FALSE            0
#define TRUE             1

#ifndef NORMTORC
   #define gdAtomMove        0
   #define gdTorsionMove     1
#endif

/* array values for lambda and number of lambda used */
#define NCS              0
#define CCS              1
#define ICS              2
#define NC               3
#define NH               4
#define IH               5
#define DIS              6
#define CD               7
#ifdef NORMTORC
   #define gdNumLambda       8
#else
   #define E                 8
   #define gdNumLambda       9
#endif

#define gdLineLength     256
#define gdPi             3.1415926535897932
#define gdRadPerDeg      (gdPi / 180.0)
#define gdBoltzmann      1.9874643E-03

/* returns double square of 'a' */
static double sqrArg;
#define DSQR(a)    ((sqrArg=(a)) == 0.0 ? 0.0 : sqrArg * sqrArg)
/* returns integer sign of 'a' */
#define SIGN(a)    ((a) < 0.0 ? -1 : 1)

#define gdVersion        "TORC (TOtal Refinement of Constraints) v5.4"

/* holds PDB ATOM lines */
typedef struct
{
   char     header[7];
   int      atomSeqNum;
   char     atomName[5],
            altLocInd[2],
            resName[4],
            chainIdent[2];
   int      resSeqNum;
   char      insertRes[2];
   double    coords[3],
             occupancy,
             tempFactor;
   int       footnoteNum;
} atom;

/* holds begin and end values of residues */
typedef struct
{
   int    begin,
          end;
} residue;

/* holds distance data */
typedef struct
{
   int       atomOneResNumber;
   char      atomOneType[gdLineLength];
   int       atomTwoResNumber;
   char      atomTwoType[gdLineLength];
   double    a1a2Distance,
             expError;
} disData;

/* holds chemical shift data */
typedef struct
{
   int       resNum;
   char      atom1Name[gdLineLength],
             atom2Name[gdLineLength],
             atom3Name[gdLineLength];
   double    PAS[3][3],
             MF[3][3],
             alpha,
             beta,
             obsCS,
             delta,
             expError;
} csData;

/* holds dipolar splitting data */
typedef struct
{
   int       resNum;
   char      atom1Name[gdLineLength],
             atom2Name[gdLineLength];
   double    dip,
             nuParallel,
             expError;
} dipData;

/* holds 2H quadrupolar splitting data */
typedef struct
{
   int       resNum;
   char      atom1Name[gdLineLength],
             atom2Name[gdLineLength];
   double    quad,
             qcc,
             expError;
} cdData;
```

```
/* holds torsion angle limits */
typedef struct
{
   int      resNum;
   char     bondName[gdLineLength];
   double   low,
            high;
} limit;

/* check to see if main() is including this file */
/* if so, declare variables extern */
#ifndef MAIN
#define WHO extern
#else
#define WHO
#endif

WHO atom     *gAtoms,
             *gAtomsInitial,
             *gAtomsLowest,
             *gAtomsLastState;
WHO residue *gResidues;
WHO disData *gDisDatas;
WHO csData  *gNDatas,
            *gCDatas,
            *gIDatas;
WHO dipData *gNCDatas,
            *gNHDatas,
            *gIHDatas;
WHO cdData  *gCDDatas;
WHO limit   *gLimits;
WHO int     gNumLimits,
            gNumAtoms,
            gNumRes,
            gNumDisData,
            gNumNData,
            gNumCData,
            gNumIData,
            gNumNCData,
            gNumNHData,
            gNumIHData,
            gNumCDData,
            gCyclesAtT, /* number of cycles at one temperature */
            gCyclesForceT, /* number of successful moves at which
                              temperature is forced to change */
            gTCycles, /* number of temperature cycles */
                      /* if set to zero, go until zero numSuccess */
            gEquilSteps, /* number of attempted steps at the */
                         /* initial temperature */
            gHistoryFlag, /* 1 outputs history files, 0 does not */
            gMovePerHistory,  /* number of moves for each history frame */
            gSeedFlag, /* 1 uses input as random seed, 0 uses time() */
            gOriFlag, /* 1 outputs orientation file, 0 does not */
            gMovePerOri,  /* number of moves for each orientation */
            gTrajFlag, /* 1 outputs trajectory file, 0 does not */
            gTrajRes, /* Residue number to be used in trajectory */
            gMovePerTraj,  /* number of moves for each trajectory */
            *gNumMoveCompAtt, /* number of attempted */
                                 /* compensate moves/residue */
            *gNumMoveCompAcc, /* number of accepted */
                                 /* compensate moves/residue */
            *gNumMoveTunnAtt, /* number of attempted */
                              /* tunneling moves/residue */
            *gNumMoveTunnAcc, /* number of accepted */
                              /* tunneling moves/residue */
            *gNumMoveTorsAtt, /* number of attempted */
                               /* torsion moves/residue */
            *gNumMoveTorsAcc; /* number of accepted */
                               /* torsion moves/residue */
WHO long    gSRandSeed,       /* input random seed */
            gMaxRand; /* maximum long returned by random() */
WHO double  gTemperature, /* Current temperature */
            gInitTemp,  /* Initial temperature */
            gKBInitT,  /* Initial kbt */
            gTempFactor,  /* factor for temperature reduction */
            gLambda[gdNumLambda],    /* lambda values */
            gCompensateRatio,    /* Ratio of compensating moves to total */
                                  /* moves */
            gTunnelRatio,    /* Ratio of tunneling moves to total */
                             /* moves */
            gTorsionRatio, /* Ratio of torsion moves to total moves */
            gAtomRatio; /* Ratio of atom moves to total moves */
WHO double  gCalcNCS,
            gCalcCCS,
            gCalcICS,
            gCalcNC,
            gCalcNH,
            gCalcIH,
            gCalcCD,
            gCalcNCSP,
            gCalcCCSP,
            gCalcICSP,
            gCalcNCP,
            gCalcNHP,
            gCalcIHP,
            gCalcDISP,
            gCalcCDP,
            gCalcTotP;
#ifndef NORMTORC
WHO double  gCalcE,
            gCalcEP;
WHO double  *gXArray,
            *gYArray,
            *gZArray,
            *gXForce,
            *gYForce,
            *gZForce,
            *gXForceOldState,
            *gYForceOldState,
            *gZForceOldState,
            gDiffusion;
#endif
WHO FILE    *gTrajFile, /* output file for trajectories */
            *gOriFile; /* output file for orientations */

#undef WHO

/* declaration of functions */
double CalcCCSP(int print);
double CalcCDP(int print);
double CalcDeltaR(void);
double CalcDisP(int print);
double CalcICSP(int print);
double CalcIHP(int print);
double CalcNCP(int print);
double CalcNCSP(int print);
double CalcNHP(int print);
void CalcNorm(double vec1[3],
              double vec2[3],
              double norm[3]);
void CalcOri(char bondType[],
             unsigned long int totalCalc,
             int print);
double CalcPenalty(double calc,
                   double obs,
                   double expError);
int CalcResPT(void);
void CalcTheta(double atom1[3],
               double atom2[3],
               double atom3[3],
               double theta[3]);
double CalcTorAngle(double atom1[3],
                    double atom2[3],
                    double atom3[3],
                    double atom4[3]);
void CalcTorsion(void);
double CalcTotP(int print);
void CopyCoord(atom  *atomsIn,
               atom  *atomsCopy);
void CopyXYZ(atom  *atomsIn,
             atom  *atomsCopy);
void DeclArray(void);
void DoRotation(double rotMat[3][3],
                double atomToRotate[3],
                double origin[3]);
double DotProd(double vec1[3],
               double vec2[3]);
void EulerMatrix(double q[4],
                 double rotMatrix[3][3]);
void FindAtom(int residueNumber,
              char atomToFind[],
              double coords[3],
              int *found);
double GaussRand(double width);
double GetMag(double low,
              double high);
void MF_LF(double sigmaMF[3][3],
           double theta[3],
           double sigmaLF[3][3]);
void MakeXYZArray(atom *atomsIn,
                  double *x,
                  double *y,
                  double *z);
void MatMult(double matrix1[3][3],
             double matrix2[3][3],
             double result[3][3]);
int Metropolis(double dPenalty);
void MotavgTensor(double tensorIn[3][3],
                  double motAvgBond[3],
                  double delta,
                  double tensorOut[3][3]);
void MoveAtoms(atom *atomsIn,
               double diffusion);
int MoveCompensate(int *resNum);
int MoveTunnel(int *resNum);
```

```
void Normalize(double vector[3]);
void OutputPDB(char *fileName,
               atom *outAtoms);
void PAS_MF(double sigmaPAS[3][3],
            double alpha,
            double beta,
            double sigmaMF[3][3]);
void PrintHelp(void);
int ReadCDData(char fileName[]);
int ReadCData(char fileName[]);
void ReadContFile(char fileName[]);
int ReadDisFile(char fileName[]);
void ReadFiles(char dirName[]);
int ReadIData(char fileName[]);
void ReadLamFile(char fileName[]);
int ReadLimFile(char fileName[]);
int ReadNCData(char fileName[]);
int ReadNData(char fileName[]);
int ReadNHData(char fileName[]);
void ReadPDBFile(char fileName[]);
int RotateAtoms(atom *atomsIn,
                int resNum,
                char torType[],
                double mag);
#ifdef NORMTORC
int main(int argc,
         char *argv[]);
#else
int torc_(char inDirName[],
          int *dirLength);
#endif
void Transpose(double matrixIn[3][3],
               double matrixOut[3][3]);
void Vectorize(double atom1[3],
               double atom2[3],
               double vector[3]);


/*
File CalcCCSP.c
Calculates the penalty for the carbon chemical shift.
*/

#include "Torc.h"

double CalcCCSP(int print)
{
   int      resCount,
            dataCount,
            row,
            col,
            found;
   double   atomC[3],
            atomN[3],
            atomO[3],
/*
            PAS[3][3],
            alpha,
            beta,
*/
            delta,
            MF[3][3],
            obsCS,
            expError,
            theta[3],
            LF[3][3],
            sigmaAvg[3][3],
            motAvgBond[3],
            ca1[3],
            ca2[3],
            calcCS,
            csPenalty = 0.0,
            penalty;

   gCalcCCS = -1.0;
   for(dataCount = 0; dataCount < gNumCData; ++dataCount)
   {
      resCount = gCDatas[dataCount].resNum - 1;

      for(row = X; row <= Z; ++row)
      {
         for(col = X; col <= Z; ++col)
         {
/*
            PAS[row][col] = gCDatas[dataCount].PAS[row][col];
*/
            MF[row][col] = gCDatas[dataCount].MF[row][col];
         }
      }
/*
      alpha = gCDatas[dataCount].alpha;
      beta = gCDatas[dataCount].beta;
*/
      obsCS = gCDatas[dataCount].obsCS;
      delta = gCDatas[dataCount].delta;
      expError = gCDatas[dataCount].expError;

      found = FALSE;
      if((resCount + 1) < gNumRes)
      {
         FindAtom(resCount, gCDatas[dataCount].atom1Name, atomC, &found);
         if(found)
            FindAtom(resCount + 1, gCDatas[dataCount].atom2Name,
                  atomN, &found);
         if(found)
            FindAtom(resCount, gCDatas[dataCount].atom3Name,
                  atomO, &found);
      }

      if(!found)
      {
         if(print)
            fprintf(stdout, "ccs %-4s%3d%1s = ccs not found\n",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
      }
      else
      {
         CalcTheta(atomC, atomN, atomO, theta);
/*
         PAS_MF(PAS, alpha, beta, MF);
*/
         MF_LF(MF, theta, LF);
         if(delta == 0.0)
```

```
            calcCS = LF[Z][Z];
         else
         {
            FindAtom(resCount, "CA", ca1, &found);
            if(found && ((resCount + 1) < gNumRes))
               FindAtom(resCount + 1, "CA", ca2, &found);
            if(found)
            {
               Vectorize(ca1, ca2, motAvgBond);
               Normalize(motAvgBond);
               MotavgTensor(LF, motAvgBond, delta, sigmaAvg);
               calcCS = sigmaAvg[Z][Z];
            }
            else
               calcCS = LF[Z][Z];
         }

         penalty = CalcPenalty(calcCS, obsCS, expError);
         csPenalty += penalty;

         if(print)
         {
            fprintf(stdout, "ccs %-4s%3d%1s = ",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
            fprintf(stdout, "%10.4f %10.4f %10.4f",
                  calcCS,
                  obsCS,
                  calcCS - obsCS);
            if(fabs(calcCS - obsCS) <= expError)
               fprintf(stdout, " ");
            else
               fprintf(stdout, "*");
            fprintf(stdout, "%10.4f\n", expError);
         }

         if(resCount == gTrajRes - 1)
            gCalcCCS = calcCS;
      }
   }

   if(print && gNumCData)
      fprintf(stdout, "\n");

   return(csPenalty);
}

/*
File CalcCDP.c
Calculates the penalty for 2H quadrupole splittings.
*/

#include "Torc.h"

double CalcCDP(int print)
{
   int      found,
            resCount,
            dataCount;
   double   cdPenalty = 0.0,
            atomC[3],
         atomD[3],
         atomZ[3],
         vecCD[3],
         vecCZ[3],
         angle,
         calcQuad,
         obsQuad,
         expError,
         penalty;

gCalcCD = -1.0;
for(dataCount = 0; dataCount < gNumCDData; ++dataCount)
{
   resCount = gCDDatas[dataCount].resNum - 1;

   FindAtom(resCount, gCDDatas[dataCount].atom1Name, atomC, &found);
   if(found)
      FindAtom(resCount, gCDDatas[dataCount].atom2Name, atomD, &found);
   if(!found)
   {
      if(print)
         fprintf(stdout,
               "cd  %-4s%3d%1s = cd not found\n",
               gAtoms[gResidues[resCount].begin].resName,
               gAtoms[gResidues[resCount].begin].resSeqNum,
               gAtoms[gResidues[resCount].begin].insertRes);
   }
   else
   {
      obsQuad = gCDDatas[dataCount].quad;
      expError = gCDDatas[dataCount].expError;

      atomZ[X] = atomC[X];
      atomZ[Y] = atomC[Y];
      atomZ[Z] = atomC[Z] + 1.0;

      Vectorize(atomC, atomZ, vecCZ);
      Vectorize(atomC, atomD, vecCD);

      angle = DotProd(vecCD, vecCZ);
      calcQuad = 3.0 * DSQR(angle);
      calcQuad = calcQuad - 1.0;
      calcQuad = 0.75 * gCDDatas[dataCount].qcc * calcQuad;
      calcQuad = fabs(calcQuad);

      penalty = CalcPenalty(calcQuad, obsQuad, expError);
      cdPenalty += penalty;

      if(print)
      {
         fprintf(stdout, "cd  %-4s%3d%1s = ",
               gAtoms[gResidues[resCount].begin].resName,
               gAtoms[gResidues[resCount].begin].resSeqNum,
               gAtoms[gResidues[resCount].begin].insertRes);
         fprintf(stdout, "%10.4f %10.4f %10.4f",
               calcQuad,
               obsQuad,
               calcQuad - obsQuad);
         if(fabs(calcQuad - obsQuad) <= expError)
            fprintf(stdout, " ");
         else
            fprintf(stdout, "*");
            fprintf(stdout, "%10.4f", expError);
            fprintf(stdout, " angle = %5.1f\n", acos(angle) / gdRadPerDeg);
         }

         if(resCount == gTrajRes - 1)
            gCalcCD = calcQuad;
      }
   }

   if(print && gNumCDData)
      fprintf(stdout, "\n");

   return(cdPenalty);
}

/*
File CalcDeltaR.c
Calculates the total atomic dR and rmse between the original and refined
coordinates. Outputs both to stdout, but returns dR only.
*/

#include "Torc.h"

double CalcDeltaR(void)
{
   int      atomCount;
   double   rmse,
            dR = 0.0;

   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
   {
      dR += DSQR(gAtomsInitial[atomCount].coords[X]
               - gAtoms[atomCount].coords[X])
          + DSQR(gAtomsInitial[atomCount].coords[Y]
               - gAtoms[atomCount].coords[Y])
          + DSQR(gAtomsInitial[atomCount].coords[Z]
               - gAtoms[atomCount].coords[Z]);
   }
   dR = sqrt(dR) / (double) gNumAtoms;
   rmse = dR * sqrt((double) gNumAtoms);

   fprintf(stdout, "dR   of Refined to Original: %15.6e\n", dR);
   fprintf(stdout, "RMSE of Refined to Original: %15.6e\n", rmse);

   return(dR);
}

/*
File CalcDisP.c
Calculate the penalty for atom distances.
*/

#include "Torc.h"
#include <string.h>

double CalcDisP(int print)
{
   int      found = FALSE,
            dataCount;
   double   disPenalty = 0.0,
            atom1[3],
            atom2[3],
          calcDis,
          obsDis,
          expError,
          penalty;
char      disName[gdLineLength];

for(dataCount = 0; dataCount < gNumDisData; ++dataCount)
{
   FindAtom(gDisDatas[dataCount].atomOneResNumber - 1,
            gDisDatas[dataCount].atomOneType,
            atom1,
            &found);
   if(found)
      FindAtom(gDisDatas[dataCount].atomTwoResNumber - 1,
               gDisDatas[dataCount].atomTwoType,
               atom2,
               &found);
   if(!found)
   {
      if(print)
      {
         sprintf(disName, "%d%s-%d%s",
               gDisDatas[dataCount].atomOneResNumber,
               gDisDatas[dataCount].atomOneType,
               gDisDatas[dataCount].atomTwoResNumber,
               gDisDatas[dataCount].atomTwoType);
         if(strlen(disName) > 8) disName[8] = '\0';
         fprintf(stdout, "dis %-8s = dis not found\n", disName);
      }
   }
   else
   {
      calcDis = sqrt(DSQR(atom2[X] - atom1[X])
                   + DSQR(atom2[Y] - atom1[Y])
                   + DSQR(atom2[Z] - atom1[Z]));
      obsDis = gDisDatas[dataCount].a1a2Distance;
      expError = gDisDatas[dataCount].expError;

      penalty = CalcPenalty(calcDis, obsDis, expError);
      disPenalty += penalty;

      if(print)
      {
         sprintf(disName, "%d%s-%d%s",
               gDisDatas[dataCount].atomOneResNumber,
               gDisDatas[dataCount].atomOneType,
               gDisDatas[dataCount].atomTwoResNumber,
               gDisDatas[dataCount].atomTwoType);
         if(strlen(disName) > 8) disName[8] = '\0';
         fprintf(stdout, "dis %-8s = %10.4f %10.4f %10.4f",
               disName,
               calcDis,
               obsDis,
               calcDis - obsDis);
         if(fabs(calcDis - obsDis) <= expError)
            fprintf(stdout, " ");
         else
            fprintf(stdout, "*");
         fprintf(stdout, "%10.4f\n", expError);
      }
   }
}
```

```
   if(print && gNumDisData)
      fprintf(stdout, "\n");

   return(disPenalty);
}

/*
File CalcICSP.c
Calculates the penalty for the indole nitrogen chemical shift.
*/

#include "Torc.h"

double CalcICSP(int print)
{
   int      resCount,
            dataCount,
            row,
            col,
            found;
   double   atomN[3],
            atomC[3],
            atomH[3],
/*
            PAS[3][3],
            alpha,
            beta,
*/
            delta,
            MF[3][3],
            obsCS,
            expError,
            theta[3],
            LF[3][3],
            sigmaAvg[3][3],
            motAvgBond[3],
            cb[3],
            cg[3],
            calcCS,
            csPenalty = 0.0,
            penalty;

   gCalcICS = -1.0;
   for(dataCount = 0; dataCount < gNumIData; ++dataCount)
   {
      resCount = gIDatas[dataCount].resNum - 1;

      for(row = X; row <= Z; ++row)
      {
         for(col = X; col <= Z; ++col)
         {
/*
            PAS[row][col] = gIDatas[dataCount].PAS[row][col];
*/
            MF[row][col] = gIDatas[dataCount].MF[row][col];
         }
      }
/*
      alpha = gIDatas[dataCount].alpha;
      beta = gIDatas[dataCount].beta;
*/
```

```
      obsCS = gIDatas[dataCount].obsCS;
      delta = gIDatas[dataCount].delta;
      expError = gIDatas[dataCount].expError;

      FindAtom(resCount, gIDatas[dataCount].atom1Name, atomN, &found);
      if(found)
         FindAtom(resCount, gIDatas[dataCount].atom2Name, atomC, &found);
      if(found)
         FindAtom(resCount, gIDatas[dataCount].atom3Name, atomH, &found);

      if(!found)
      {
         if(print)
            fprintf(stdout, "ics %-4s%3d%1s = ics not found\n",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
      }
      else
      {
         CalcTheta(atomN, atomH, atomC, theta);
/*
         PAS_MF(PAS, alpha, beta, MF);
*/
         MF_LF(MF, theta, LF);
         if(delta == 0.0)
            calcCS = LF[Z][Z];
         else
         {
            FindAtom(resCount, "CB", cb, &found);
            if(found)
               FindAtom(resCount, "CG", cg, &found);
            if(found)
            {
               Vectorize(cb, cg, motAvgBond);
               Normalize(motAvgBond);
               MotavgTensor(LF, motAvgBond, delta, sigmaAvg);
               calcCS = sigmaAvg[Z][Z];
            }
            else
               calcCS = LF[Z][Z];
         }

         penalty = CalcPenalty(calcCS, obsCS, expError);
         csPenalty += penalty;

         if(print)
         {
            fprintf(stdout, "ics %-4s%3d%1s = ",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
            fprintf(stdout, "%10.4f %10.4f %10.4f",
                  calcCS,
                  obsCS,
                  calcCS - obsCS);
            if(fabs(calcCS - obsCS) <= expError)
               fprintf(stdout, " ");
            else
               fprintf(stdout, "*");
            fprintf(stdout, "%10.4f\n", expError);
         }
         if(resCount == gTrajRes - 1)
            gCalcICS = calcCS;
      }
   }

   if(print && gNumIData)
      fprintf(stdout, "\n");

   return(csPenalty);
}

/*
File CalcIHP.c
Calculates the penalty for the IH dipole splitting.
*/

#include "Torc.h"

double CalcIHP(int print)
{
   int      found,
            resCount,
            dataCount;
   double   angle,
            calcDip,
            obsDip,
            expError,
            atomN[3],
            atomH[3],
            atomZ[3],
            vecNH[3],
            vecNZ[3],
            ihPenalty = 0.0,
            penalty;

   gCalcIH = -1.0;
   for(dataCount = 0; dataCount < gNumIHData; ++dataCount)
   {
      resCount = gIHDatas[dataCount].resNum - 1;

      FindAtom(resCount, gIHDatas[dataCount].atom1Name, atomN, &found);
      if(found)
         FindAtom(resCount, gIHDatas[dataCount].atom2Name, atomH, &found);

      if(!found)
      {
         if(print)
            fprintf(stdout, "ih  %-4s%3d%1s = ih not found\n",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
      }
      else
      {
         obsDip = gIHDatas[dataCount].dip;
         expError = gIHDatas[dataCount].expError;

         atomZ[X] = atomN[X];
         atomZ[Y] = atomN[Y];
         atomZ[Z] = atomN[Z] + 1.0;
         Vectorize(atomN, atomZ, vecNZ);
         Vectorize(atomN, atomH, vecNH);

         angle = DotProd(vecNH, vecNZ);
         calcDip = 3.0 * DSQR(angle);
         calcDip = calcDip - 1.0;
         calcDip = gIHDatas[dataCount].nuParallel * fabs(calcDip);

         penalty = CalcPenalty(calcDip, obsDip, expError);
         ihPenalty += penalty;

         if(print)
         {
            fprintf(stdout, "ih  %-4s%3d%1s = ",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
            fprintf(stdout, "%10.4f %10.4f %10.4f",
                  calcDip,
                  obsDip,
                  calcDip - obsDip);
            if(fabs(calcDip - obsDip) <= expError)
               fprintf(stdout, " ");
            else
               fprintf(stdout, "*");
            fprintf(stdout, "%10.4f", expError);
            fprintf(stdout, " angle = %5.1f\n", acos(angle) / gdRadPerDeg);
         }

         if(resCount == gTrajRes - 1)
            gCalcIH = calcDip;
      }
   }

   if(print && gNumIHData)
      fprintf(stdout, "\n");

   return(ihPenalty);
}

/*
File CalcNCP.c
Calculates the penalty for the NC dipole splitting.
*/

#include "Torc.h"

double CalcNCP(int print)
{
   int      found,
            resCount,
            dataCount;
   double   angle,
            calcDip,
            obsDip,
            expError,
            atomN[3],
            atomC[3],
            atomZ[3],
            vecNC[3],
            vecNZ[3],
            ncPenalty = 0.0,
         penalty;

gCalcNC = -1.0;
for(dataCount = 0; dataCount < gNumNCData; ++dataCount)
{
   resCount = gNCDatas[dataCount].resNum - 1;

   found = FALSE;
   if(resCount > 0)
   {
      FindAtom(resCount, gNCDatas[dataCount].atom1Name,
            atomN, &found);
      if(found)
         FindAtom(resCount - 1, gNCDatas[dataCount].atom2Name,
            atomC, &found);
   }

   if(!found)
   {
      if(print)
         fprintf(stdout, "nc  %-4s%3d%1s = nc not found\n",
               gAtoms[gResidues[resCount].begin].resName,
               gAtoms[gResidues[resCount].begin].resSeqNum,
               gAtoms[gResidues[resCount].begin].insertRes);
   }
   else
   {
      obsDip = gNCDatas[dataCount].dip;
      expError = gNCDatas[dataCount].expError;

      atomZ[X] = atomN[X];
      atomZ[Y] = atomN[Y];
      atomZ[Z] = atomN[Z] + 1.0;

      Vectorize(atomN, atomZ, vecNZ);
      Vectorize(atomN, atomC, vecNC);

      angle = DotProd(vecNC, vecNZ);
      calcDip = 3.0 * DSQR(angle);
      calcDip = calcDip - 1.0;
      calcDip = gNCDatas[dataCount].nuParallel * fabs(calcDip);

      penalty = CalcPenalty(calcDip, obsDip, expError);
      ncPenalty += penalty;

      if(print)
      {
         fprintf(stdout, "nc  %-4s%3d%1s = ",
               gAtoms[gResidues[resCount].begin].resName,
               gAtoms[gResidues[resCount].begin].resSeqNum,
               gAtoms[gResidues[resCount].begin].insertRes);
         fprintf(stdout, "%10.4f %10.4f %10.4f",
               calcDip,
               obsDip,
               calcDip - obsDip);
         if(fabs(calcDip - obsDip) <= expError)
            fprintf(stdout, " ");
         else
            fprintf(stdout, "*");
         fprintf(stdout, "%10.4f", expError);
         fprintf(stdout, " angle = %5.1f\n", acos(angle) / gdRadPerDeg);
      }
```

```
         if(resCount == gTrajRes - 1)
            gCalcNC = calcDip;
      }
   }

   if(print && gNumNCData)
      fprintf(stdout, "\n");

   return(ncPenalty);
}

/*
File CalcNCSP.c
Calculates the penalty for the nitrogen chemical shift.
*/

#include "Torc.h"

double CalcNCSP(int print)
{
   int      resCount,
            dataCount,
            row,
            col,
            found;
   double   atomN[3],
            atomC[3],
            atomH[3],
/*
            PAS[3][3],
            alpha,
            beta,
*/
            delta,
            MF[3][3],
            obsCS,
            expError,
            theta[3],
            LF[3][3],
            sigmaAvg[3][3],
            motAvgBond[3],
            ca1[3],
            ca2[3],
            calcCS,
            csPenalty = 0.0,
            penalty;

   gCalcNCS = -1.0;
   for(dataCount = 0; dataCount < gNumNData; ++dataCount)
   {
      resCount = gNDatas[dataCount].resNum - 1;

      for(row = X; row <= Z; ++row)
      {
         for(col = X; col <= Z; ++col)
         {
/*
            PAS[row][col] = gNDatas[dataCount].PAS[row][col];
*/
            MF[row][col] = gNDatas[dataCount].MF[row][col];
         }
      }
/*
      alpha = gNDatas[dataCount].alpha;
      beta = gNDatas[dataCount].beta;
*/
      obsCS = gNDatas[dataCount].obsCS;
      delta = gNDatas[dataCount].delta;
      expError = gNDatas[dataCount].expError;

      found = FALSE;
      if(resCount > 0)
      {
         FindAtom(resCount, gNDatas[dataCount].atom1Name,
               atomN, &found);
         if(found)
            FindAtom(resCount - 1, gNDatas[dataCount].atom2Name,
               atomC, &found);
         if(found)
            FindAtom(resCount, gNDatas[dataCount].atom3Name,
               atomH, &found);
      }

      if(!found)
      {
         if(print)
            fprintf(stdout, "ncs %-4s%3d%1s = ncs not found\n",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
      }
      else
      {
         CalcTheta(atomN, atomC, atomH, theta);
/*
         PAS_MF(PAS, alpha, beta, MF);
*/
         MF_LF(MF, theta, LF);
         if(delta == 0.0)
            calcCS = LF[Z][Z];
         else
         {
            FindAtom(resCount, "CA", ca1, &found);
            if(found && ((resCount + 1) < gNumRes))
               FindAtom(resCount + 1, "CA", ca2, &found);
            if(found)
            {
               Vectorize(ca1, ca2, motAvgBond);
               Normalize(motAvgBond);
               MotavgTensor(LF, motAvgBond, delta, sigmaAvg);
               calcCS = sigmaAvg[Z][Z];
            }
            else
               calcCS = LF[Z][Z];
         }

         penalty = CalcPenalty(calcCS, obsCS, expError);
         csPenalty += penalty;

         if(print)
         {
            fprintf(stdout, "ncs %-4s%3d%1s = ",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
            fprintf(stdout, "%10.4f %10.4f %10.4f",
                  calcCS,
                  obsCS,
                  calcCS - obsCS);
            if(fabs(calcCS - obsCS) <= expError)
               fprintf(stdout, "  ");
            else
               fprintf(stdout, "**");
            fprintf(stdout, "%10.4f\n", expError);
         }

         if(resCount == gTrajRes - 1)
            gCalcNCS = calcCS;
      }
   }

   if(print && gNumNData)
      fprintf(stdout, "\n");

   return(csPenalty);
}

/*
File CalcNHP.c
Calculates the penalty for the NH dipole splitting.
*/

#include "Torc.h"

double CalcNHP(int print)
{
   int      found,
            resCount,
            dataCount;
   double   angle,
            calcDip,
            obsDip,
            expError,
            atomN[3],
            atomH[3],
            atomZ[3],
            vecNH[3],
            vecNZ[3],
            nhPenalty = 0.0,
            penalty;

   gCalcNH = -1.0;
   for(dataCount = 0; dataCount < gNumNHData; ++dataCount)
   {
      resCount = gNHDatas[dataCount].resNum - 1;

      FindAtom(resCount, gNHDatas[dataCount].atom1Name, atomN, &found);
      if(found)
         FindAtom(resCount, gNHDatas[dataCount].atom2Name, atomH, &found);

      if(!found)
      {
         if(print)
            fprintf(stdout, "nh  %-4s%3d%1s = nh not found\n",
                  gAtoms[gResidues[resCount].begin].resName,
                gAtoms[gResidues[resCount].begin].resSeqNum,
                gAtoms[gResidues[resCount].begin].insertRes);
      }
      else
      {
         obsDip = gNHDatas[dataCount].dip;
         expError = gNHDatas[dataCount].expError;

         atomZ[X] = atomN[X];
         atomZ[Y] = atomN[Y];
         atomZ[Z] = atomN[Z] + 1.0;

         Vectorize(atomN, atomZ, vecNZ);
         Vectorize(atomN, atomH, vecNH);

         angle = DotProd(vecNH, vecNZ);
         calcDip = 3.0 * DSQR(angle);
         calcDip = calcDip - 1.0;
         calcDip = gNHDatas[dataCount].nuParallel * fabs(calcDip);

         penalty = CalcPenalty(calcDip, obsDip, expError);
         nhPenalty += penalty;

         if(print)
         {
            fprintf(stdout, "nh  %-4s%3d%1s = ",
                  gAtoms[gResidues[resCount].begin].resName,
                  gAtoms[gResidues[resCount].begin].resSeqNum,
                  gAtoms[gResidues[resCount].begin].insertRes);
            fprintf(stdout, "%10.4f %10.4f %10.4f",
                  calcDip,
                  obsDip,
                  calcDip - obsDip);
            if(fabs(calcDip - obsDip) <= expError)
               fprintf(stdout, " ");
            else
               fprintf(stdout, "*");
            fprintf(stdout, "%10.4f", expError);
            fprintf(stdout, " angle = %5.1f\n", acos(angle) / gdRadPerDeg);
         }

         if(resCount == gTrajRes - 1)
            gCalcNH = calcDip;
      }
   }

   if(print && gNumNHData)
      fprintf(stdout, "\n");

   return(nhPenalty);
}

/*
File CalcNorm.c
Calculates the normal of two vectors.
*/

#include "Torc.h"

void CalcNorm(double vec1[3],
              double vec2[3],
              double norm[3])
```

```
{
   norm[X] = (vec1[Y] * vec2[Z]) - (vec2[Y] * vec1[Z]);
   norm[Y] = (vec1[Z] * vec2[X]) - (vec2[Z] * vec1[X]);
   norm[Z] = (vec1[X] * vec2[Y]) - (vec2[X] * vec1[Y]);
}

/*
File CalcOri.c
Calculates the N-H and C-O bond orientations relative to Z.
*/

#include "Torc.h"
#include <string.h>

void CalcOri(char bondType[],
             unsigned long int totalCalc,
             int print)
{
   int      resCount,
            found;
   static int  firstTime = TRUE;
   char     atom1Type[gdLineLength],
            atom2AType[gdLineLength],
            atom2BType[gdLineLength],
            atom3Type[gdLineLength];
   double   atom1[3],
            atom2[3],
            atom3[3],
            atomZ[3],
            vec1Z[3],
            vec12[3],
            vec13[3],
            normal[3],
            planeAngle,
            bondAngle;

   if(strcmp(bondType, "N-H") == 0)
   {
      sprintf(atom1Type, "N");
      sprintf(atom2AType, "HN");
      sprintf(atom2BType, "H");
      sprintf(atom3Type, "CA");
   }

   if(strcmp(bondType, "C-O") == 0)
   {
      sprintf(atom1Type, "C");
      sprintf(atom2AType, "O");
      sprintf(atom2BType, "O");
      sprintf(atom3Type, "CA");
   }

   if(gOriFlag && !firstTime)
      fprintf(gOriFile, "%i\t%s", totalCalc, bondType);

   for(resCount = 0; resCount < gNumRes; ++resCount)
   {
      FindAtom(resCount, atom1Type, atom1, &found);
      if(found)
      {
         FindAtom(resCount, atom2AType, atom2, &found);
         if(!found)
         FindAtom(resCount, atom2BType, atom2, &found);
      }
      if(found)
         FindAtom(resCount, atom3Type, atom3, &found);

      if(found)
      {
         atomZ[X] = atom1[X];
         atomZ[Y] = atom1[Y];
         atomZ[Z] = atom1[Z] + 1.0;
         Vectorize(atom1, atomZ, vec1Z);
         Vectorize(atom1, atom2, vec12);
         Vectorize(atom1, atom3, vec13);
         CalcNorm(vec12, vec13, normal);

         planeAngle = acos(DotProd(normal, vec1Z)) / gdRadPerDeg;
         bondAngle = acos(DotProd(vec12, vec1Z)) / gdRadPerDeg;

         if(strcmp(atom1Type, "N")  == 0)
            planeAngle = planeAngle - 90.0;
         if(strcmp(atom1Type, "C")  == 0)
            planeAngle = 90.0 - planeAngle;

         if(gOriFlag)
         {
            if(firstTime)
               fprintf(gOriFile, "\t%d",
                     gAtoms[gResidues[resCount].begin].resSeqNum);
            else
               fprintf(gOriFile, "\t%.1f", planeAngle);
         }

         if(print)
         {
            fprintf(stdout, "%-4s%3d%1s %s",
                     gAtoms[gResidues[resCount].begin].resName,
                     gAtoms[gResidues[resCount].begin].resSeqNum,
                     gAtoms[gResidues[resCount].begin].insertRes,
                     bondType);

            if(planeAngle < -3.0)
               fprintf(stdout, "      out");
            else if(planeAngle > 3.0)
               fprintf(stdout, "       in");
            else
               fprintf(stdout, " parallel");
            fprintf(stdout, " %6.1f", planeAngle);

            if(bondAngle > 93.0)
               fprintf(stdout, "   down");
            else if(bondAngle < 87.0)
               fprintf(stdout, "     up");
            else
               fprintf(stdout, "   perp");

            fprintf(stdout, "\n");
         }
      }
   }

   if(gOriFlag)
      fprintf(gOriFile, "\n");
   if(firstTime)
      firstTime = FALSE;
}

/*
File CalcPenalty.c
Calculates the penalty.
*/

#include "Torc.h"

#undef ldFLAT
#undef ldPARABOLIC
/*
#define ldFLAT
*/
#define ldPARABOLIC

double CalcPenalty(double calc,
                   double obs,
                   double expError)
{
   double   penalty;
#ifdef ldFLAT
   double   errorLimit,
            amplitude;

   errorLimit = fabs(calc - obs) - expError;
#endif

/* keep
   amplitude = gTemperature * gdBoltzmann / DSQR(expError);
   amplitude = gKBInitT / DSQR(expError);
*/
#ifdef ldFLAT
   amplitude = 1.0 / DSQR(expError);

   penalty = (errorLimit <= 0.0) ? 0.0 : 0.5 * amplitude * DSQR(errorLimit);

   return(penalty);
#endif

#ifdef ldPARABOLIC
   penalty = 0.5 * DSQR((calc - obs) / expError);
   return(penalty);
#endif
}
#undef ldFLAT
#undef ldPARABOLIC

/*
File CalcResPT.c
Calculates residues per turn and pitch.
*/

#include "Torc.h"

int CalcResPT(void)
{
   typedef struct
   {
```

```
   int      resNum;
   double   ca[3];
} caAtom;

caAtom   *caAtoms;
double   resPerTurn,
         pitch,
         totRPT = 0.0,
         totPitch = 0.0,
         atom1[3],
         atom2[3],
         atom3[3],
         vector1[3],
         vector2[3],
         avgDelta,
         angle;
int      resCount,
         caCount,
         numCA = 0,
         numCalc = 0,
         found,
         linear;

for(resCount = 0; resCount < gNumRes; ++resCount)
{
   FindAtom(resCount, "CA", atom1, &found);
   if(found)
      ++numCA;
}

caAtoms = (caAtom *) malloc(numCA * sizeof(caAtom));
if(caAtoms == (caAtom *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "CalcResPT\n");
   fprintf(stderr, "\n");
   exit(1);
}

linear = TRUE;
caCount = 0;
for(resCount = 0; resCount < gNumRes; ++resCount)
{
   FindAtom(resCount, "CA", caAtoms[caCount].ca, &found);
   if(found)
   {
      if(caCount > 0)
         if(caAtoms[caCount - 1].resNum + 1 != resCount)
            linear = FALSE;
      caAtoms[caCount].resNum = resCount;
      ++caCount;
   }
   if(!linear)
      break;
}

if(numCA < 3)
{
   fprintf(stdout, "\n***Need at least three CA.***\n");
   fprintf(stdout, "***I cannot calculate Residues Per Turn.***\n\n");
   return(0);
```

```
}
if(!linear)
{
   fprintf(stdout, "\n***CA must be continuous.***\n");
   fprintf(stdout, "***I cannot calculate Residues Per Turn.***\n\n");
   return(0);
}

fprintf(stdout, "\n");
fprintf(stdout, "%33s %15s\n", "DeltaZ", "DeltaZ");
fprintf(stdout, "%6s %9s", "Res #", "CA[Z]");
fprintf(stdout, " %9s %7s", "Res-Res", "n-(n-1)");
fprintf(stdout, " %7s %7s", "Res-Res", "n-(n-2)");
fprintf(stdout, " %7s %5s\n", "Res/Trn", "Pitch");
fprintf(stdout, "%6s %7s", "--------", "-----");
fprintf(stdout, " %9s %7s", "-------", "-------");
fprintf(stdout, " %7s %7s", "-------", "-------");
fprintf(stdout, " %7s %5s\n", "-------", "-----");

caCount = 1;
resCount = caAtoms[caCount].resNum - 1;
fprintf(stdout, "%-4s%3d%1s %7.2f\n",
      gAtoms[gResidues[resCount].begin].resName,
      gAtoms[gResidues[resCount].begin].resSeqNum,
      gAtoms[gResidues[resCount].begin].insertRes,
      caAtoms[caCount - 1].ca[Z]);

for(caCount = 1; caCount < numCA - 1; ++caCount)
{
   resCount = caAtoms[caCount].resNum;

   atom1[X] = caAtoms[caCount - 1].ca[X];
   atom1[Y] = caAtoms[caCount - 1].ca[Y];
   atom1[Z] = caAtoms[caCount].ca[Z];

   atom2[X] = caAtoms[caCount].ca[X];
   atom2[Y] = caAtoms[caCount].ca[Y];
   atom2[Z] = caAtoms[caCount].ca[Z];

   atom3[X] = caAtoms[caCount + 1].ca[X];
   atom3[Y] = caAtoms[caCount + 1].ca[Y];
   atom3[Z] = caAtoms[caCount].ca[Z];

   Vectorize(atom2, atom1, vector1);
   Vectorize(atom2, atom3, vector2);

   angle = acos(DotProd(vector1, vector2));
   resPerTurn = (2.0 * gdPi) / (gdPi - fabs(angle));
   avgDelta = fabs(caAtoms[caCount + 1].ca[Z] -
         caAtoms[caCount - 1].ca[Z]) / 2.0;
   pitch = resPerTurn * avgDelta;

   totRPT += resPerTurn;
   totPitch += pitch;
   ++numCalc;

   fprintf(stdout, "%-4s%3d%1s %7.2f",
         gAtoms[gResidues[resCount].begin].resName,
         gAtoms[gResidues[resCount].begin].resSeqNum,
         gAtoms[gResidues[resCount].begin].insertRes,
         caAtoms[caCount].ca[Z]);
   fprintf(stdout, " %5i-%3i", caCount, caCount + 1);
```

```
      fprintf(stdout, " %6.2f",
            caAtoms[caCount].ca[Z] - caAtoms[caCount - 1].ca[Z]);
      if(caCount > 1)
      {
         fprintf(stdout, " %4i-%3i", caCount - 1, caCount + 1);
         fprintf(stdout, " %6.2f",
               caAtoms[caCount].ca[Z] - caAtoms[caCount - 2].ca[Z]);
         fprintf(stdout, " %6.1f %7.1f\n", resPerTurn, pitch);
      }
      else
         fprintf(stdout, " %22.1f %7.1f\n", resPerTurn, pitch);
   }

   caCount = numCA - 1;
   fprintf(stdout, "%-4s%3d%1s %7.2f",
         gAtoms[gResidues[resCount + 1].begin].resName,
         gAtoms[gResidues[resCount + 1].begin].resSeqNum,
         gAtoms[gResidues[resCount + 1].begin].insertRes,
         caAtoms[caCount].ca[Z]);
   fprintf(stdout, " %5i-%3i", caCount, caCount + 1);
   fprintf(stdout, " %6.2f",
         caAtoms[caCount].ca[Z] - caAtoms[caCount - 1].ca[Z]);
   if(((caCount + 1) % 2) != 0)
   {
      fprintf(stdout, " %4i-%3i", caCount - 1, caCount + 1);
      fprintf(stdout, " %6.2f\n",
            caAtoms[caCount].ca[Z] - caAtoms[caCount - 2].ca[Z]);
   }
   else
      fprintf(stdout, "\n");

   fprintf(stdout, "Average Residues/Turn: %5.1f\n", totRPT / numCalc);
   fprintf(stdout, "Average Pitch:         %5.1f\n", totPitch / numCalc);
   fprintf(stdout, "\n");

   free(caAtoms);
   return(1);
}

/*
File CalcTheta.c
Calculates the theta values needed for rotating the chemical shift tensor
from the molecular frame to the laboratory frame.
*/

#include "Torc.h"

void CalcTheta(double atom1[3],
               double atom2[3],
               double atom3[3],
               double theta[3])
{
   double   vec12[3],
            vec13[3],
            x[3],
            y[3],
            z[3],
            yprime[3],
            sinYGamma,
            cosYGamma,
            yGamma;
   Vectorize(atom1, atom2, vec12);
   Vectorize(atom1, atom3, vec13);
   CalcNorm(vec12, vec13, y);
   Normalize(y);

   z[X] = vec12[X];
   z[Y] = vec12[Y];
   z[Z] = vec12[Z];
   Normalize(z);

/*
   CalcNorm(y, z, x);
   Normalize(x);
*/

   yGamma = acos(z[Y]/sqrt(DSQR(z[X]) + DSQR(z[Y]))) * (double) SIGN(z[X]);
   sinYGamma = sin(yGamma);
   cosYGamma = cos(yGamma);

   yprime[X] = (y[X] * cosYGamma) - (y[Y] * sinYGamma);
/*
   yprime[Y] = (y[X] * sinYGamma) + (y[Y] * cosYGamma);
   yprime[Z] = y[Z];
*/

   theta[ONE] = acos(yprime[X]) * (double) SIGN(y[Z]);
   theta[TWO] = acos(z[Z]);
   theta[THREE] = yGamma + (90.0 * gdRadPerDeg);
}

/*
File CalcTorAngle.c
Calculates the torsion angle for four atoms.
*/

#include "Torc.h"

double CalcTorAngle(double atom1[3],
                    double atom2[3],
                    double atom3[3],
                    double atom4[3])
{
   double   vec21[3],
            vec23[3],
            vec34[3],
            norm123[3],
            norm234[3],
            crossNorm[3],
            angle,
            crossAngle;

   Vectorize(atom2, atom1, vec21);
   Vectorize(atom2, atom3, vec23);
   Vectorize(atom3, atom4, vec34);

   CalcNorm(vec23, vec21, norm123);
   CalcNorm(vec23, vec34, norm234);
   CalcNorm(norm123, norm234, crossNorm);

   angle = DotProd(norm123, norm234);
   if(fabs(angle) > 1.0)
      angle = -1.0;
   angle = acos(angle);

   crossAngle = DotProd(vec23, crossNorm);
   if(crossAngle < 0.0)
      return(-angle);
   else
      return(angle);
}

/*
File CalcTorsion.c
Calculates the (phi, psi, omega, x1, x2, x3) torsion angles of a peptide.
This function uses CalcTorAngle to actually calculate individual torsion
angles.
*/

#include "Torc.h"

void CalcTorsion(void)
{
   double   angle,
            atom1[3],
            atom2[3],
            atom3[3],
            atom4[3];
   int      residueCount = 0,
            found;

   fprintf(stdout, " Residue     Phi       Psi      Omega");
   fprintf(stdout, "       X1        X2        X3        X4\n");
   fprintf(stdout, "---------  -------   -------   -------");
   fprintf(stdout, "   -------   -------   -------   -------\n");

   fprintf(stdout, "%4s%4d%1s %8s",
         gAtoms[gResidues[residueCount].begin].resName,
         gAtoms[gResidues[residueCount].begin].resSeqNum,
         gAtoms[gResidues[residueCount].begin].insertRes,
         "--  ");

   for(residueCount = 0; residueCount < gNumRes; ++residueCount)
   {
      if(residueCount + 1 < gNumRes)
      {
         FindAtom(residueCount, "N", atom1, &found);
         if(found)
            FindAtom(residueCount, "CA", atom2, &found);
         if(found)
            FindAtom(residueCount, "C", atom3, &found);
         if(found)
            FindAtom(residueCount + 1, "N", atom4, &found);
         if(found)
         {
            angle = CalcTorAngle(atom1, atom2, atom3, atom4);
            fprintf(stdout, "  %8.2f", angle / gdRadPerDeg);
         }
         else
            fprintf(stdout, "%10s", "--  ");
      }
      else
         fprintf(stdout, "%10s", "--  ");

      if(residueCount + 1 < gNumRes)
```

```
{
   FindAtom(residueCount, "CA", atom1, &found);
   if(found)
      FindAtom(residueCount, "C", atom2, &found);
   if(found)
      FindAtom(residueCount + 1, "N", atom3, &found);
   if(found)
      FindAtom(residueCount + 1, "CA", atom4, &found);
   if(found)
   {
      angle = CalcTorAngle(atom1, atom2, atom3, atom4);
      fprintf(stdout, "%1s %8.2f", " ", angle / gdRadPerDeg);
   }
   else
      fprintf(stdout, "%10s", "--  ");
}
else
   fprintf(stdout, "%10s", "--  ");

FindAtom(residueCount, "N", atom1, &found);
if(found)
   FindAtom(residueCount, "CA", atom2, &found);
if(found)
   FindAtom(residueCount, "CB", atom3, &found);
if(found)
{
   FindAtom(residueCount, "CG", atom4, &found);
   if(!found)
      FindAtom(residueCount, "CG1", atom4, &found);
}
if(found)
{
   angle = CalcTorAngle(atom1, atom2, atom3, atom4);
   fprintf(stdout, "%1s %8.2f", " ", angle / gdRadPerDeg);
}
else
   fprintf(stdout, "%10s", "--  ");

FindAtom(residueCount, "CA", atom1, &found);
if(found)
   FindAtom(residueCount, "CB", atom2, &found);
if(found)
{
   FindAtom(residueCount, "CG", atom3, &found);
   if(!found)
      FindAtom(residueCount, "CG1", atom3, &found);
}
if(found)
{
   FindAtom(residueCount, "CD", atom4, &found);
   if(!found)
      FindAtom(residueCount, "CD1", atom4, &found);
}
if(found)
{
   angle = CalcTorAngle(atom1, atom2, atom3, atom4);
   fprintf(stdout, "%1s %8.2f", " ", angle / gdRadPerDeg);
}
else
   fprintf(stdout, "%10s", "--  ");

FindAtom(residueCount, "CB", atom1, &found);
```

```
      if(found)
      {
         FindAtom(residueCount, "CG", atom2, &found);
         if(!found)
            FindAtom(residueCount, "CG1", atom2, &found);
      }
      if(found)
      {
         FindAtom(residueCount, "CD", atom3, &found);
         if(!found)
            FindAtom(residueCount, "CD1", atom3, &found);
      }
      if(found)
      {
         FindAtom(residueCount, "NE", atom4, &found);
         if(!found)
            FindAtom(residueCount, "NE1", atom4, &found);
      }
      if(found)
      {
         angle = CalcTorAngle(atom1, atom2, atom3, atom4);
         fprintf(stdout, "%1s %8.2f\n", " ", angle / gdRadPerDeg);
      }
      else
         fprintf(stdout, "%10s\n", "--  ");

      if(residueCount + 1 < gNumRes)
      {
         fprintf(stdout, "%4s%4d%1s",
               gAtoms[gResidues[residueCount + 1].begin].resName,
               gAtoms[gResidues[residueCount + 1].begin].resSeqNum,
               gAtoms[gResidues[residueCount + 1].begin].insertRes);
         FindAtom(residueCount, "C", atom1, &found);
         if(found)
            FindAtom(residueCount + 1, "N", atom2, &found);
         if(found)
            FindAtom(residueCount + 1, "CA", atom3, &found);
         if(found)
            FindAtom(residueCount + 1, "C", atom4, &found);
         if(found)
         {
            angle = CalcTorAngle(atom1, atom2, atom3, atom4);
            fprintf(stdout, " %8.2f", angle / gdRadPerDeg);
         }
         else
            fprintf(stdout, " %8s", "--  ");
      }
   }
}

/*
File CalcTotP.c
Calculates the total penalty of the refined structure.
*/

#include "Torc.h"

double CalcTotP(int print)
{
   static int  printNum = 1;
   double   totPenalty = 0.0,
            penaltyNCS,
         penaltyCCS,
         penaltyICS,
         penaltyNC,
         penaltyNH,
         penaltyIH,
         penaltyDIS,
         penaltyCD;

#ifndef NORMTORC
   double   penaltyE;

   double   etorc_(double x[], double y[], double z[],
                   double fX[], double fY[], double fZ[]);
#endif

   if(print)
   {
      fprintf(stdout, "%17s  calc      obs      calc-obs   expError\n", " ");
      fprintf(stdout, "%17s--------   --------   --------   --------\n", " ");
   }

   penaltyNCS = CalcNCSP(print);
   totPenalty = gLambda[NCS] * penaltyNCS;
   penaltyCCS = CalcCCSP(print);
   totPenalty += gLambda[CCS] * penaltyCCS;
   penaltyICS = CalcICSP(print);
   totPenalty += gLambda[ICS] * penaltyICS;
   penaltyNC  = CalcNCP(print);
   totPenalty += gLambda[NC]  * penaltyNC;
   penaltyNH  = CalcNHP(print);
   totPenalty += gLambda[NH]  * penaltyNH;
   penaltyIH  = CalcIHP(print);
   totPenalty += gLambda[IH]  * penaltyIH;
   penaltyDIS = CalcDisP(print);
   totPenalty += gLambda[DIS] * penaltyDIS;
   penaltyCD  = CalcCDP(print);
   totPenalty += gLambda[CD]  * penaltyCD;
#ifndef NORMTORC
/* CHARMM is sslllooww. This is to speed up the dihedral refinement. */
   if(gLambda[E] == 0.0)
      penaltyE = 0.0;
   else
      penaltyE = etorc_(gXArray, gYArray, gZArray,
                        gXForce, gYForce, gZForce);

   gCalcE = penaltyE;
   totPenalty += (gLambda[E] * penaltyE);
#endif

   gCalcNCSP = gLambda[NCS] * penaltyNCS;
   gCalcCCSP = gLambda[CCS] * penaltyCCS;
   gCalcICSP = gLambda[ICS] * penaltyICS;
   gCalcNCP  = gLambda[NC] * penaltyNC;
   gCalcNHP  = gLambda[NH] * penaltyNH;
   gCalcIHP  = gLambda[IH] * penaltyIH;
   gCalcDISP = gLambda[DIS] * penaltyDIS;
   gCalcCDP  = gLambda[CD] * penaltyCD;
#ifndef NORMTORC
   gCalcEP   = gLambda[E] * penaltyE;
#endif
   gCalcTotP = totPenalty;
   if(print)
   {
      fprintf(stdout, ":%d         penalty     lambda    penalty\n",
            printNum);
      fprintf(stdout, ":%d  NCS %10.4f %10.4f %10.4f\n",
            printNum, penaltyNCS, gLambda[NCS], penaltyNCS * gLambda[NCS]);
      fprintf(stdout, ":%d  CCS %10.4f %10.4f %10.4f\n",
            printNum, penaltyCCS, gLambda[CCS], penaltyCCS * gLambda[CCS]);
      fprintf(stdout, ":%d  ICS %10.4f %10.4f %10.4f\n",
            printNum, penaltyICS, gLambda[ICS], penaltyICS * gLambda[ICS]);
      fprintf(stdout, ":%d  NC  %10.4f %10.4f %10.4f\n",
            printNum, penaltyNC, gLambda[NC], penaltyNC * gLambda[NC]);
      fprintf(stdout, ":%d  NH  %10.4f %10.4f %10.4f\n",
            printNum, penaltyNH, gLambda[NH], penaltyNH * gLambda[NH]);
      fprintf(stdout, ":%d  IH  %10.4f %10.4f %10.4f\n",
            printNum, penaltyIH, gLambda[IH], penaltyIH * gLambda[IH]);
      fprintf(stdout, ":%d  DIS %10.4f %10.4f %10.4f\n",
            printNum, penaltyDIS, gLambda[DIS], penaltyDIS * gLambda[DIS]);
      fprintf(stdout, ":%d  CD  %10.4f %10.4f %10.4f\n",
            printNum, penaltyCD, gLambda[CD], penaltyCD * gLambda[CD]);
#ifndef NORMTORC
      fprintf(stdout, ":%d  E   %10.4f %10.4f %10.4f\n",
            printNum, penaltyE, gLambda[E], penaltyE * gLambda[E]);
#endif
      fprintf(stdout, ":%d                              -----------\n",
            printNum);
      fprintf(stdout, ":%d  total                        %10.4f\n",
            printNum, totPenalty);
      fprintf(stdout, ":%d\n\n", printNum);
      ++printNum;
   }

   return(totPenalty);
}

/*
File CopyCoord.c
Copies a molecule.
*/

#include "Torc.h"

void CopyCoord(atom  *atomsIn,
               atom  *atomsCopy)
{
   int   atomCount;

   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
      atomsCopy[atomCount] = atomsIn[atomCount];
}

/*
File CopyXYZ.c
Copies the XYZ coordinates of a molecule.
*/

#include "Torc.h"

void CopyXYZ(atom  *atomsIn,
             atom  *atomsCopy)
{
   int   atomCount;
   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
   {
      atomsCopy[atomCount].coords[X] = atomsIn[atomCount].coords[X];
      atomsCopy[atomCount].coords[Y] = atomsIn[atomCount].coords[Y];
      atomsCopy[atomCount].coords[Z] = atomsIn[atomCount].coords[Z];
   }
}

/*
File DeclArray.c
Declares the necessary memory for the global arrays.
*/

#include "Torc.h"

void DeclArray(void)
{
   int   count;

   gAtomsInitial = (atom *) malloc(gNumAtoms * sizeof(atom));
   if(gAtomsInitial == (atom *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "DeclArray - Setting up gAtomsInitial\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   gAtomsLowest = (atom *) malloc(gNumAtoms * sizeof(atom));
   if(gAtomsLowest == (atom *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "DeclArray - Setting up gAtomsLowest\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   gAtomsLastState = (atom *) malloc(gNumAtoms * sizeof(atom));
   if(gAtomsLastState == (atom *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "DeclArray - Setting up gAtomsLastState\n");
      fprintf(stderr, "\n");
      exit(1);
   }

#ifndef NORMTORC
   gXArray = (double *) malloc(2 * gNumAtoms * sizeof(double));
   if(gXArray == (double *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "DeclArray - Setting up gXArray\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   gYArray = (double *) malloc(2 * gNumAtoms * sizeof(double));
```

```
if(gYArray == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gYArray\n");
   fprintf(stderr, "\n");
   exit(1);
}

gZArray = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gZArray == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gZArray\n");
   fprintf(stderr, "\n");
   exit(1);
}

gXForce = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gXForce == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gXForce\n");
   fprintf(stderr, "\n");
   exit(1);
}

gYForce = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gYForce == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gYForce\n");
   fprintf(stderr, "\n");
   exit(1);
}

gZForce = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gZForce == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gZForce\n");
   fprintf(stderr, "\n");
   exit(1);
}

gXForceOldState = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gXForceOldState == (double *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "DeclArray - Setting up gXForceOldState\n");
   fprintf(stderr, "\n");
   exit(1);
}

gYForceOldState = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gYForceOldState == (double *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gYForceOldState\n");
    fprintf(stderr, "\n");
    exit(1);
}

gZForceOldState = (double *) malloc(2 * gNumAtoms * sizeof(double));
if(gZForceOldState == (double *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gZForceOldState\n");
    fprintf(stderr, "\n");
    exit(1);
}

gNumMoveCompAtt = (int *) malloc(gNumRes * sizeof(int));
if(gNumMoveCompAtt == (int *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gNumMoveCompAtt\n");
    fprintf(stderr, "\n");
    exit(1);
}

gNumMoveCompAcc = (int *) malloc(gNumRes * sizeof(int));
if(gNumMoveCompAcc == (int *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gNumMoveCompAcc\n");
    fprintf(stderr, "\n");
    exit(1);
}

gNumMoveTunnAtt = (int *) malloc(gNumRes * sizeof(int));
if(gNumMoveTunnAtt == (int *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gNumMoveTunnAtt\n");
    fprintf(stderr, "\n");
    exit(1);
}

gNumMoveTunnAcc = (int *) malloc(gNumRes * sizeof(int));
if(gNumMoveTunnAcc == (int *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
    fprintf(stderr, "DeclArray - Setting up gNumMoveTunnAcc\n");
    fprintf(stderr, "\n");
    exit(1);
}

gNumMoveTorsAtt = (int *) malloc(gNumRes * sizeof(int));
if(gNumMoveTorsAtt == (int *) NULL)
{
    fprintf(stderr, "\n");
    fprintf(stderr, "out of memory!\n");
```

```
      fprintf(stderr, "DeclArray - Setting up gNumMoveTorsAtt\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   gNumMoveTorsAcc = (int *) malloc(gNumRes * sizeof(int));
   if(gNumMoveTorsAcc == (int *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "DeclArray - Setting up gNumMoveTorsAcc\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   for(count = 0; count < gNumRes; ++count)
   {
      gNumMoveCompAtt[count] = 0;
      gNumMoveCompAcc[count] = 0;
      gNumMoveTunnAtt[count] = 0;
      gNumMoveTunnAcc[count] = 0;
      gNumMoveTorsAtt[count] = 0;
      gNumMoveTorsAcc[count] = 0;
   }
#endif
}

/*
File DoRotation.c
Rotates an atom about a defined origin.
*/

#include "Torc.h"

void DoRotation(double rotMat[3][3],
                double atomToRotate[3],
                double origin[3])
{
   double   xx,
            xy,
            xz,
            yx,
            yy,
            yz;

   xx = (rotMat[X][X] - 1.0) * origin[X]
       + rotMat[X][Y]        * origin[Y]
       + rotMat[X][Z]        * origin[Z];
   xy =  rotMat[Y][X]        * origin[X]
      + (rotMat[Y][Y] - 1.0) * origin[Y]
       + rotMat[Y][Z]        * origin[Z];
   xz =  rotMat[Z][X]        * origin[X]
       + rotMat[Z][Y]        * origin[Y]
      + (rotMat[Z][Z] - 1.0) * origin[Z];

   yx = rotMat[X][X] * atomToRotate[X]
      + rotMat[X][Y] * atomToRotate[Y]
      + rotMat[X][Z] * atomToRotate[Z];
   yy = rotMat[Y][X] * atomToRotate[X]
      + rotMat[Y][Y] * atomToRotate[Y]
      + rotMat[Y][Z] * atomToRotate[Z];
   yz = rotMat[Z][X] * atomToRotate[X]
```

```
      + rotMat[Z][Y] * atomToRotate[Y]
      + rotMat[Z][Z] * atomToRotate[Z];

   atomToRotate[X] = yx - xx;
   atomToRotate[Y] = yy - xy;
   atomToRotate[Z] = yz - xz;
}

/*
File DotProd.c
Calculates the dot product between two vectors.
*/

#include "Torc.h"

double DotProd(double vec1[3],
               double vec2[3])
{
   double   comp,
            mag1,
            mag2,
            result;

   comp = (vec1[X] * vec2[X]) + (vec1[Y] * vec2[Y]) + (vec1[Z] * vec2[Z]);
   mag1 = sqrt(DSQR(vec1[X]) + DSQR(vec1[Y]) + DSQR(vec1[Z]));
   mag2 = sqrt(DSQR(vec2[X]) + DSQR(vec2[Y]) + DSQR(vec2[Z]));
   result = comp / (mag1 * mag2);

   return(result);
}

/*
File EulerMatrix.c
Calculates a rotation matrix from q.
*/

#include "Torc.h"

void EulerMatrix(double q[4],
                 double rotMatrix[3][3])
{
   double   q0q0,
            q1q1,
            q2q2,
            q3q3;

   q0q0 = DSQR(q[0]);
   q1q1 = DSQR(q[1]);
   q2q2 = DSQR(q[2]);
   q3q3 = DSQR(q[3]);

   rotMatrix[X][X] = q0q0 + q1q1 - q2q2 - q3q3;
   rotMatrix[Y][X] = 2.0 * (q[1] * q[2] - q[0] * q[3]);
   rotMatrix[Z][X] = 2.0 * (q[1] * q[3] + q[0] * q[2]);

   rotMatrix[X][Y] = 2.0 * (q[2] * q[1] + q[0] * q[3]);
   rotMatrix[Y][Y] = q0q0 - q1q1 + q2q2 - q3q3;
   rotMatrix[Z][Y] = 2.0 * (q[2] * q[3] - q[0] * q[1]);

   rotMatrix[X][Z] = 2.0 * (q[3] * q[1] - q[0] * q[2]);
   rotMatrix[Y][Z] = 2.0 * (q[3] * q[2] + q[0] * q[1]);
   rotMatrix[Z][Z] = q0q0 - q1q1 - q2q2 + q3q3;
```

```
}

/*
File FindAtom.c
Finds atomic coordinates within a residue.
*/

#include "Torc.h"
#include <string.h>

void FindAtom(int residueNumber,
              char atomToFind[],
              double coords[3],
              int *found)
{
   int   atomCount;

   if((residueNumber + 1) > gNumRes)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "***Operator Error***\n");
      fprintf(stderr, "residueNumber %d > %d residues\n",
            residueNumber + 1,
            gNumRes);
      fprintf(stderr, "FindAtom\n");
      fprintf(stderr, "\n");
      exit(1);
   }
   if(residueNumber < 0)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "***Operator Error***\n");
      fprintf(stderr, "residueNumber %d < 0\n", residueNumber);
      fprintf(stderr, "FindAtom\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   atomCount = gResidues[residueNumber].begin;
   while((strcmp(gAtoms[atomCount].atomName, atomToFind) != 0) &&
         (atomCount < gResidues[residueNumber].end))
      ++atomCount;
   if(strcmp(gAtoms[atomCount].atomName, atomToFind) != 0)
      *found = FALSE;
   else
   {
      coords[X] = gAtoms[atomCount].coords[X];
      coords[Y] = gAtoms[atomCount].coords[Y];
      coords[Z] = gAtoms[atomCount].coords[Z];
      *found = TRUE;
   }
}

/*
File GaussRand.c
Generate a gaussian random number.

This function generates a Gaussian random deviate of 0.0 mean and standard
deviation width. The algorithm is based on Box and Muller and is found in
Numerical Recipes in C, 2nd ed., p 289-290.
*/
```

```
#include "Torc.h"

double GaussRand(double width)
{
   static int     gotExtraDev = FALSE;
   static double  extraDev;
   double         random1,
                  random2,
                  rsq,
                  fac;

   if(!gotExtraDev)
   {
      do
      {
         random1 = 2.0 * ((double) random() / gMaxRand) - 1.0;
         random2 = 2.0 * ((double) random() / gMaxRand) - 1.0;
         rsq = DSQR(random1) + DSQR(random2);
      }
      while(rsq >= 1.0 || rsq == 0.0);

      fac = sqrt(-2.0 * log(rsq) / rsq);
      extraDev = width * random1 * fac;
      gotExtraDev = TRUE;

      return(width * random2 * fac);
   }
   else
   {
      gotExtraDev = FALSE;
      return(extraDev);
   }
}

/*
File GetMag.c
Returns a random nonzero magnitude between and including two limits.
*/

#include "Torc.h"

#define ldLowLimit 0.01

double GetMag(double low,
              double high)
{
   double   mag,
            range,
            rpRM,
            small,
            lowest;

   small = fabs(high);
   if(fabs(low) < small)
      small = fabs(low);
   lowest = small * ldLowLimit;

   range = high - low;
   rpRM = range / gMaxRand;
   do
   {
      mag = low + (rpRM * random());
   }
   while(fabs(mag) < lowest);
/*
   while(mag == 0.0);
*/

   return(mag);
}
#undef ldLowLimit

/*
File MF_LF.c
Rotates the chemical shift tensor from the molecular frame to the laboratory
frame.
*/

#include "Torc.h"

void MF_LF(double sigmaMF[3][3],
           double theta[3],
           double sigmaLF[3][3])
{
   double   rt1[3][3],
            rt1t[3][3],
            rt2[3][3],
            rt2t[3][3],
            rt3[3][3],
            rt3t[3][3],
            result1[3][3],
            result2[3][3],
            result3[3][3],
            result4[3][3],
            result5[3][3];

   rt1[X][X] = cos(theta[ONE]);
   rt1[X][Y] = sin(theta[ONE]);
   rt1[X][Z] = 0.0;
   rt1[Y][X] = -sin(theta[ONE]);
   rt1[Y][Y] = cos(theta[ONE]);
   rt1[Y][Z] = 0.0;
   rt1[Z][X] = 0.0;
   rt1[Z][Y] = 0.0;
   rt1[Z][Z] = 1.0;

   Transpose(rt1, rt1t);

   rt2[X][X] = cos(theta[TWO]);
   rt2[X][Y] = 0.0;
   rt2[X][Z] = -sin(theta[TWO]);
   rt2[Y][X] = 0.0;
   rt2[Y][Y] = 1.0;
   rt2[Y][Z] = 0.0;
   rt2[Z][X] = sin(theta[TWO]);
   rt2[Z][Y] = 0.0;
   rt2[Z][Z] = cos(theta[TWO]);

   Transpose(rt2, rt2t);

   rt3[X][X] = cos(theta[THREE]);
   rt3[X][Y] = sin(theta[THREE]);
   rt3[X][Z] = 0.0;
   rt3[Y][X] = -sin(theta[THREE]);
   rt3[Y][Y] = cos(theta[THREE]);
   rt3[Y][Z] = 0.0;
   rt3[Z][X] = 0.0;
   rt3[Z][Y] = 0.0;
   rt3[Z][Z] = 1.0;

   Transpose(rt3, rt3t);

   MatMult(rt3, rt2, result1);
   MatMult(result1, rt1, result2);
   MatMult(result2, sigmaMF, result3);
   MatMult(result3, rt1t, result4);
   MatMult(result4, rt2t, result5);
   MatMult(result5, rt3t, sigmaLF);
}

/*
File MakeXYZArray.c
Transfers the atomic coordinates into x, y and z arrays for use in charmm.
*/

#include "Torc.h"

void MakeXYZArray(atom *atomsIn,
                  double *x,
                  double *y,
                  double *z)
{
   int   atomCount;

   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
   {
      x[atomCount] = atomsIn[atomCount].coords[X];
      y[atomCount] = atomsIn[atomCount].coords[Y];
      z[atomCount] = atomsIn[atomCount].coords[Z];
   }
}

/*
File MatMult.c
Multiplies two 3x3 matrices.
*/

#include "Torc.h"

void MatMult(double matrix1[3][3],
             double matrix2[3][3],
             double result[3][3])
{
   int   row,
         col;

   for(row = X; row <= Z; ++row)
      for(col = X; col <= Z; ++col)
         result[row][col] = matrix1[row][X] * matrix2[X][col]
                          + matrix1[row][Y] * matrix2[Y][col]
                          + matrix1[row][Z] * matrix2[Z][col];
}

/*
File Metropolis.c
Determines whether a given change in penalty is acceptable.
```

```
*/

#include "Torc.h"

int Metropolis(double dPenalty)
{
   double   kbt,
            randomNum;

   kbt = gTemperature * gdBoltzmann;

   randomNum = (double) random() / gMaxRand;

   if((dPenalty <= 0.0) || (randomNum < exp(-dPenalty / kbt)))
      return(TRUE);
   else
      return(FALSE);
}

/*
File MotavgTensor.c
Motionally averages the chemical shift tensor.
*/

#include "Torc.h"

#define ldMotAvgRes  100.0

void MotavgTensor(double tensorIn[3][3],
                  double motAvgBond[3],
                  double delta,
                  double tensorOut[3][3])
{
   double   theta,
            temp1,
            temp2,
            motRes,
            prob,
            probSum = 0.0,
            result[3][3],
            q[4],
            rotMat[3][3],
            rotMatT[3][3],
            tensorSum[3][3],
            tensorOutProb[3][3],
            tensorOutTotal[3][3];
   int      row,
            col;

   for(row = X; row <= Z; ++row)
      for(col = X; col <= Z; ++col)
         tensorSum[row][col] = 0.0;
   motRes = 4.0 * delta / ldMotAvgRes;
   for(theta = (-2.0 * delta); theta <= (2.0 * delta); theta += motRes)
   {
      q[0] = cos(theta * 0.5);
      q[1] = motAvgBond[X] * sin(theta * 0.5);
      q[2] = motAvgBond[Y] * sin(theta * 0.5);
      q[3] = motAvgBond[Z] * sin(theta * 0.5);
      EulerMatrix(q, rotMat);
      Transpose(rotMat, rotMatT);
      MatMult(rotMat, tensorIn, result);
      MatMult(result, rotMatT, tensorOut);

      temp1 = DSQR(theta);
      temp2 = 1.4427 * DSQR(delta);
      prob = exp(-temp1 / temp2);
      probSum += prob;

      for(row = X; row <= Z; ++row)
         for(col = X; col <= Z; ++col)
            tensorOutProb[row][col] = prob * tensorOut[row][col];
      for(row = X; row <= Z; ++row)
         for(col = X; col <= Z; ++col)
            tensorOutTotal[row][col] = tensorSum[row][col]
                                    + tensorOutProb[row][col];
      for(row = X; row <= Z; ++row)
         for(col = X; col <= Z; ++col)
            tensorSum[row][col] = tensorOutTotal[row][col];
   }
   for(row = X; row <= Z; ++row)
      for(col = X; col <= Z; ++col)
         tensorOut[row][col] = tensorSum[row][col] / probSum;
}
#undef ldMotAvgRes

/*
File MoveAtoms.c
Move all of the atoms.

kbT in [kcal/mol]= kB*T*A/4.18/1000 = 1.38E-23*300*6.02E23/4180.
                 = 0.5962393  kcal/mol at room temperature (300K)
kb               = 1.9874643E-03 in kcal/mol/K
*/

#include "Torc.h"

void MoveAtoms(atom *atomsIn,
               double diffusion)
{
   int      atomCount;
   double   atomMove[3],
            translation[3],
            width,
            kbt,
            moveFactor;

/*
   width = 2.0 * diffusion;
   kbt = gTemperature * gdBoltzmann;
   moveFactor = diffusion / kbt;
*/

/*
Apply a small global translation to aid in dimerization. This is cheating.
*/
   translation[X] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;
   translation[Y] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;
   translation[Z] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;

   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
   {
/*
Make moves based on the atomic force vectors and diffusion parameter.
```

```
*/
/*
      atomMove[X] = (moveFactor * gXForce[atomCount]);
      atomMove[Y] = (moveFactor * gYForce[atomCount]);
      atomMove[Z] = (moveFactor * gZForce[atomCount]);
*/

/*
Make moves based on the atomic force vectors, diffusion parameter, and a
gaussian random deviation.
*/
/*
      atomMove[X] = (moveFactor * gXForce[atomCount]) + GaussRand(width);
      atomMove[Y] = (moveFactor * gYForce[atomCount]) + GaussRand(width);
      atomMove[Z] = (moveFactor * gZForce[atomCount]) + GaussRand(width);
*/

/*
Make moves based on the move factor and a gaussian random deviation.
*/
/*
      atomMove[X] = moveFactor + GaussRand(width);
      atomMove[Y] = moveFactor + GaussRand(width);
      atomMove[Z] = moveFactor + GaussRand(width);
*/

/*
Make moves based on the diffusion parameter and a gaussian random deviation.
*/
/*
      atomMove[X] = GaussRand(diffusion);
      atomMove[Y] = GaussRand(diffusion);
      atomMove[Z] = GaussRand(diffusion);
*/

/*
Make moves based on the diffusion parameter and a random deviation.
*/
      atomMove[X] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;
      atomMove[Y] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;
      atomMove[Z] = (2.0 * ((double) random() / gMaxRand) - 1.0) * diffusion;

      atomsIn[atomCount].coords[X] += (atomMove[X] + translation[X]);
      atomsIn[atomCount].coords[Y] += (atomMove[Y] + translation[Y]);
      atomsIn[atomCount].coords[Z] += (atomMove[Z] + translation[Z]);
   }
}

/*
File MoveCompensate.c
Performs a compensating (peptide plane) move.
*/

#include "Torc.h"

int MoveCompensate(int *resNum)
{
   int      limitLineNum,
            rotateValue;
   double   mag;

   do
   {
      limitLineNum = (int)
            (gNumLimits * random() / (gMaxRand + 1.0));
   }
   while(strcmp(gLimits[limitLineNum].bondName, "phi") != 0 &&
         strcmp(gLimits[limitLineNum].bondName, "psi") != 0);

   mag = GetMag(gLimits[limitLineNum].low,
         gLimits[limitLineNum].high) * gdRadPerDeg;
   rotateValue = RotateAtoms(gAtoms,
                             gLimits[limitLineNum].resNum,
                             gLimits[limitLineNum].bondName,
                             mag);
   if(rotateValue == 0)
      return(0);

   rotateValue = 0;
   if(strcmp(gLimits[limitLineNum].bondName, "phi") == 0)
      rotateValue = RotateAtoms(gAtoms,
                                gLimits[limitLineNum].resNum - 1,
                                "psi",
                                -mag);
   if(strcmp(gLimits[limitLineNum].bondName, "psi") == 0)
      rotateValue = RotateAtoms(gAtoms,
                                gLimits[limitLineNum].resNum + 1,
                                "phi",
                                -mag);
   if(rotateValue == 0)
   {
      RotateAtoms(gAtoms,
                  gLimits[limitLineNum].resNum,
                  gLimits[limitLineNum].bondName,
                  -mag);
      return(0);
   }

   *resNum = gLimits[limitLineNum].resNum;

   return(1);
}

/*
File MoveTunnel.c
Performs a tunnel (peptide plane) move based on the magnitude of the
carbonyl orientation with respect to the channel axis.
*/

#include "Torc.h"

#define ldNoPrint    0
#define ldYesPrint   1

int MoveTunnel(int *resNum)
{
   int      limitLineNum,
            rotateValue,
            found;
   double   atom1[3],
            atom2[3],
            atom3[3],
            atomZ[3],
            vec1Z[3],
          vec12[3],
          vec13[3],
          normal[3],
          mag,
          planeAngle,
          bondAngle;

   do
   {
      limitLineNum = (int)
            (gNumLimits * random() / (gMaxRand + 1.0));
   }
   while(strcmp(gLimits[limitLineNum].bondName, "phi") != 0 &&
         strcmp(gLimits[limitLineNum].bondName, "psi") != 0);

   FindAtom(gLimits[limitLineNum].resNum - 1, "C", atom1, &found);
   if(found)
      FindAtom(gLimits[limitLineNum].resNum - 1, "O", atom2, &found);
   if(found)
      FindAtom(gLimits[limitLineNum].resNum - 1, "CA", atom3, &found);

   if(!found)
      return(0);

   atomZ[X] = atom1[X];
   atomZ[Y] = atom1[Y];
   atomZ[Z] = atom1[Z] + 1.0;
   Vectorize(atom1, atomZ, vec1Z);
   Vectorize(atom1, atom2, vec12);
   Vectorize(atom1, atom3, vec13);
   CalcNorm(vec12, vec13, normal);

   planeAngle = 90.0 - acos(DotProd(normal, vec1Z)) / gdRadPerDeg;
   bondAngle = acos(DotProd(vec12, vec1Z)) / gdRadPerDeg;

   mag = 2.0 * gdRadPerDeg * planeAngle;
   if(bondAngle > 90.0)
      mag = -mag;

   rotateValue = RotateAtoms(gAtoms,
                             gLimits[limitLineNum].resNum,
                             "psi",
                             mag);
   if(rotateValue == 0)
      return(0);
   rotateValue = RotateAtoms(gAtoms,
                             gLimits[limitLineNum].resNum + 1,
                             "phi",
                             -mag);
   if(rotateValue == 0)
   {
      RotateAtoms(gAtoms,
                  gLimits[limitLineNum].resNum,
                  "psi",
                  -mag);
      return(0);
   }

   *resNum = gLimits[limitLineNum].resNum;

   return(1);
}
```

```
/*
File Normalize.c
Normalizes a vector.
*/

#include "Torc.h"

void Normalize(double vector[3])
{
   double   length;

   length = sqrt(DSQR(vector[X]) + DSQR(vector[Y]) + DSQR(vector[Z]));
   if(length != 0.0)
   {
      vector[X] = vector[X] / length;
      vector[Y] = vector[Y] / length;
      vector[Z] = vector[Z] / length;
   }
   else
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "length == 0!  No normalization!\n");
      fprintf(stderr, "Normalize\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}

/*
File OutputPDB.c
Outputs the coordinates to disk.
*/

#include "Torc.h"
#include <string.h>
#include <time.h>
#include <sys/times.h>

void OutputPDB(char *fileName,
               atom *outAtoms)
{
   int   atomCount,
         digit;
   struct tm   *tmPointer;
   time_t   localTime;
   FILE  *outputFile;

   if((outputFile = fopen(fileName, "w")) == NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "File '%s' cannot be opened\n", fileName);
      fprintf(stderr, "Cannot save coordinates\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   fprintf(outputFile, "REMARK   1 Structure refined with:\n");
   fprintf(outputFile, "REMARK   1 %s\n", gdVersion);
   localTime = time('\0');
   tmPointer = localtime(&localTime);
   fprintf(outputFile, "REMARK   1 %s", asctime(tmPointer));
   fprintf(outputFile, "REMARK   2 Randal R. Ketchem\n");
   fprintf(outputFile, "REMARK   2 Institute of Molecular Biophysics");
   fprintf(outputFile, "   904.644.1309 (voice)\n");
   fprintf(outputFile, "REMARK   2 Florida State University");
   fprintf(outputFile, "            904.644.1366 (FAX)\n");
   fprintf(outputFile, "REMARK   2 Tallahassee, FL 32306-3015");
   fprintf(outputFile, "          rrk@magnet.fsu.edu (email)\n");


   for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
   {
      fprintf(outputFile, "%-6s",      outAtoms[atomCount].header);
      fprintf(outputFile, "%5d",       outAtoms[atomCount].atomSeqNum);
#ifdef NORMTORC
      digit = outAtoms[atomCount].atomName[0] - '0';
      if((digit <= 9) && (digit >= 0))
#else
      digit = strlen(outAtoms[atomCount].atomName);
      if(digit == 4)
#endif
         fprintf(outputFile, " %-4s",  outAtoms[atomCount].atomName);
      else
         fprintf(outputFile, "  %-3s", outAtoms[atomCount].atomName);
      fprintf(outputFile, "%1s",       outAtoms[atomCount].altLocInd);
      fprintf(outputFile, "%-4s",      outAtoms[atomCount].resName);
      fprintf(outputFile, "%1s",       outAtoms[atomCount].chainIdent);
      fprintf(outputFile, "%4d",       outAtoms[atomCount].resSeqNum);
      fprintf(outputFile, "%1s",       outAtoms[atomCount].insertRes);
      fprintf(outputFile, "   %8.3f",  outAtoms[atomCount].coords[X]);
      fprintf(outputFile, "%8.3f",     outAtoms[atomCount].coords[Y]);
      fprintf(outputFile, "%8.3f",     outAtoms[atomCount].coords[Z]);
      fprintf(outputFile, "%6.2f",     outAtoms[atomCount].occupancy);
      fprintf(outputFile, "%6.2f",     outAtoms[atomCount].tempFactor);
      if(outAtoms[atomCount].footnoteNum == -1)
         fprintf(outputFile, " %3s",   " ");
      else
         fprintf(outputFile, " %3d",   outAtoms[atomCount].footnoteNum);
#ifndef NORMTORC
      fprintf(outputFile, "  MONO");
#endif
      fprintf(outputFile, "\n");
   }
   fprintf(outputFile, "TER\n");
   fprintf(outputFile, "END\n");

   fclose(outputFile);
}

/*
File PAS_MF.c
Rotates the chemical shift tensor from the principal axis system to the
molecular frame.
*/

#include "Torc.h"

void PAS_MF(double sigmaPAS[3][3],
            double alpha,
            double beta,
            double sigmaMF[3][3])
{
   double   rad[3][3],
             radt[3][3],
             rbd[3][3],
             rbdt[3][3],
             result1[3][3],
             result2[3][3],
             result3[3][3];

   rad[X][X] = cos(alpha);
   rad[X][Y] = sin(alpha);
   rad[X][Z] = 0.0;
   rad[Y][X] = -sin(alpha);
   rad[Y][Y] = cos(alpha);
   rad[Y][Z] = 0.0;
   rad[Z][X] = 0.0;
   rad[Z][Y] = 0.0;
   rad[Z][Z] = 1.0;

   Transpose(rad, radt);

   rbd[X][X] = cos(beta);
   rbd[X][Y] = 0.0;
   rbd[X][Z] = sin(beta);
   rbd[Y][X] = 0.0;
   rbd[Y][Y] = 1.0;
   rbd[Y][Z] = 0.0;
   rbd[Z][X] = -sin(beta);
   rbd[Z][Y] = 0.0;
   rbd[Z][Z] = cos(beta);

   Transpose(rbd, rbdt);

   MatMult(rbd, rad, result1);
   MatMult(result1, sigmaPAS, result2);
   MatMult(result2, radt, result3);
   MatMult(result3, rbdt, sigmaMF);
}

/*
File PrintHelp.c
Prints help for running TORC.
*/

#include "Torc.h"

void PrintHelp(void)
{
   fprintf(stdout, "\n");
   fprintf(stdout, "This program will attempt to modify a peptide ");
   fprintf(stdout, "structure in order to\n");
   fprintf(stdout, "minimize the deviation between the calculated ");
   fprintf(stdout, "and experimental\n");
   fprintf(stdout, "constraints.\n");
   fprintf(stdout, "\n");
   fprintf(stdout, "The penalty is calculated as:\n");
   fprintf(stdout, "0.5 * ((calc - obs) / expError)^2\n");
   fprintf(stdout, "The summed penalty for each constraint type ");
   fprintf(stdout, "is then multiplied by a\n");
   fprintf(stdout, "weighting factor (normally 1).\n");
   fprintf(stdout, "\n");
   fprintf(stdout, "The constraints are calculated from the ");
   fprintf(stdout, "coordinates and used to calculate\n");
   fprintf(stdout, "the penalty between the calculated constraints ");
   fprintf(stdout, "and the experimental\n");
   fprintf(stdout, "constraints. The program attempts to minimize ");
   fprintf(stdout, "the total penalty by\n");
   fprintf(stdout, "introducing random conformational changes, ");
   fprintf(stdout, "accepting changes based on\n");
   fprintf(stdout, "simulated annealing.\n");
   fprintf(stdout, "\n");
   fprintf(stdout, "To use this program you must supply several ");
   fprintf(stdout, "input files. Create an empty\n");
   fprintf(stdout, "directory and run the program using that ");
   fprintf(stdout, "directory name. The program will\n");
   fprintf(stdout, "let you know each file it needs and the format ");
   fprintf(stdout, "of each file.\n");
   fprintf(stdout, "\n");
   fprintf(stdout, "The final coordinates are saved as ");
   fprintf(stdout, "dir/coord.out.pdb.\n");
   fprintf(stdout, "\n");
   fprintf(stdout, "Randal R. Ketchem\n");
   fprintf(stdout, "Institute of Molecular Biophysics   ");
   fprintf(stdout, "904.644.1309 (voice)\n");
   fprintf(stdout, "Florida State University            ");
   fprintf(stdout, "904.644.1366 (FAX)\n");
   fprintf(stdout, "Tallahassee, FL 32306-3015          ");
   fprintf(stdout, "rrk@magnet.fsu.edu (email)\n");
   fprintf(stdout, "\n");

   exit(1);
}

/*
File ReadCDData.c
Reads the CD quadrupole splitting data file.
*/

#include "Torc.h"
#include <string.h>

int ReadCDData(char fileName[])
{
   FILE  *cdFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0;

   if((cdFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "resNum cName dName quad qcc expError\n");
      fprintf(stdout, "x      x     x     x.x  x.x x.x\n");
      fprintf(stdout, "x      x     x     x.x  x.x x.x\n");
      fprintf(stdout, "etc.\n");
      fprintf(stdout, "(include the header line)\n");
      fprintf(stdout, "\n");
      return(0);
   }

   gNumCDData = 0;
   while(fgets(aLine, gdLineLength, cdFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
         ++gNumCDData;
   }
   rewind(cdFile);

   gCDDatas = (cdData *) malloc(gNumCDData * sizeof(cdData));
   if(gCDDatas == (cdData *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "ReadCDData\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   while(fgets(aLine, gdLineLength, cdFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      {
         sscanf(aLine, "%d %s %s %lf %lf %lf",
               &gCDDatas[count].resNum,
               &gCDDatas[count].atom1Name,
               &gCDDatas[count].atom2Name,
               &gCDDatas[count].quad,
               &gCDDatas[count].qcc,
               &gCDDatas[count].expError);

         if(gCDDatas[count].resNum <= 0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d <= 0\n", gCDDatas[count].resNum);
            fprintf(stderr, "ReadCDData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gCDDatas[count].resNum > gNumRes)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d > %d residues\n",
                  gCDDatas[count].resNum,
                  gNumRes);
            fprintf(stderr, "ReadCDData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gCDDatas[count].expError == 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Nobody is perfect.\n");
            fprintf(stderr, "expError cannot be zero.\n");
            fprintf(stderr, "(unless you want an infinite penalty)\n");
            fprintf(stderr, "ReadCDData\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         ++count;
      }
   }

   fclose(cdFile);
   return(1);
}

/*
File ReadCData.c
Reads the carbon chemical shift data file.
*/

#include "Torc.h"
#include <string.h>

int ReadCData(char fileName[])
{
   FILE  *cFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0,
         row,
         col;

   if((cFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "resNum cName nName oName PASx PASy PASz");
      fprintf(stdout, " alpha(deg) beta(deg) obsCS delta expError\n");
      fprintf(stdout, "x      s     s     s     x.x  x.x  x.x");
      fprintf(stdout, "  x.x        x.x       x.x    x.x    x.x\n");
      fprintf(stdout, "etc.\n");
      fprintf(stdout, "(include the header line)\n");
      fprintf(stdout, "\n");
      return(0);
   }

   gNumCData = 0;
   while(fgets(aLine, gdLineLength, cFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
         ++gNumCData;
   }
   rewind(cFile);

   gCDatas = (csData *) malloc(gNumCData * sizeof(csData));
   if(gCDatas == (csData *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "ReadCData\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   while(fgets(aLine, gdLineLength, cFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      {
            for(row = X; row <= Z; ++row)
               for(col = X; col <= Z; ++col)
                  gCDatas[count].PAS[row][col] = 0.0;

            sscanf(aLine, "%d %s %s %s %lf %lf %lf %lf %lf %lf %lf %lf",
                  &gCDatas[count].resNum,
                  &gCDatas[count].atom1Name,
                  &gCDatas[count].atom2Name,
                  &gCDatas[count].atom3Name,
                  &gCDatas[count].PAS[X][X],
                  &gCDatas[count].PAS[Y][Y],
                  &gCDatas[count].PAS[Z][Z],
                  &gCDatas[count].alpha,
                  &gCDatas[count].beta,
                  &gCDatas[count].obsCS,
                  &gCDatas[count].delta,
                  &gCDatas[count].expError);
            gCDatas[count].alpha *= gdRadPerDeg;
            gCDatas[count].beta *= gdRadPerDeg;
            gCDatas[count].delta *= gdRadPerDeg;

            if(gCDatas[count].resNum <= 0)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "resNum %d <= 0\n", gCDatas[count].resNum);
               fprintf(stderr, "ReadCData\n");
               fprintf(stderr, "\n");
               exit(1);
            }
            if(gCDatas[count].resNum > gNumRes)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "resNum %d > %d residues\n",
                     gCDatas[count].resNum,
                     gNumRes);
               fprintf(stderr, "ReadCData\n");
               fprintf(stderr, "\n");
               exit(1);
            }
            if(gCDatas[count].expError == 0.0)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "Nobody is perfect.\n");
               fprintf(stderr, "expError cannot be zero.\n");
               fprintf(stderr, "(unless you want an infinite penalty)\n");
               fprintf(stderr, "ReadCData\n");
               fprintf(stderr, "\n");
               exit(1);
            }

            ++count;
         }
      }

      fclose(cFile);
      return(1);
   }

   /*
```

```
File ReadContFile.c
Reads the control file.
*/

#include "Torc.h"
#include <string.h>

void ReadContFile(char fileName[])
{
   FILE  *controlFile;
   char  aLine[gdLineLength],
         header[gdLineLength],
         moveType[gdLineLength],
         moveString[gdLineLength];

   if((controlFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "File '%s' not found\n", fileName);
      fprintf(stderr, "gCyclesAtT       x\n");
      fprintf(stderr, "gCyclesForceT    x\n");
      fprintf(stderr, "gTCycles         x\n");
      fprintf(stderr, "gTemperature     x.x\n");
      fprintf(stderr, "gTempFactor      x.x\n");
      fprintf(stderr, "gEquilSteps      x\n");
#ifndef NORMTORC
      fprintf(stderr, "gDiffusion       x.x\n");
#endif
      fprintf(stderr, "gCompensateRatio s (0.0 to 1.0, r for random\n");
      fprintf(stderr, "gTunnelRatio     s (0.0 to 1.0, r for random\n");
      fprintf(stderr, "gTorsionRatio    s (0.0 to 1.0, r for random\n");
      fprintf(stderr, "gAtomRatio       s (0.0 to 1.0, r for random\n");
      fprintf(stderr, "gMovePerHistory  x\n");
      fprintf(stderr, "gMovePerOri      x\n");
      fprintf(stderr, "gMovePerTraj     x\n");
      fprintf(stderr, "gTrajRes         x\n");
      fprintf(stderr, "gSRandSeed       x\n");
      fprintf(stderr, "(include the headers)\n");
      fprintf(stderr, "(use '#' to comment a line and use default value)\n");
      fprintf(stderr, "\n");
      exit(1);
   }

/* Hard wired default values */
   gCyclesAtT = 2000;
   gCyclesForceT = 200;
   gTCycles = 0;
   gTemperature = 300.0;
   gTempFactor = 0.9;
   gEquilSteps = 0;
   gCompensateRatio = 0.0;
   gTunnelRatio = 0.0;
   gTorsionRatio = 0.0;
   gAtomRatio = 0.0;
   gHistoryFlag = FALSE;
   gOriFlag = FALSE;
   gTrajFlag = FALSE;
   gSeedFlag = FALSE;

   fprintf(stdout, "\n");
   fprintf(stdout, "control file:\n");
   while(fgets(aLine, gdLineLength, controlFile) != NULL)
```

```
{
   sscanf(aLine, "%s", &header);

   if(strcmp(header, "gCyclesAtT") == 0)
   {
      sscanf(aLine, "%s %d", &header, &gCyclesAtT);
      fprintf(stdout, "gCyclesAtT       %d\n", gCyclesAtT);
      if(gCyclesAtT <= 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "gCyclesAtT must be >= 1\n");
         fprintf(stderr, "ReadContFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
   }
   if(strcmp(header, "gCyclesForceT") == 0)
   {
      sscanf(aLine, "%s %d", &header, &gCyclesForceT);
      fprintf(stdout, "gCyclesForceT    %d\n", gCyclesForceT);
      if(gCyclesForceT <= 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "gCyclesForceT must be >= 1\n");
         fprintf(stderr, "ReadContFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
   }
   if(strcmp(header, "gTCycles") == 0)
   {
      sscanf(aLine, "%s %d", &header, &gTCycles);
      fprintf(stdout, "gTCycles         %d\n", gTCycles);
      if(gTCycles < 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "gTCycles must be >= 0\n");
         fprintf(stderr, "ReadContFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
   }
   if(strcmp(header, "gTemperature") == 0)
   {
      sscanf(aLine, "%s %lf", &header, &gTemperature);
      fprintf(stdout, "gTemperature     %.1f\n", gTemperature);
      if(gTemperature <= 0.0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "gTemperature must be > 0.0\n");
         fprintf(stderr, "ReadContFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
   }
   if(strcmp(header, "gTempFactor") == 0)
   {
      sscanf(aLine, "%s %lf", &header, &gTempFactor);
      fprintf(stdout, "gTempFactor      %.2f\n", gTempFactor);
      if(gTempFactor <= 0.0)
      {
         fprintf(stderr, "\n");
```

```
            fprintf(stderr, "gTempFactor must be > 0.0\n");
            fprintf(stderr, "ReadContFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
      }
      if(strcmp(header, "gEquilSteps") == 0)
      {
         sscanf(aLine, "%s %d", &header, &gEquilSteps);
         fprintf(stdout, "gEquilSteps       %d\n", gEquilSteps);
         if(gEquilSteps < 0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "gEquilSteps must be >= 0\n");
            fprintf(stderr, "ReadContFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
      }
#ifndef NORMTORC
      if(strcmp(header, "gDiffusion") == 0)
      {
         sscanf(aLine, "%s %lf", &header, &gDiffusion);
         fprintf(stdout, "gDiffusion        %.1e\n", gDiffusion);
         if(gDiffusion <= 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "gDiffusion must be > 0.0\n");
            fprintf(stderr, "ReadContFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
      }
#endif
      if(strcmp(header, "gCompensateRatio") == 0)
      {
         sscanf(aLine, "%s %s", &header, &moveString);
         if(strcmp(moveString, "r") == 0)
         {
            do
            {
               gCompensateRatio = (double) random() / gMaxRand; /* random */
            }
            while(gAtomRatio + gTorsionRatio +
                  gCompensateRatio + gTunnelRatio > 1.0);
         }
         else
            gCompensateRatio = atof(moveString);
         fprintf(stdout, "gCompensateRatio %.2f\n", gCompensateRatio);
         if(gCompensateRatio < 0.0 || gCompensateRatio > 1.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "gCompensateRatio must be >= 0.0 and <= 1.0\n");
            fprintf(stderr, "ReadContFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
      }
      if(strcmp(header, "gTunnelRatio") == 0)
      {
         sscanf(aLine, "%s %s", &header, &moveString);
         if(strcmp(moveString, "r") == 0)
```

```
        {
           do
           {
              gTunnelRatio = (double) random() / gMaxRand; /* random */
           }
           while(gAtomRatio + gTorsionRatio +
                 gCompensateRatio + gTunnelRatio > 1.0);
        }
        else
           gTunnelRatio = atof(moveString);
        fprintf(stdout, "gTunnelRatio     %.2f\n", gTunnelRatio);
        if(gTunnelRatio < 0.0 || gTunnelRatio > 1.0)
        {
           fprintf(stderr, "\n");
           fprintf(stderr, "gTunnelRatio must be >= 0.0 and <= 1.0\n");
           fprintf(stderr, "ReadContFile\n");
           fprintf(stderr, "\n");
           exit(1);
        }
     }
     if(strcmp(header, "gTorsionRatio") == 0)
     {
        sscanf(aLine, "%s %s", &header, &moveString);
        if(strcmp(moveString, "r") == 0)
        {
           do
           {
              gTorsionRatio = (double) random() / gMaxRand; /* random */
           }
           while(gAtomRatio + gTorsionRatio +
                 gCompensateRatio + gTunnelRatio > 1.0);
        }
        else
           gTorsionRatio = atof(moveString);
        fprintf(stdout, "gTorsionRatio    %.2f\n", gTorsionRatio);
        if(gTorsionRatio < 0.0 || gTorsionRatio > 1.0)
        {
           fprintf(stderr, "\n");
           fprintf(stderr, "gTorsionRatio must be >= 0.0 and <= 1.0\n");
           fprintf(stderr, "ReadContFile\n");
           fprintf(stderr, "\n");
           exit(1);
        }
     }
     if(strcmp(header, "gAtomRatio") == 0)
     {
        sscanf(aLine, "%s %s", &header, &moveString);
        if(strcmp(moveString, "r") == 0)
        {
           do
           {
              gAtomRatio = (double) random() / gMaxRand; /* random */
           }
           while(gAtomRatio + gTorsionRatio +
                 gCompensateRatio + gTunnelRatio > 1.0);
        }
        else
           gAtomRatio = atof(moveString);
        fprintf(stdout, "gAtomRatio       %.2f\n", gAtomRatio);
        if(gAtomRatio < 0.0 || gAtomRatio > 1.0)
        {
           fprintf(stderr, "\n");
```

```
      fprintf(stderr, "gAtomRatio must be >= 0.0 and <= 1.0\n");
      fprintf(stderr, "ReadContFile\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}
if(strcmp(header, "gMovePerHisFrame") == 0)
{
   gHistoryFlag = TRUE;
   sscanf(aLine, "%s %d", &header, &gMovePerHistory);
   fprintf(stdout, "gMovePerHistory  %d\n", gMovePerHistory);
   if(gMovePerHistory <= 0)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "gMovePerHistory must be > 0\n");
      fprintf(stderr, "ReadContFile\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}
if(strcmp(header, "gMovePerOri") == 0)
{
   gOriFlag = TRUE;
   sscanf(aLine, "%s %d", &header, &gMovePerOri);
   fprintf(stdout, "gMovePerOri      %d\n", gMovePerOri);
   if(gMovePerOri <= 0)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "gMovePerOri must be > 0\n");
      fprintf(stderr, "ReadContFile\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}
if(strcmp(header, "gMovePerTraj") == 0)
{
   gTrajFlag = TRUE;
   gTrajRes = 1;
   sscanf(aLine, "%s %d", &header, &gMovePerTraj);
   fprintf(stdout, "gMovePerTraj     %d\n", gMovePerTraj);
   if(gMovePerTraj <= 0)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "gMovePerTraj must be > 0\n");
      fprintf(stderr, "ReadContFile\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}
if(strcmp(header, "gTrajRes") == 0)
{
   sscanf(aLine, "%s %d", &header, &gTrajRes);
   fprintf(stdout, "gTrajRes         %d\n", gTrajRes);
   if(gTrajRes <= 0 || gTrajRes >= gNumRes)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "gTrajRes must be > 0 and < gNumRes\n");
      fprintf(stderr, "ReadContFile\n");
      fprintf(stderr, "\n");
      exit(1);
   }
}
```

```
      if(strcmp(header, "gSRandSeed") == 0)
      {
         gSeedFlag = TRUE;
         sscanf(aLine, "%s %ld", &header, &gSRandSeed);
         fprintf(stdout, "gSRandSeed       %d\n", gSRandSeed);
         if(gSRandSeed < 0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "gSRandSeed must be >= 0\n");
            fprintf(stderr, "ReadContFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
      }
   }

   if(gAtomRatio + gTorsionRatio + gCompensateRatio + gTunnelRatio != 1.0)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "Move ratios do not add to 1.0.\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   fprintf(stdout, "\n");
   fclose(controlFile);
}

/*
File ReadDisFile.c
Reads the distance data file.
*/

#include "Torc.h"

int ReadDisFile(char fileName[])
{
   FILE     *distanceFile;
   char     aLine[gdLineLength],
            header[gdLineLength];
   int      found = FALSE,
            disCount = 0;
   double   atom[3];

   if((distanceFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "11 H  16 O 1.96 0.3\n");
      fprintf(stdout, " 1 O   8 N 2.91 0.3\n");
      fprintf(stdout, "\n");
      return(0);
   }

   gNumDisData = 0;
   while(fgets(aLine, gdLineLength, distanceFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if(header[0] != '#')
         ++gNumDisData;
   }
```

```
rewind(distanceFile);

gDisDatas = (disData *) malloc(gNumDisData * sizeof(disData));
if(gDisDatas == (disData *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "ReadDisFile\n");
   fprintf(stderr, "\n");
   exit(1);
}

while(fgets(aLine, gdLineLength, distanceFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if(header[0] != '#')
   {
      sscanf(aLine, "%d %s %d %s %lf %lf",
            &gDisDatas[disCount].atomOneResNumber,
            &gDisDatas[disCount].atomOneType,
            &gDisDatas[disCount].atomTwoResNumber,
            &gDisDatas[disCount].atomTwoType,
            &gDisDatas[disCount].a1a2Distance,
            &gDisDatas[disCount].expError);

      if(gDisDatas[disCount].atomOneResNumber <= 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "residue number %d <= 0\n",
               gDisDatas[disCount].atomOneResNumber);
         fprintf(stderr, "%s", aLine);
         fprintf(stderr, "ReadDisFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
      if(gDisDatas[disCount].atomOneResNumber > gNumRes)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "residue number %d > %d\n",
               gDisDatas[disCount].atomOneResNumber,
               gNumRes);
         fprintf(stderr, "%s", aLine);
         fprintf(stderr, "ReadDisFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }

      if(gDisDatas[disCount].atomTwoResNumber <= 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "residue number %d <= 0\n",
               gDisDatas[disCount].atomTwoResNumber);
         fprintf(stderr, "%s", aLine);
         fprintf(stderr, "ReadDisFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
      if(gDisDatas[disCount].atomTwoResNumber > gNumRes)
      {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "residue number %d > %d residues\n",
                  gDisDatas[disCount].atomTwoResNumber,
                  gNumRes);
            fprintf(stderr, "%s", aLine);
            fprintf(stderr, "ReadDisFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         FindAtom(gDisDatas[disCount].atomOneResNumber - 1,
                  gDisDatas[disCount].atomOneType,
                  atom,
                  &found);
         if(!found)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Atom '%s' not found\n",
                  gDisDatas[disCount].atomOneType);
            fprintf(stderr, "residue %d\n",
                  gDisDatas[disCount].atomOneResNumber);
            fprintf(stderr, "ReadDisFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         FindAtom(gDisDatas[disCount].atomTwoResNumber - 1,
                  gDisDatas[disCount].atomTwoType,
                  atom,
                  &found);
         if(!found)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Atom '%s' not found\n",
                  gDisDatas[disCount].atomTwoType);
            fprintf(stderr, "residue %d\n",
                  gDisDatas[disCount].atomTwoResNumber);
            fprintf(stderr, "ReadDisFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         if(gDisDatas[disCount].expError == 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Nobody is perfect.\n");
            fprintf(stderr, "expError cannot be zero.\n");
            fprintf(stderr, "(unless you want an infinite penalty)\n");
            fprintf(stderr, "ReadDisFile\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         ++disCount;
      }
   }

   fclose(distanceFile);
   return(1);
```

```
}

/*
File ReadFiles.c
Engine to read all needed files.
*/

#include "Torc.h"

void ReadFiles(char dirName[])
{
   char  coordInName[gdLineLength],
         controlName[gdLineLength],
         lambdaName[gdLineLength],
         limitName[gdLineLength],
         nDataName[gdLineLength],
         cDataName[gdLineLength],
         iDataName[gdLineLength],
         ncDataName[gdLineLength],
         nhDataName[gdLineLength],
         ihDataName[gdLineLength],
         distanceName[gdLineLength],
         cdDataName[gdLineLength];

   sprintf(coordInName, "%s/%s", dirName, "coord.in.pdb");
   sprintf(controlName, "%s/%s", dirName, "control");
   sprintf(lambdaName, "%s/%s", dirName, "lambda");
   sprintf(limitName, "%s/%s", dirName, "limit");
   sprintf(nDataName, "%s/%s", dirName, "nData");
   sprintf(cDataName, "%s/%s", dirName, "cData");
   sprintf(iDataName, "%s/%s", dirName, "iData");
   sprintf(ncDataName, "%s/%s", dirName, "ncData");
   sprintf(nhDataName, "%s/%s", dirName, "nhData");
   sprintf(ihDataName, "%s/%s", dirName, "ihData");
   sprintf(distanceName, "%s/%s", dirName, "disData");
   sprintf(cdDataName, "%s/%s", dirName, "cdData");

   ReadPDBFile(coordInName);
   ReadContFile(controlName);
   ReadLamFile(lambdaName);
   ReadLimFile(limitName);

   ReadNData(nDataName);
   ReadCData(cDataName);
   ReadIData(iDataName);
   ReadNCData(ncDataName);
   ReadNHData(nhDataName);
   ReadIHData(ihDataName);
   ReadDisFile(distanceName);
   ReadCDData(cdDataName);
}

/*
File ReadIData.c
Reads the indole chemical shift data file.
*/

#include "Torc.h"
#include <string.h>

int ReadIData(char fileName[])
{
```

```
FILE  *iFile;
char  aLine[gdLineLength],
      header[gdLineLength];
int   count = 0,
      row,
      col;

if((iFile = fopen(fileName, "r")) == NULL)
{
   fprintf(stdout, "\n");
   fprintf(stdout, "File '%s' not found\n", fileName);
   fprintf(stdout, "The format of the file should be:\n");
   fprintf(stdout, "resNum nName cName hName PASx PASy PASz");
   fprintf(stdout, " alpha(deg) beta(deg) obsCS delta expError\n");
   fprintf(stdout, "x      s     s     s     x.x  x.x  x.x");
   fprintf(stdout, "  x.x        x.x       x.x   x.x   x.x\n");
   fprintf(stdout, "etc.\n");
   fprintf(stdout, "(include the header line)\n");
   fprintf(stdout, "\n");
   return(0);
}

gNumIData = 0;
while(fgets(aLine, gdLineLength, iFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      ++gNumIData;
}
rewind(iFile);

gIDatas = (csData *) malloc(gNumIData * sizeof(csData));
if(gIDatas == (csData *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "ReadIData\n");
   fprintf(stderr, "\n");
   exit(1);
}

while(fgets(aLine, gdLineLength, iFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
   {
      for(row = X; row <= Z; ++row)
         for(col = X; col <= Z; ++col)
            gIDatas[count].PAS[row][col] = 0.0;

      sscanf(aLine, "%d %s %s %s %lf %lf %lf %lf %lf %lf %lf %lf",
            &gIDatas[count].resNum,
            &gIDatas[count].atom1Name,
            &gIDatas[count].atom2Name,
            &gIDatas[count].atom3Name,
            &gIDatas[count].PAS[X][X],
            &gIDatas[count].PAS[Y][Y],
            &gIDatas[count].PAS[Z][Z],
            &gIDatas[count].alpha,
            &gIDatas[count].beta,
            &gIDatas[count].obsCS,
            &gIDatas[count].delta,
                &gIDatas[count].expError);
            gIDatas[count].alpha *= gdRadPerDeg;
            gIDatas[count].beta *= gdRadPerDeg;
            gIDatas[count].delta *= gdRadPerDeg;

            if(gIDatas[count].resNum <= 0)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "resNum %d <= 0\n", gIDatas[count].resNum);
               fprintf(stderr, "ReadIData\n");
               fprintf(stderr, "\n");
               exit(1);
            }
            if(gIDatas[count].resNum > gNumRes)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "resNum %d > %d residues\n",
                     gIDatas[count].resNum,
                     gNumRes);
               fprintf(stderr, "ReadIData\n");
               fprintf(stderr, "\n");
               exit(1);
            }
            if(gIDatas[count].expError == 0.0)
            {
               fprintf(stderr, "\n");
               fprintf(stderr, "***Operator Error***\n");
               fprintf(stderr, "Nobody is perfect.\n");
               fprintf(stderr, "expError cannot be zero.\n");
               fprintf(stderr, "(unless you want an infinite penalty)\n");
               fprintf(stderr, "ReadIData\n");
               fprintf(stderr, "\n");
               exit(1);
            }

            ++count;
         }
      }

      fclose(iFile);
      return(1);
   }

/*
File ReadIHData.c
Reads the ih dipolar splitting data file.
*/

#include "Torc.h"
#include <string.h>

int ReadIHData(char fileName[])
{
   FILE  *ihFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0;

   if((ihFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "resNum nName hName dip nuParallel expError\n");
      fprintf(stdout, "x      s     s     x.x x.x        x.x\n");
      fprintf(stdout, "etc.\n");
      fprintf(stdout, "(include the header lines)\n");
      fprintf(stdout, "\n");
      return(0);
   }

   gNumIHData = 0;
   while(fgets(aLine, gdLineLength, ihFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
         ++gNumIHData;
   }
   rewind(ihFile);

   gIHDatas = (dipData *) malloc(gNumIHData * sizeof(dipData));
   if(gIHDatas == (dipData *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "ReadIHData\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   while(fgets(aLine, gdLineLength, ihFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      {
         sscanf(aLine, "%d %s %s %lf %lf %lf",
               &gIHDatas[count].resNum,
               &gIHDatas[count].atom1Name,
               &gIHDatas[count].atom2Name,
               &gIHDatas[count].dip,
               &gIHDatas[count].nuParallel,
               &gIHDatas[count].expError);

         if(gIHDatas[count].resNum <= 0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d <= 0\n", gIHDatas[count].resNum);
            fprintf(stderr, "ReadIHData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gIHDatas[count].resNum > gNumRes)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d > %d residues\n",
                  gIHDatas[count].resNum, gNumRes);
            fprintf(stderr, "ReadIHData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gIHDatas[count].expError == 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Nobody is perfect.\n");
            fprintf(stderr, "expError cannot be zero.\n");
            fprintf(stderr, "(unless you want an infinite penalty)\n");
            fprintf(stderr, "ReadIHData\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         ++count;
      }
   }

   fclose(ihFile);
   return(1);
}

/*
File ReadLamFile.c
Reads the lambda file.
*/

#include "Torc.h"
#include <string.h>

void ReadLamFile(char fileName[])
{
   FILE  *lambdaFile;
   char  aLine[gdLineLength],
         header[gdLineLength];

   if((lambdaFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "File '%s' not found\n", fileName);
      fprintf(stderr, "The format of the file should be:\n");
      fprintf(stderr, "ncs lambda1(ncs)\n");
      fprintf(stderr, "ccs lambda2(ccs)\n");
      fprintf(stderr, "ics lambda3(ics)\n");
      fprintf(stderr, "nc  lambda4(nc)\n");
      fprintf(stderr, "nh  lambda5(nh)\n");
      fprintf(stderr, "ih  lambda6(ih)\n");
      fprintf(stderr, "dis lambda7(dis)\n");
      fprintf(stderr, "cd  lambda8(cd)\n");
#ifndef NORMTORC
      fprintf(stderr, "e   lambda9(e)\n");
#endif
      fprintf(stderr, "(include the headers)\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   gLambda[NCS] = 0.0;
   gLambda[CCS] = 0.0;
   gLambda[ICS] = 0.0;
   gLambda[NC] = 0.0;
   gLambda[NH] = 0.0;
   gLambda[IH] = 0.0;
   gLambda[DIS] = 0.0;
   gLambda[CD] = 0.0;
#ifndef NORMTORC
   gLambda[E] = 0.0;
#endif

   while(fgets(aLine, gdLineLength, lambdaFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if(strcmp(header, "ncs") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[NCS]);
      if(strcmp(header, "ccs") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[CCS]);
      if(strcmp(header, "ics") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[ICS]);
      if(strcmp(header, "nc") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[NC]);
      if(strcmp(header, "nh") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[NH]);
      if(strcmp(header, "ih") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[IH]);
      if(strcmp(header, "dis") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[DIS]);
      if(strcmp(header, "cd") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[CD]);
#ifndef NORMTORC
      if(strcmp(header, "e") == 0)
         sscanf(aLine, "%s %lf", &header, &gLambda[E]);
#endif
   }

   fclose(lambdaFile);
}

/*
File ReadLimFile.c
Reads the limit file.
*/

#include "Torc.h"
#include <string.h>

int ReadLimFile(char fileName[])
{
   FILE  *limitFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   limitCount = 0;

   if((limitFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "File '%s' not found\n", fileName);
      fprintf(stderr, "The format of the file should be:\n");
      fprintf(stderr, "resNum label    low high\n");
      fprintf(stderr, "x      phi      x.x x.x\n");
      fprintf(stderr, "x      psi      x.x x.x\n");
      fprintf(stderr, "x      omega    x.x x.x\n");
      fprintf(stderr, "(include the headers)\n");
      fprintf(stderr, "etc.\n");
      fprintf(stderr, "\n");
      return(0);
   }
```

```
gNumLimits = 0;

while(fgets(aLine, gdLineLength, limitFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      ++gNumLimits;
}
rewind(limitFile);

gLimits = (limit *) malloc(gNumLimits * sizeof(limit));
if(gLimits == (limit *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "file %s\n", fileName);
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}

while(fgets(aLine, gdLineLength, limitFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
   {
      sscanf(aLine, "%d %s %lf %lf",
             &gLimits[limitCount].resNum,
             &gLimits[limitCount].bondName,
             &gLimits[limitCount].low,
             &gLimits[limitCount].high);

      if(gLimits[limitCount].low >= gLimits[limitCount].high)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "low limit >= high limit\n");
         fprintf(stderr, "%d %s %f %f\n",
               gLimits[limitCount].resNum,
               gLimits[limitCount].bondName,
               gLimits[limitCount].low,
               gLimits[limitCount].high);
         fprintf(stderr, "ReadLimFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }

      if(fabs(gLimits[limitCount].low) > 180.0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "fabs(low) greater than 180.0\n");
         fprintf(stderr, "%d %s %f %f\n",
               gLimits[limitCount].resNum,
               gLimits[limitCount].bondName,
               gLimits[limitCount].low,
               gLimits[limitCount].high);
         fprintf(stderr, "ReadLimFile\n");
         fprintf(stderr, "\n");
         exit(1);
      }
```

```
if(fabs(gLimits[limitCount].high) > 180.0)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "***Operator Error***\n");
   fprintf(stderr, "fabs(high) greater than 180.0\n");
   fprintf(stderr, "%d %s %f %f\n",
         gLimits[limitCount].resNum,
         gLimits[limitCount].bondName,
         gLimits[limitCount].low,
         gLimits[limitCount].high);
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}

if((strcmp(gLimits[limitCount].bondName, "phi") != 0) &&
   (strcmp(gLimits[limitCount].bondName, "psi") != 0) &&
   (strcmp(gLimits[limitCount].bondName, "omega") != 0))
{
   fprintf(stderr, "\n");
   fprintf(stderr, "***Operator Error***\n");
   fprintf(stderr, "Torsion type '%s' wrong\n",
         gLimits[limitCount].bondName);
   fprintf(stderr, "Use either phi, psi, or omega as labels.\n");
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}
if((gLimits[limitCount].resNum == gNumRes) &&
   (strcmp(gLimits[limitCount].bondName, "omega") == 0))
{
   fprintf(stderr, "\n");
   fprintf(stderr, "***Operator Error***\n");
   fprintf(stderr, "residue %d", gLimits[limitCount].resNum);
   fprintf(stderr, " does not have an omega torsion angle.\n");
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}
if(gLimits[limitCount].resNum < 1)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "***Operator Error***\n");
   fprintf(stderr, "residue %d", gLimits[limitCount].resNum);
   fprintf(stderr, " does not exist.\n");
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}
if(gLimits[limitCount].resNum > gNumRes)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "***Operator Error***\n");
   fprintf(stderr, "residue %d", gLimits[limitCount].resNum);
   fprintf(stderr, " larger than peptide (%d residues)\n", gNumRes);
   fprintf(stderr, "ReadLimFile\n");
   fprintf(stderr, "\n");
   exit(1);
}

++limitCount;
```

```
      }
   }

   fclose(limitFile);
   return(1);
}

/*
File ReadNCData.c
Reads the nc dipolar splitting data file.
*/

#include "Torc.h"
#include <string.h>

int ReadNCData(char fileName[])
{
   FILE  *ncFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0;

   if((ncFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "resNum nName cName dip nuParallel expError\n");
      fprintf(stdout, "x      s     s     x.x x.x        x.x\n");
      fprintf(stdout, "etc.\n");
      fprintf(stdout, "(include the headers)\n");
      fprintf(stdout, "\n");
      return(0);
   }

   gNumNCData = 0;
   while(fgets(aLine, gdLineLength, ncFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
         ++gNumNCData;
   }
   rewind(ncFile);

   gNCDatas = (dipData *) malloc(gNumNCData * sizeof(dipData));
   if(gNCDatas == (dipData *) NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "out of memory!\n");
      fprintf(stderr, "ReadNCData\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   while(fgets(aLine, gdLineLength, ncFile) != NULL)
   {
      sscanf(aLine, "%s", &header);
      if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      {
         sscanf(aLine, "%d %s %s %lf %lf %lf",
               &gNCDatas[count].resNum,
               &gNCDatas[count].atom1Name,
                &gNCDatas[count].atom2Name,
                &gNCDatas[count].dip,
                &gNCDatas[count].nuParallel,
                &gNCDatas[count].expError);

         if(gNCDatas[count].resNum <= 0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d <= 0\n", gNCDatas[count].resNum);
            fprintf(stderr, "ReadNCData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gNCDatas[count].resNum > gNumRes)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d > %d residues\n",
                  gNCDatas[count].resNum,
                  gNumRes);
            fprintf(stderr, "ReadNCData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gNCDatas[count].expError == 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Nobody is perfect.\n");
            fprintf(stderr, "expError cannot be zero.\n");
            fprintf(stderr, "(unless you want an infinite penalty)\n");
            fprintf(stderr, "ReadNCData\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         ++count;
      }
   }

   fclose(ncFile);
   return(1);
}

/*
File ReadNData.c
Reads the nitrogen chemical shift data file.
*/

#include "Torc.h"
#include <string.h>

int ReadNData(char fileName[])
{
   FILE  *nFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0,
         row,
         col;
```

```
if((nFile = fopen(fileName, "r")) == NULL)
{
   fprintf(stdout, "\n");
   fprintf(stdout, "File '%s' not found\n", fileName);
   fprintf(stdout, "The format of the file should be:\n");
   fprintf(stdout, "resNum nName cName hName PASx PASy PASz");
   fprintf(stdout, " alpha(deg) beta(deg) obsCS delta expError\n");
   fprintf(stdout, "x      s     s     s     x.x  x.x  x.x");
   fprintf(stdout, "  x.x        x.x       x.x   x.x   x.x\n");
   fprintf(stdout, "etc.\n");
   fprintf(stdout, "(include the header line)\n");
   fprintf(stdout, "\n");
   return(0);
}

gNumNData = 0;
while(fgets(aLine, gdLineLength, nFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      ++gNumNData;
}
rewind(nFile);

gNDatas = (csData *) malloc(gNumNData * sizeof(csData));
if(gNDatas == (csData *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "ReadNData\n");
   fprintf(stderr, "\n");
   exit(1);
}

while(fgets(aLine, gdLineLength, nFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
   {
      for(row = X; row <= Z; ++row)
         for(col = X; col <= Z; ++col)
            gNDatas[count].PAS[row][col] = 0.0;

      sscanf(aLine, "%d %s %s %s %lf %lf %lf %lf %lf %lf %lf %lf",
            &gNDatas[count].resNum,
            &gNDatas[count].atom1Name,
            &gNDatas[count].atom2Name,
            &gNDatas[count].atom3Name,
            &gNDatas[count].PAS[X][X],
            &gNDatas[count].PAS[Y][Y],
            &gNDatas[count].PAS[Z][Z],
            &gNDatas[count].alpha,
            &gNDatas[count].beta,
            &gNDatas[count].obsCS,
            &gNDatas[count].delta,
            &gNDatas[count].expError);
      gNDatas[count].alpha *= gdRadPerDeg;
      gNDatas[count].beta *= gdRadPerDeg;
      gNDatas[count].delta *= gdRadPerDeg;

      if(gNDatas[count].resNum <= 0)
      {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d <= 0\n", gNDatas[count].resNum);
            fprintf(stderr, "ReadNData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gNDatas[count].resNum > gNumRes)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "resNum %d > %d residues\n",
                  gNDatas[count].resNum, gNumRes);
            fprintf(stderr, "ReadNData\n");
            fprintf(stderr, "\n");
            exit(1);
         }
         if(gNDatas[count].expError == 0.0)
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "***Operator Error***\n");
            fprintf(stderr, "Nobody is perfect.\n");
            fprintf(stderr, "expError cannot be zero.\n");
            fprintf(stderr, "(unless you want an infinite penalty)\n");
            fprintf(stderr, "ReadNData\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         ++count;
      }
   }

   fclose(nFile);
   return(1);
}

/*
File ReadNHData.c
Reads the nh dipolar splitting data file.
*/

#include "Torc.h"
#include <string.h>

int ReadNHData(char fileName[])
{
   FILE  *nhFile;
   char  aLine[gdLineLength],
         header[gdLineLength];
   int   count = 0;

   if((nhFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stdout, "\n");
      fprintf(stdout, "File '%s' not found\n", fileName);
      fprintf(stdout, "The format of the file should be:\n");
      fprintf(stdout, "resNum nName hName dip nuParallel expError\n");
      fprintf(stdout, "x      s     s     x.x x.x        x.x\n");
      fprintf(stdout, "etc.\n");
      fprintf(stdout, "(include the header lines)\n");
      fprintf(stdout, "\n");
   return(0);
}

gNumNHData = 0;
while(fgets(aLine, gdLineLength, nhFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
      ++gNumNHData;
}
rewind(nhFile);

gNHDatas = (dipData *) malloc(gNumNHData * sizeof(dipData));
if(gNHDatas == (dipData *) NULL)
{
   fprintf(stderr, "\n");
   fprintf(stderr, "out of memory!\n");
   fprintf(stderr, "ReadNHData\n");
   fprintf(stderr, "\n");
   exit(1);
}

while(fgets(aLine, gdLineLength, nhFile) != NULL)
{
   sscanf(aLine, "%s", &header);
   if((header[0] != '#') && (strcmp(header, "resNum") != 0))
   {
      sscanf(aLine, "%d %s %s %lf %lf %lf",
            &gNHDatas[count].resNum,
            &gNHDatas[count].atom1Name,
            &gNHDatas[count].atom2Name,
            &gNHDatas[count].dip,
            &gNHDatas[count].nuParallel,
            &gNHDatas[count].expError);

      if(gNHDatas[count].resNum <= 0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "resNum %d <= 0\n", gNHDatas[count].resNum);
         fprintf(stderr, "ReadNHData\n");
         fprintf(stderr, "\n");
         exit(1);
      }
      if(gNHDatas[count].resNum > gNumRes)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "resNum %d > %d residues\n",
               gNHDatas[count].resNum, gNumRes);
         fprintf(stderr, "ReadNHData\n");
         fprintf(stderr, "\n");
         exit(1);
      }
      if(gNHDatas[count].expError == 0.0)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "***Operator Error***\n");
         fprintf(stderr, "Nobody is perfect.\n");
         fprintf(stderr, "expError cannot be zero.\n");
         fprintf(stderr, "(unless you want an infinite penalty)\n");
         fprintf(stderr, "ReadNHData\n");
```

```
            fprintf(stderr, "\n");
            exit(1);
         }

         ++count;
      }
   }

   fclose(nhFile);
   return(1);
}

/*
File ReadPDBFile.c
Reads the coodinates file.
*/

#include "Torc.h"
#include <string.h>

void ReadPDBFile(char fileName[])
{
   FILE  *coordFile;
   char  aLine[gdLineLength],
         lastRes[gdLineLength],
         tmp[2],
         header[7],
         residueNum[6],
         currentRes[gdLineLength];
   int   atomCount = 0,
         residueCount = 0,
         charCount;

   if((coordFile = fopen(fileName, "r")) == NULL)
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "File '%s' not found\n", fileName);
      fprintf(stderr, "The file should be in PDB format.\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   strcpy(lastRes, "empty");

   gNumAtoms = 0;
   gNumRes = 0;
   while(fgets(aLine, gdLineLength, coordFile) != NULL)
   {
      strcpy(header, "");
      strcpy(tmp, "");
      strcpy(residueNum, "");
      sscanf(aLine, "%s", &header);
      if((strcmp("ATOM", header) == 0) || (strcmp("HETATM", header) == 0))
      {
         for(charCount = strlen(aLine); charCount < 70; ++charCount)
            strcat(aLine, " ");
         aLine[30] = '\0';
         sscanf(&aLine[26], "%s", &tmp);
         aLine[26] = '\0';
         sscanf(&aLine[22], "%s", &residueNum);
         strcat(residueNum, tmp);
```

```
        ++gNumAtoms;
        if(strcmp(lastRes, residueNum) != 0)
        {
           strcpy(lastRes, residueNum);
           ++gNumRes;
        }
     }
  }
  if(gNumAtoms == 0)
  {
     fprintf(stderr, "\n");
     fprintf(stderr, "***Operator Error***\n");
     fprintf(stderr, "No atoms read from file '%s'.\n", fileName);
     fprintf(stderr, "\n");
     exit(1);
  }
  rewind(coordFile);

  gAtoms = (atom *) malloc(gNumAtoms * sizeof(atom));
  if(gAtoms == (atom *) NULL)
  {
     fprintf(stderr, "\n");
     fprintf(stderr, "out of memory!\n");
     fprintf(stderr, "ReadPDBFile\n");
     fprintf(stderr, "\n");
     exit(1);
  }

  gResidues = (residue *) malloc(gNumRes * sizeof(residue));
  if(gResidues == (residue *) NULL)
  {
     fprintf(stderr, "\n");
     fprintf(stderr, "out of memory!\n");
     fprintf(stderr, "ReadPDBFile\n");
     fprintf(stderr, "\n");
     exit(1);
  }

  strcpy(lastRes, "empty");
  while(fgets(aLine, gdLineLength, coordFile) != NULL)
  {
     sscanf(aLine, "%s", &header);
     if((strcmp("ATOM", header) == 0) || (strcmp("HETATM", header) == 0))
     {
        for(charCount = strlen(aLine); charCount < 70; ++charCount)
           strcat(aLine, " ");

        gAtoms[atomCount].footnoteNum = -1;
        sscanf(&aLine[67], "%d", &gAtoms[atomCount].footnoteNum);
        aLine[67] = '\0';

        gAtoms[atomCount].tempFactor = 0.0;
        sscanf(&aLine[60], "%lf", &gAtoms[atomCount].tempFactor);
        aLine[60] = '\0';

        gAtoms[atomCount].occupancy = 1.0;
        sscanf(&aLine[54], "%lf", &gAtoms[atomCount].occupancy);
        aLine[54] = '\0';

        gAtoms[atomCount].coords[Z] = 0.0;
        sscanf(&aLine[46], "%lf", &gAtoms[atomCount].coords[Z]);
        aLine[46] = '\0';
```

```
         gAtoms[atomCount].coords[Y] = 0.0;
         sscanf(&aLine[38], "%lf", &gAtoms[atomCount].coords[Y]);
         aLine[38] = '\0';

         gAtoms[atomCount].coords[X] = 0.0;
         sscanf(&aLine[30], "%lf", &gAtoms[atomCount].coords[X]);
         aLine[30] = '\0';

         strcpy(gAtoms[atomCount].insertRes, " ");
         sscanf(&aLine[26], "%s", &gAtoms[atomCount].insertRes);
         aLine[26] = '\0';

         gAtoms[atomCount].resSeqNum = residueCount + 1;
         sscanf(&aLine[22], "%d", &gAtoms[atomCount].resSeqNum);
         aLine[22] = '\0';

         strcpy(gAtoms[atomCount].chainIdent, " ");
         sscanf(&aLine[21], "%s", &gAtoms[atomCount].chainIdent);
         aLine[21] = '\0';

         strcpy(gAtoms[atomCount].resName, "???");
         sscanf(&aLine[17], "%s", &gAtoms[atomCount].resName);
         aLine[17] = '\0';

         strcpy(gAtoms[atomCount].altLocInd, " ");
         sscanf(&aLine[16], "%s", &gAtoms[atomCount].altLocInd);
         aLine[16] = '\0';

         strcpy(gAtoms[atomCount].atomName, "???");
         sscanf(&aLine[12], "%s", &gAtoms[atomCount].atomName);
         aLine[12] = '\0';

         gAtoms[atomCount].atomSeqNum = atomCount + 1;
         sscanf(&aLine[6], "%d", &gAtoms[atomCount].atomSeqNum);
         aLine[6] = '\0';

         strcpy(gAtoms[atomCount].header, "???");
         sscanf(&aLine[0], "%s", &gAtoms[atomCount].header);
         aLine[0] = '\0';

         sprintf(currentRes, "%d%s",
               gAtoms[atomCount].resSeqNum,
               gAtoms[atomCount].insertRes);
         ++atomCount;
         if(strcmp(lastRes, currentRes) != 0)
         {
            gResidues[residueCount].begin = atomCount - 1;
            if(strcmp(lastRes, "empty") != 0)
               gResidues[residueCount - 1].end = atomCount - 2;
            strcpy(lastRes, currentRes);
            ++residueCount;
         }
         if(atomCount == gNumAtoms)
            gResidues[residueCount - 1].end = gNumAtoms - 1;
      }
   }
   fclose(coordFile);
}

/*
File RotateAtoms.c
```

```
Rotates necessary atoms after a defined bond.
*/

#include "Torc.h"
#include <string.h>

int RotateAtoms(atom *atomsIn,
                int resNum,
                char torType[],
                double mag)
{
   int       found = FALSE,
             backbone = FALSE,
             atomCount;
   double    q[4],
             rotMat[3][3],
             eulerMag,
             sinEulerMag,
             cosEulerMag,
             aboutBond[3],
             atom1[3],
             atom2[3];

   if(strcmp(torType, "phi") == 0)
   {
      FindAtom(resNum - 1, "N", atom1, &found);
      if(found)
         FindAtom(resNum - 1, "CA", atom2, &found);
      if(!found)
         return(0);

      backbone = TRUE;

      Vectorize(atom2, atom1, aboutBond);
      Normalize(aboutBond);
      eulerMag = mag * 0.5;
      sinEulerMag = sin(eulerMag);
      cosEulerMag = cos(eulerMag);
      q[0] = cosEulerMag;
      q[1] = aboutBond[X] * sinEulerMag;
      q[2] = aboutBond[Y] * sinEulerMag;
      q[3] = aboutBond[Z] * sinEulerMag;
      EulerMatrix(q, rotMat);

      for(atomCount = gResidues[resNum - 1].begin;
          atomCount <= gResidues[resNum - 1].end;
          ++atomCount)
      {
         if((strcmp(atomsIn[atomCount].atomName, "N") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "H") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "HN") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "HN1") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "CA") != 0))
         {
            DoRotation(rotMat, atomsIn[atomCount].coords, atom2);
         }
      }
   }
   else if(strcmp(torType, "psi") == 0)
   {
      FindAtom(resNum - 1, "CA", atom1, &found);
      if(found)
```

```
         FindAtom(resNum - 1, "C", atom2, &found);
      if(!found)
         return(0);

      backbone = TRUE;

      Vectorize(atom2, atom1, aboutBond);
      Normalize(aboutBond);
      eulerMag = mag * 0.5;
      sinEulerMag = sin(eulerMag);
      cosEulerMag = cos(eulerMag);
      q[0] = cosEulerMag;
      q[1] = aboutBond[X] * sinEulerMag;
      q[2] = aboutBond[Y] * sinEulerMag;
      q[3] = aboutBond[Z] * sinEulerMag;
      EulerMatrix(q, rotMat);

      for(atomCount = gResidues[resNum - 1].begin;
          atomCount <= gResidues[resNum - 1].end;
          ++atomCount)
      {
         if(strcmp(atomsIn[atomCount].atomName, "O") == 0)
            DoRotation(rotMat, atomsIn[atomCount].coords, atom2);
      }
   }
   else if(strcmp(torType, "omega") == 0)
   {
      FindAtom(resNum - 1, "C", atom1, &found);
      if(found)
         FindAtom(resNum, "N", atom2, &found);
      if(!found)
         return(0);

      backbone = TRUE;

      Vectorize(atom2, atom1, aboutBond);
      Normalize(aboutBond);
      eulerMag = mag * 0.5;
      sinEulerMag = sin(eulerMag);
      cosEulerMag = cos(eulerMag);
      q[0] = cosEulerMag;
      q[1] = aboutBond[X] * sinEulerMag;
      q[2] = aboutBond[Y] * sinEulerMag;
      q[3] = aboutBond[Z] * sinEulerMag;
      EulerMatrix(q, rotMat);
   }
   else if(strcmp(torType, "chi1") == 0)
   {
      FindAtom(resNum - 1, "CA", atom1, &found);
      if(found)
         FindAtom(resNum, "CB", atom2, &found);
      if(!found)
         return(0);

      Vectorize(atom2, atom1, aboutBond);
      Normalize(aboutBond);
      eulerMag = mag * 0.5;
      sinEulerMag = sin(eulerMag);
      cosEulerMag = cos(eulerMag);
      q[0] = cosEulerMag;
      q[1] = aboutBond[X] * sinEulerMag;
      q[2] = aboutBond[Y] * sinEulerMag;
      q[3] = aboutBond[Z] * sinEulerMag;
      EulerMatrix(q, rotMat);

      for(atomCount = gResidues[resNum - 1].begin;
          atomCount <= gResidues[resNum - 1].end;
          ++atomCount)
      {
         if((strcmp(atomsIn[atomCount].atomName, "N") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "H") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "HN") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "CA") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "HA") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "C") != 0) &&
            (strcmp(atomsIn[atomCount].atomName, "O") != 0))
         {
            DoRotation(rotMat, atomsIn[atomCount].coords, atom2);
         }
      }
   }
   else
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "***Operator Error***\n");
      fprintf(stderr, "Torsion type '%s' wrong\n", torType);
      fprintf(stderr, "Use phi, psi, omega, chi1, chi2, chi3 as labels.\n");
      fprintf(stderr, "RotateAtoms\n");
      fprintf(stderr, "\n");
      exit(1);
   }

   if(backbone)
   {
      for(atomCount = gResidues[resNum].begin;
          atomCount < gNumAtoms;
          ++atomCount)
         DoRotation(rotMat, atomsIn[atomCount].coords, atom2);

/*
Rotate everything back by 0.5 so that entire system is affected by the
torsion move.
*/
      eulerMag = -mag * 0.25;
      sinEulerMag = sin(eulerMag);
      cosEulerMag = cos(eulerMag);
      q[0] = cosEulerMag;
      q[1] = aboutBond[X] * sinEulerMag;
      q[2] = aboutBond[Y] * sinEulerMag;
      q[3] = aboutBond[Z] * sinEulerMag;
      EulerMatrix(q, rotMat);

      for(atomCount = 0; atomCount < gNumAtoms; ++atomCount)
         DoRotation(rotMat, atomsIn[atomCount].coords, atom2);
   }

   return(1);
}

/*
File Torc.c
Drives the refinement.
*/
```

```
#define MAIN
#define ldNoPrint    0
#define ldYesPrint   1

#include "Torc.h"
#include <string.h>
/* for time command */
#include <time.h>
/* for times command */
#include <sys/types.h>
#include <sys/times.h>
/* for CLK_TCK */
#include <limits.h>

#ifdef NORMTORC
int main(int argc,
         char *argv[])
#else
int torc_(char inDirName[],
          int *dirLength)
#endif
{
   char      dirName[gdLineLength],
             coordOutName[gdLineLength],
             historyName[gdLineLength],
             trajFileName[gdLineLength],
             oriFileName[gdLineLength];
   struct tms  cpuTimeStart,
                cpuTimeEnd;
   struct tm   *tmPointer;
   time_t    localTime;
   int       limitLineNum,
             resNum,
             foundMove,
             keepMove,
             numIterAtT,
             numTIterations = 0,
             continueT,
             continueAtT,
             numSuccess,
             count,
             timeStart,
             timeEnd,
             sRandSeed,
             historyCount = 0,
             compensateFlag,
             tunnelFlag,
             atomFlag,
             torsionFlag;
   unsigned long int totalCalc = 0,
                      successfulMoves = 0,
                      numAttCompensate = 0,
                      numAccCompensate = 0,
                      numAttTunnel = 0,
                      numAccTunnel = 0,
                      numAttAtom = 0,
                      numAccAtom = 0,
                      numAttTorsion = 0,
                      numAccTorsion = 0;
   double    penalty,
             newPenalty,
             mag,
            ranNum,
/*
            highLimit,
            highMag,
            lowMag,
*/
            cpuUserTime,
            cpuSystemTime,
            cpuTimeTotal,
            lowestPenalty;

#ifdef NORMTORC
   if(argc == 1)
   {
      fprintf(stdout, "\nProgram %s\n", argv[0]);
      fprintf(stdout, "(use '%s [directory name]')\n", argv[0]);
      fprintf(stdout, "(use '%s help' for help)\n", argv[0]);
      fprintf(stdout, "\nEnter directory name: ");
      scanf("%s", &dirName);
   }
   else if(argc == 2)
      sprintf(dirName, "%s", argv[1]);
   else
   {
      fprintf(stderr, "\n");
      fprintf(stderr, "***Only one directory name, please.***\n");
      fprintf(stderr, "usage: %s [directory name]\n", argv[0]);
      fprintf(stderr, "The directory name is optional.\n");
      fprintf(stderr, "\n");
      exit(1);
   }
#else
   for(count = 0; count < *dirLength; ++count)
      dirName[count] = inDirName[count];
   dirName[count] = '\0';
#endif

   if((strcmp(dirName, "help") == 0) || (strcmp(dirName, "HELP") == 0))
   {
      PrintHelp();
      exit(1);
   }

   sprintf(coordOutName, "%s/%s", dirName, "coord.out.pdb");

   fprintf(stdout, "\n");
   fprintf(stdout, "%s\n", gdVersion);

   localTime = time('\0');
   tmPointer = localtime(&localTime);
   fprintf(stdout, "%s\n", asctime(tmPointer));

   fprintf(stdout, "Refinement of %s\n", dirName);

   gMaxRand = pow(2, 31) - 1;

   ReadFiles(dirName);
   fprintf(stdout, "%d residues\n", gNumRes);
   fprintf(stdout, "%d atoms\n", gNumAtoms);
   fprintf(stdout, "\n");
   fflush(stdout);
   if(gSeedFlag == TRUE)
      sRandSeed = gSRandSeed;
   else
      sRandSeed = time(0);
   srandom(sRandSeed);
   fprintf(stdout, "random seed = %10d\n\n", sRandSeed);

   DeclArray();
   CopyCoord(gAtoms, gAtomsInitial);
   CopyCoord(gAtoms, gAtomsLowest);
   CopyCoord(gAtoms, gAtomsLastState);

#ifndef NORMTORC
   MakeXYZArray(gAtoms, gXArray, gYArray, gZArray);
#endif

   for(count = 0; count < gNumNData; ++count)
   {
      PAS_MF(gNDatas[count].PAS,
             gNDatas[count].alpha,
             gNDatas[count].beta,
             gNDatas[count].MF);
   }
   for(count = 0; count < gNumCData; ++count)
   {
      PAS_MF(gCDatas[count].PAS,
             gCDatas[count].alpha,
             gCDatas[count].beta,
             gCDatas[count].MF);
   }
   for(count = 0; count < gNumIData; ++count)
   {
      PAS_MF(gIDatas[count].PAS,
             gIDatas[count].alpha,
             gIDatas[count].beta,
             gIDatas[count].MF);
   }

   gInitTemp = gTemperature;
   gKBInitT = gInitTemp * gdBoltzmann;

   if(gTrajFlag)
   {
      sprintf(trajFileName, "%s/%s", dirName, "trajectory");
      if((gTrajFile = fopen(trajFileName, "w")) == NULL)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "Could not open trajectory file: ");
         fprintf(stderr, "%s\n", trajFileName);
         fprintf(stderr, "\n");
         exit(1);
      }
      fprintf(gTrajFile, "%-4s%3d%1s\n",
            gAtoms[gResidues[gTrajRes - 1].begin].resName,
            gAtoms[gResidues[gTrajRes - 1].begin].resSeqNum,
            gAtoms[gResidues[gTrajRes - 1].begin].insertRes);
      fprintf(gTrajFile, "step\tt");
      fprintf(gTrajFile, "\tncs\tccs\tics\tnc\tnh\tih\tcd");
#ifndef NORMTORC
      fprintf(gTrajFile, "\te");
#endif
      fprintf(gTrajFile, "\tncsP\tccsP\ticsP\tncP\tnhP\tihP\tdisP\tcdP");
#ifndef NORMTORC
      fprintf(gTrajFile, "\teP");
#endif
      fprintf(gTrajFile, "\ttotP");
      fprintf(gTrajFile, "\n");
      fprintf(gTrajFile, "%i", totalCalc);
      fprintf(gTrajFile, "\t%.4e", gTemperature);
      CalcTotP(ldNoPrint);
      fprintf(gTrajFile, "\t%.4f", gCalcNCS);
      fprintf(gTrajFile, "\t%.4f", gCalcCCS);
      fprintf(gTrajFile, "\t%.4f", gCalcICS);
      fprintf(gTrajFile, "\t%.4f", gCalcNC);
      fprintf(gTrajFile, "\t%.4f", gCalcNH);
      fprintf(gTrajFile, "\t%.4f", gCalcIH);
      fprintf(gTrajFile, "\t%.4f", gCalcCD);
#ifndef NORMTORC
      fprintf(gTrajFile, "\t%.4f", gCalcE);
#endif
      fprintf(gTrajFile, "\t%.4f", gCalcNCSP);
      fprintf(gTrajFile, "\t%.4f", gCalcCCSP);
      fprintf(gTrajFile, "\t%.4f", gCalcICSP);
      fprintf(gTrajFile, "\t%.4f", gCalcNCP);
      fprintf(gTrajFile, "\t%.4f", gCalcNHP);
      fprintf(gTrajFile, "\t%.4f", gCalcIHP);
      fprintf(gTrajFile, "\t%.4f", gCalcDISP);
      fprintf(gTrajFile, "\t%.4f", gCalcCDP);
#ifndef NORMTORC
      fprintf(gTrajFile, "\t%.4f", gCalcEP);
#endif
      fprintf(gTrajFile, "\t%.4f", gCalcTotP);
      fprintf(gTrajFile, "\n");
      fflush(gTrajFile);
   }

   penalty = CalcTotP(ldYesPrint);
   lowestPenalty = penalty;
   CalcTorsion();
   CalcResPT();

   if(gOriFlag)
   {
      sprintf(oriFileName, "%s/%s", dirName, "orientation");
      if((gOriFile = fopen(oriFileName, "w")) == NULL)
      {
         fprintf(stderr, "\n");
         fprintf(stderr, "Could not open orientation file: ");
         fprintf(stderr, "%s\n", oriFileName);
         fprintf(stderr, "\n");
         exit(1);
      }
      fprintf(gOriFile, "step\tbondType");
      CalcOri("N-H", totalCalc, ldNoPrint);
      fflush(gOriFile);
   }
   fprintf(stdout, "Residue Bond   Orient  Angle Direct\n");
   fprintf(stdout, "------- ---- -------- ------ ------\n");
   CalcOri("N-H", totalCalc, ldYesPrint);
   fprintf(stdout, "\n");
   CalcOri("C-O", totalCalc, ldYesPrint);
   fprintf(stdout, "\n");

   fflush(stdout);
```

```
#ifndef NORMTORC
   for(count = 0; count < gNumAtoms; ++count)
   {
      gXForceOldState[count] = gXForce[count];
      gYForceOldState[count] = gYForce[count];
      gZForceOldState[count] = gZForce[count];
   }
#endif

/*
   highLimit = 0.0;
   for(count = 0; count < gNumLimits; ++count)
   {
      if(highLimit < fabs(gLimits[count].low))
         highLimit = fabs(gLimits[count].low);
      if(highLimit < fabs(gLimits[count].high))
         highLimit = fabs(gLimits[count].high);
   }
*/

   if(gHistoryFlag)
   {
      ++historyCount;
      sprintf(historyName, "%s/%s%04d.pdb",
            dirName, "history/coord", historyCount);
      OutputPDB(historyName, gAtoms);
   }

   timeStart = time(0);
   times(&cpuTimeStart);

   continueT = TRUE;
   do
   {
      continueAtT = TRUE;
      numSuccess = 0;
      numIterAtT = 0;
      ++numTIterations;
/*
      lowMag = highLimit;
      highMag = -highLimit;
*/
      do
      {
         ++numIterAtT;
         compensateFlag = FALSE;
         tunnelFlag = FALSE;
         atomFlag = FALSE;
         torsionFlag = FALSE;

         ranNum = (double) random() / gMaxRand;

         if((gCompensateRatio > 0.0) &&
               (ranNum <= gCompensateRatio))
         {
            compensateFlag = TRUE;

            do
               foundMove = MoveCompensate(&resNum);
            while(!foundMove);
            ++gNumMoveCompAtt[resNum - 1];
         }
         else if((gTunnelRatio > 0.0) &&
               (ranNum <= gTunnelRatio + gCompensateRatio) &&
               (ranNum > gCompensateRatio))
         {
            tunnelFlag = TRUE;

            do
               foundMove = MoveTunnel(&resNum);
            while(!foundMove);
            ++gNumMoveTunnAtt[resNum - 1];
         }
#ifndef NORMTORC
         else if((gTorsionRatio > 0.0) &&
               (ranNum <= gTorsionRatio + gTunnelRatio + gCompensateRatio) &&
               (ranNum > gCompensateRatio + gTunnelRatio))
#else
         else
#endif
         {
            torsionFlag = TRUE;

            do
            {
               limitLineNum = (int)
                     (gNumLimits * random() / (gMaxRand + 1.0));
               mag = GetMag(gLimits[limitLineNum].low,
                     gLimits[limitLineNum].high) * gdRadPerDeg;
               foundMove = RotateAtoms(gAtoms,
                                       gLimits[limitLineNum].resNum,
                                       gLimits[limitLineNum].bondName,
                                       mag);
               resNum = gLimits[limitLineNum].resNum;
            }
            while(!foundMove);
            ++gNumMoveTorsAtt[resNum - 1];
         }
#ifndef NORMTORC
         else if((gAtomRatio > 0.0) &&
               (ranNum <= gAtomRatio + gTorsionRatio +
                     gTunnelRatio + gCompensateRatio) &&
               (ranNum > gTorsionRatio + gCompensateRatio + gTunnelRatio))
         {
            atomFlag = TRUE;
            MoveAtoms(gAtoms, gDiffusion);
            ++numAttAtom;
         }
         else
         {
            fprintf(stderr, "\n");
            fprintf(stderr, "No move possible.\n");
            fprintf(stderr, "\n");
            exit(1);
         }

         MakeXYZArray(gAtoms, gXArray, gYArray, gZArray);
#endif

         newPenalty = CalcTotP(ldNoPrint);
         ++totalCalc;
/*
         if(newPenalty < lowestPenalty)
         {
            lowestPenalty = newPenalty;
            CopyXYZ(gAtoms, gAtomsLowest);
         }
*/

         keepMove = Metropolis(newPenalty - penalty);
         if(keepMove)
         {
            penalty = newPenalty;
            CopyXYZ(gAtoms, gAtomsLastState);
#ifndef NORMTORC
            for(count = 0; count < gNumAtoms; ++count)
            {
               gXForceOldState[count] = gXForce[count];
               gYForceOldState[count] = gYForce[count];
               gZForceOldState[count] = gZForce[count];
            }
#endif
            ++numSuccess;
            ++successfulMoves;

            if(compensateFlag) ++gNumMoveCompAcc[resNum - 1];
            else if(tunnelFlag) ++gNumMoveTunnAcc[resNum - 1];
            else if(atomFlag) ++numAccAtom;
            else if(torsionFlag) ++gNumMoveTorsAcc[resNum - 1];

            if((gOriFlag) && (successfulMoves % gMovePerOri == 0))
            {
               CalcOri("N-H", totalCalc, ldNoPrint);
               CalcOri("C-O", totalCalc, ldNoPrint);
               fflush(gOriFile);
            }

            if((gTrajFlag) && (successfulMoves % gMovePerTraj == 0))
            {
               fprintf(gTrajFile, "%i", totalCalc);
               fprintf(gTrajFile, "\t%.4e", gTemperature);
               fprintf(gTrajFile, "\t%.4f", gCalcNCS);
               fprintf(gTrajFile, "\t%.4f", gCalcCCS);
               fprintf(gTrajFile, "\t%.4f", gCalcICS);
               fprintf(gTrajFile, "\t%.4f", gCalcNC);
               fprintf(gTrajFile, "\t%.4f", gCalcNH);
               fprintf(gTrajFile, "\t%.4f", gCalcIH);
               fprintf(gTrajFile, "\t%.4f", gCalcCD);
#ifndef NORMTORC
               fprintf(gTrajFile, "\t%.4f", gCalcE);
#endif
               fprintf(gTrajFile, "\t%.4f", gCalcNCSP);
               fprintf(gTrajFile, "\t%.4f", gCalcCCSP);
               fprintf(gTrajFile, "\t%.4f", gCalcICSP);
               fprintf(gTrajFile, "\t%.4f", gCalcNCP);
               fprintf(gTrajFile, "\t%.4f", gCalcNHP);
               fprintf(gTrajFile, "\t%.4f", gCalcIHP);
               fprintf(gTrajFile, "\t%.4f", gCalcDISP);
               fprintf(gTrajFile, "\t%.4f", gCalcCDP);
#ifndef NORMTORC
               fprintf(gTrajFile, "\t%.4f", gCalcEP);
#endif
               fprintf(gTrajFile, "\t%.4f", gCalcTotP);
               fprintf(gTrajFile, "\n");
               fflush(gTrajFile);
            }
/*
            if(highMag < fabs(mag))
               highMag = fabs(mag);
            if(lowMag > fabs(mag))
               lowMag = fabs(mag);
*/
         }
         else
         {
            CopyXYZ(gAtomsLastState, gAtoms);
#ifndef NORMTORC
            MakeXYZArray(gAtoms, gXArray, gYArray, gZArray);
            for(count = 0; count < gNumAtoms; ++count)
            {
               gXForce[count] = gXForceOldState[count];
               gYForce[count] = gYForceOldState[count];
               gZForce[count] = gZForceOldState[count];
            }
#endif
         }

         if((gHistoryFlag) &&
               (totalCalc % gMovePerHistory == 0))
         {
            ++historyCount;
            sprintf(historyName, "%s/%s%04d.pdb",
                  dirName, "history/coord", historyCount);
            OutputPDB(historyName, gAtoms);
         }

         if(numIterAtT >= gCyclesAtT)
            continueAtT = FALSE;
         if(numSuccess >= gCyclesForceT)
            continueAtT = FALSE;
         if(penalty == 0.0)
            continueAtT = FALSE;
      }
      while(continueAtT);

      fprintf(stdout, "\npenalty     = %17.11f\n", penalty);
      fprintf(stdout, "temperature = %14.4e\n", gTemperature);
/*
      if(numSuccess == 0)
      {
         highMag = 0.0;
         lowMag  = 0.0;
      }
#ifdef NORMTORC
      fprintf(stdout, "highMag     = %17.11f\n", highMag / gdRadPerDeg);
      fprintf(stdout, "lowMag      = %17.11f\n", lowMag / gdRadPerDeg);
#else
      if(gTorsionRatio > 0.0)
      {
         fprintf(stdout, "highMag     = %17.11f\n", highMag / gdRadPerDeg);
         fprintf(stdout, "lowMag      = %17.11f\n", lowMag / gdRadPerDeg);
      }
#endif
*/
      fprintf(stdout, "numSuccess  = %5d", numSuccess);
      fprintf(stdout, " (force %d)", gCyclesForceT);
      fprintf(stdout, " (tried %d)", numIterAtT);
      fprintf(stdout, " (new %d)\n", gCyclesAtT);
      fflush(stdout);

/*
      if((numSuccess == 0) && (lowestPenalty < penalty))
      {
         fprintf(stdout, "\nJump starting\n");
         fprintf(stdout, "Lowest penalty found = %.11f\n", lowestPenalty);
         fprintf(stdout, "Setting numSuccess to 1\n");
         fflush(stdout);
         numSuccess = 1;
         CopyXYZ(gAtomsLowest, gAtoms);
#ifndef NORMTORC
         MakeXYZArray(gAtoms, gXArray, gYArray, gZArray);
#endif
      }
*/

      if(totalCalc >= gEquilSteps)
         gTemperature *= gTempFactor;

      if(gTCycles > 0)
      {
         if(numTIterations >= gTCycles)
            continueT = FALSE;
      }
      else
      {
         if(numSuccess == 0)
            continueT = FALSE;
         if(penalty == 0.0)
            continueT = FALSE;
      }
   }
   while(continueT);

   timeEnd = time(0);
   times(&cpuTimeEnd);
   cpuUserTime = ((float)(cpuTimeEnd.tms_utime - cpuTimeStart.tms_utime)) /
         ((float)(CLK_TCK));
   cpuSystemTime = ((float)(cpuTimeEnd.tms_stime - cpuTimeStart.tms_stime)) /
        ((float)(CLK_TCK));
   cpuTimeTotal = cpuUserTime + cpuSystemTime;

   fprintf(stdout, "\n\nCompleted in %d seconds (%.2f hours)\n",
         timeEnd - timeStart, (timeEnd - timeStart) / 3600.0);
   fprintf(stdout, "Completed in %.4f cpu seconds (%.2f hours)\n",
         cpuTimeTotal, cpuTimeTotal / 3600.0);
   fprintf(stdout, "Completed %d temperature iterations\n", numTIterations);
   fprintf(stdout, "Completed %d successful moves\n",
         successfulMoves);
   fprintf(stdout, "Completed %.4f successful moves/cpu second\n",
         (float) successfulMoves / cpuTimeTotal);
   fprintf(stdout, "Completed %d calculations\n", totalCalc);
   fprintf(stdout, "Completed %.4f calculations/cpu second\n",
         (float) totalCalc / cpuTimeTotal);
   fprintf(stdout, "\n");

   fprintf(stdout, "residue      Co (acc/att)");
   fprintf(stdout, "      Tu (acc/att)      To (acc/att)\n");
   fprintf(stdout, "------- -----------------");
   fprintf(stdout, " ----------------- -----------------\n");
   for(count = 0; count < gNumRes; ++count)
   {
      fprintf(stdout, "%7d %7d / %7d %7d / %7d %7d / %7d\n",
            count + 1,
            gNumMoveCompAcc[count],
            gNumMoveCompAtt[count],
            gNumMoveTunnAcc[count],
            gNumMoveTunnAtt[count],
            gNumMoveTorsAcc[count],
            gNumMoveTorsAtt[count]);

      numAttCompensate += gNumMoveCompAtt[count];
      numAccCompensate += gNumMoveCompAcc[count];
      numAttTunnel += gNumMoveTunnAtt[count];
      numAccTunnel += gNumMoveTunnAcc[count];
      numAttTorsion += gNumMoveTorsAtt[count];
      numAccTorsion += gNumMoveTorsAcc[count];
   }
   fprintf(stdout, "\n");

   fprintf(stdout, "              accepted attempted");
   fprintf(stdout, " acc/att acc/tot att/tot\n");
   if(gCompensateRatio > 0.0)
   {
      fprintf(stdout, "compensate  %9d %9d",
            numAccCompensate,
            numAttCompensate);
      fprintf(stdout, " %7.2f",
            (double) numAccCompensate / numAttCompensate);
      fprintf(stdout, " %7.2f", (double) numAccCompensate / totalCalc);
      fprintf(stdout, " %7.2f\n", (double) numAttCompensate / totalCalc);
   }
   if(gTunnelRatio > 0.0)
   {
      fprintf(stdout, "tunnel      %9d %9d", numAccTunnel, numAttTunnel);
      fprintf(stdout, " %7.2f", (double) numAccTunnel / numAttTunnel);
      fprintf(stdout, " %7.2f", (double) numAccTunnel / totalCalc);
      fprintf(stdout, " %7.2f\n", (double) numAttTunnel / totalCalc);
   }
#ifndef NORMTORC
   if(gTorsionRatio > 0.0)
   {
#endif
      fprintf(stdout, "torsion     %9d %9d", numAccTorsion, numAttTorsion);
      fprintf(stdout, " %7.2f", (double) numAccTorsion / numAttTorsion);
      fprintf(stdout, " %7.2f", (double) numAccTorsion / totalCalc);
      fprintf(stdout, " %7.2f\n", (double) numAttTorsion / totalCalc);
#ifndef NORMTORC
   }
   if(gAtomRatio > 0.0)
   {
      fprintf(stdout, "atom        %9d %9d", numAccAtom, numAttAtom);
      fprintf(stdout, " %7.2f", (double) numAccAtom / numAttAtom);
      fprintf(stdout, " %7.2f", (double) numAccAtom / totalCalc);
      fprintf(stdout, " %7.2f\n", (double) numAttAtom / totalCalc);
   }
#endif
   fprintf(stdout, "\n");

   if(gTrajFlag)
   {
      fprintf(gTrajFile, "%i", totalCalc);
      fprintf(gTrajFile, "\t%.4e", gTemperature / gTempFactor);
      CalcTotP(ldNoPrint);
      fprintf(gTrajFile, "\t%.4f", gCalcNCS);
      fprintf(gTrajFile, "\t%.4f", gCalcCCS);
      fprintf(gTrajFile, "\t%.4f", gCalcICS);
      fprintf(gTrajFile, "\t%.4f", gCalcNC);
      fprintf(gTrajFile, "\t%.4f", gCalcNH);
      fprintf(gTrajFile, "\t%.4f", gCalcIH);
      fprintf(gTrajFile, "\t%.4f", gCalcCD);
#ifndef NORMTORC
      fprintf(gTrajFile, "\t%.4f", gCalcE);
#endif
      fprintf(gTrajFile, "\t%.4f", gCalcNCSP);
      fprintf(gTrajFile, "\t%.4f", gCalcCCSP);
      fprintf(gTrajFile, "\t%.4f", gCalcICSP);
      fprintf(gTrajFile, "\t%.4f", gCalcNCP);
      fprintf(gTrajFile, "\t%.4f", gCalcNHP);
      fprintf(gTrajFile, "\t%.4f", gCalcIHP);
      fprintf(gTrajFile, "\t%.4f", gCalcDISP);
      fprintf(gTrajFile, "\t%.4f", gCalcCDP);
#ifndef NORMTORC
      fprintf(gTrajFile, "\t%.4f", gCalcEP);
#endif
      fprintf(gTrajFile, "\t%.4f", gCalcTotP);
      fprintf(gTrajFile, "\n");
      fclose(gTrajFile);
   }

   if(gHistoryFlag)
   {
      ++historyCount;
      sprintf(historyName, "%s/%s%04d.pdb",
            dirName, "history/coord", historyCount);
      OutputPDB(historyName, gAtoms);
   }
   else
   {
      fprintf(stdout, "Saving final coordinates as: ");
      fprintf(stdout, "%s\n", coordOutName);
      OutputPDB(coordOutName, gAtoms);
   }
   fprintf(stdout, "\n\n");

   newPenalty = CalcTotP(ldYesPrint);
   CalcTorsion();
   CalcResPT();

   fprintf(stdout, "Residue Bond   Orient  Angle Direct\n");
   fprintf(stdout, "------- ---- -------- ------ ------\n");
   CalcOri("N-H", totalCalc, ldYesPrint);
   fprintf(stdout, "\n");
   CalcOri("C-O", totalCalc, ldYesPrint);
   if(gOriFlag)
      fclose(gOriFile);

   fprintf(stdout, "\n");
   CalcDeltaR();
   fprintf(stdout, "\n");
   fflush(stdout);

   return(1);
```

```
}
#undef MAIN
#undef ldNoPrint
#undef ldYesPrint

/*
File Transpose.c
Transposes a 3x3 matrix.
*/

#include "Torc.h"

void Transpose(double matrixIn[3][3],
               double matrixOut[3][3])
{
   int   row, col;

   for(row = X; row <= Z; ++row)
      for(col = X; col <= Z; ++col)
         matrixOut[col][row] = matrixIn[row][col];
}

/*
File Vectorize.c
Creates a vector from two atoms.
*/

#include "Torc.h"

void Vectorize(double atom1[3],
               double atom2[3],
               double vector[3])
{
   vector[X] = atom2[X] - atom1[X];
   vector[Y] = atom2[Y] - atom1[Y];
   vector[Z] = atom2[Z] - atom1[Z];
}
```

A.4.3.2 TORC Makefile. The following Makefile is used to compile the TORC code into a stand alone program. It has been designed to compile on a Silicon Graphics 4×R8000 Power Challenge, but should work on other platforms with minor modification.

```
CFLAGS=-O2 -mips4 -align64 -DNORMTORC -woff all
LIBNAME=torc64
EXENAME=torc64.exe
LDFLAGS=-lm
LIB=../lib

.c.a :
	$(CC) $(CFLAGS) -c $<
	ar rv $@ $*.o
	/bin/rm -f $*.o
	@echo " "

OBJS = \
	$(LIB)/$(LIBNAME).a(CalcCCSP.o) \
	$(LIB)/$(LIBNAME).a(CalcCDP.o) \
	$(LIB)/$(LIBNAME).a(CalcDeltaR.o) \
	$(LIB)/$(LIBNAME).a(CalcDisP.o) \
	$(LIB)/$(LIBNAME).a(CalcICSP.o) \
	$(LIB)/$(LIBNAME).a(CalcIHP.o) \
	$(LIB)/$(LIBNAME).a(CalcNCP.o) \
	$(LIB)/$(LIBNAME).a(CalcNCSP.o) \
	$(LIB)/$(LIBNAME).a(CalcNHP.o) \
	$(LIB)/$(LIBNAME).a(CalcNorm.o) \
	$(LIB)/$(LIBNAME).a(CalcOri.o) \
	$(LIB)/$(LIBNAME).a(CalcPenalty.o) \
	$(LIB)/$(LIBNAME).a(CalcResPT.o) \
	$(LIB)/$(LIBNAME).a(CalcTheta.o) \
	$(LIB)/$(LIBNAME).a(CalcTorAngle.o) \
	$(LIB)/$(LIBNAME).a(CalcTorsion.o) \
	$(LIB)/$(LIBNAME).a(CalcTotP.o) \
	$(LIB)/$(LIBNAME).a(CopyCoord.o) \
	$(LIB)/$(LIBNAME).a(CopyXYZ.o) \
	$(LIB)/$(LIBNAME).a(DeclArray.o) \
	$(LIB)/$(LIBNAME).a(DoRotation.o) \
	$(LIB)/$(LIBNAME).a(DotProd.o) \
	$(LIB)/$(LIBNAME).a(EulerMatrix.o) \
	$(LIB)/$(LIBNAME).a(FindAtom.o) \
	$(LIB)/$(LIBNAME).a(GaussRand.o) \
	$(LIB)/$(LIBNAME).a(GetMag.o) \
	$(LIB)/$(LIBNAME).a(MF_LF.o) \
	$(LIB)/$(LIBNAME).a(MakeXYZArray.o) \
	$(LIB)/$(LIBNAME).a(MatMult.o) \
	$(LIB)/$(LIBNAME).a(Metropolis.o) \
	$(LIB)/$(LIBNAME).a(MotavgTensor.o) \
	$(LIB)/$(LIBNAME).a(MoveAtoms.o) \
	$(LIB)/$(LIBNAME).a(MoveCompensate.o) \
	$(LIB)/$(LIBNAME).a(MoveTunnel.o) \
	$(LIB)/$(LIBNAME).a(Normalize.o) \
	$(LIB)/$(LIBNAME).a(OutputPDB.o) \
	$(LIB)/$(LIBNAME).a(PAS_MF.o) \
	$(LIB)/$(LIBNAME).a(PrintHelp.o) \
        $(LIB)/$(LIBNAME).a(ReadCDData.o) \
        $(LIB)/$(LIBNAME).a(ReadCData.o) \
        $(LIB)/$(LIBNAME).a(ReadContFile.o) \
        $(LIB)/$(LIBNAME).a(ReadDisFile.o) \
        $(LIB)/$(LIBNAME).a(ReadFiles.o) \
        $(LIB)/$(LIBNAME).a(ReadIData.o) \
        $(LIB)/$(LIBNAME).a(ReadIHData.o) \
        $(LIB)/$(LIBNAME).a(ReadLamFile.o) \
        $(LIB)/$(LIBNAME).a(ReadLimFile.o) \
        $(LIB)/$(LIBNAME).a(ReadNCData.o) \
        $(LIB)/$(LIBNAME).a(ReadNData.o) \
        $(LIB)/$(LIBNAME).a(ReadNHData.o) \
        $(LIB)/$(LIBNAME).a(ReadPDBFile.o) \
        $(LIB)/$(LIBNAME).a(RotateAtoms.o) \
        $(LIB)/$(LIBNAME).a(Torc.o) \
        $(LIB)/$(LIBNAME).a(Transpose.o) \
        $(LIB)/$(LIBNAME).a(Vectorize.o)

$(EXENAME) : $(OBJS)
        @echo "Linking"
        ${CC} ${CFLAGS} -o $(EXENAME) $(LIB)/$(LIBNAME).a ${LDFLAGS}
        @echo " "
        @echo "Succesfully Completed make"
        @echo "executable is "$(EXENAME)

$(OBJS) : Torc.h
```

A.4.3.3 TORC/CHARMM sample input file. The following inout file is used to load the necessary parameters and start the refinement within CHARMM.

```
* file atom1.inp
*

!read topology
open read card unit 10 name top_all22_prot.inp.hbond
read rtf card unit 10
close unit 10

!read parameters
open read card unit 10 name par_all22_prot.inp.hbond
read param card unit 10
close unit 10

read sequence card
* channel 1
*
  17
CHO VAL GLY ALA LEU ALA VAL VAL VAL TRP LEU TRP LEU TRP LEU TRP EAM
generate MONO first none last none setup

read imag card
* IMAGE FILE FOR THE GRAMICIDIN DIMER
* (Z AXIS ROTATION)
*
image XROT
ROTATE 1.0 0.0 0.0 180.0
END

open read card unit 10 name "ATOM1/coord.in.pdb"
read coor pdb unit 10
close unit 10

! Impose channel axis along z
MMFP
geo cylinder zdir 1.0 RCM force 100.0 select all end
END

!Electrostatics with a dielectric constant eps=1
update  inbfrq    -1   -
        ctonnb   8.0 ctofnb   10.0 cutnb   11.0 cutim   11.0 wmin     0.5 -
        elec         switch        group        cdie         eps      1.0 -
        vdw          vswitch       vgroup

!declare the 6 monomer-monomer hbonds
IMPATCH HBOH   PRIM MONO 2    XROT MONO 6  setup
IMPATCH HBOH   PRIM MONO 4    XROT MONO 4  setup
IMPATCH HBOH   PRIM MONO 6    XROT MONO 2  setup

IMPATCH HBHO   PRIM MONO 2    XROT MONO 6  setup
IMPATCH HBHO   PRIM MONO 4    XROT MONO 4  setup
IMPATCH HBHO   PRIM MONO 6    XROT MONO 2  setup

update
energy
```

```
!start the TOtal Refinement of Constraints

TORC TDIR ATOM1

open read card unit 10 name "ATOM1/coord.out.pdb"
read coor pdb unit 10
close unit 10
energy

stop
```

REFERENCES

Abragam, A. (1961). Principles of Nuclear Magnetism. New York, Oxford University Press.

Akashi, K., Kubota, K. and Kurahashi, K. (1977). "Biosynthesis of enzyme-bound formylvaline and formylvalylglycine. A possible initiation complex for gramicidin A biosynthesis." Journal of Biochemistry **81**(1): 269-72.

Akashi, K. and Kurahashi, K. (1977). "Formylation of enzyme-bound valine and stepwise elongation of intermediate peptides of gramicidin A by a cell-free enzyme system." Biochemical and Biophysical Research Communications **77**(1): 259-67.

Altman, R. B. and Jardetzky, O. (1989). "Heuristic refinement method for determination of solution structure of proteins from nuclear magnetic resonance data." Meth. Enzymol. **177**: 218-246.

Bamberg, E. and Lauger, P. (1987). "Blocking of the gramicidin channel by divalent cations." J. Membr. Biol. **35**: 351-375.

Braun, W. (1987). "Distance geometry and related methods for protein structure determination from NMR data." Quart. Rev. Biophys. **19**: 115-157.

Brenneman, M., Quine, J. and Cross, T. A. (unpublished results). .

Brenneman, M. T. and Cross, T. A. (1990). "A Novel Method for the Analytical Determination of Protein Structure Using Solid State NMR: the "Metric Method"." Journal of Chemical Physics **92**: 1483-1494.

Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S. and Karplus, M. (1983). "CHARMM: A program for macromolecular energy, minimization, and dynamics calculations." Journal of Computational Chemistry **4**: 187-217.

Brünger, A. T., Clore, G. M., Gronenborn, A. M. and Karplus, M. (1986). "Three-dimensional structure of proteins determined by molecular dynamics with interproton distance restraints: application to crambin." Proc. Natl. Acad. Sci. USA **83**: 3801-3805.

Brünger, A. T. and Karplus, M. (1991). "Molecular dynamics simulations with experimental restraints." Acc. Chem. Res. **24**: 54-61.

Bystrov, V. F., Arseniev, A. S., Barsukov, I. L. and Lomize, A. L. (1987). "2D NMR of single and double stranded helices of gramicidin A in micelles and solutions." Bull. Mag. Resonance **8**: 84-94.

Case, D. A. and Wright, P. E. (1993). Determination of high-resolution NMR structures of proteins. NMR of Proteins: 53-91.

Clore, G. M. and Gronenborn, A. M. (1989). "Determination of the three-dimensional structures of proteins and nucleic acids in solution by nuclear magnetic resonance spectroscopy." CRC Crit. Rev. Biochem. Mol. Biol. **24**: 479-564.

Clore, G. M. and Gronenborn, A. M. (1991). "Applications of three- and four-dimensional heteronuclear NMR spectroscopy to protein structure determination." Prog. NMR Spectrosc. **23**: 43-92.

Clore, G. M., Gronenborn, A. M., Brünger, A. T. and Karplus, M. (1985). "Solution conformation of a heptadecapeptide comprising the DNA binding helix F of the Cyclic AMP receptor protein of Escherichia coli. Combined use of ^{1}H nuclear magnetic resonance and restrained molecular dynamics." J. Mol. Biol. **186**: 435-455.

Cross, T. A., Ketchem, R. R., Hu, W., Lee, K.-C., Lazo, N. D. and North, C. L. (1992). "Structure and Dynamics of a Membrane Bound Polypeptide." Bulletin of Magnetic Resonance **14**: 96-101.

Cross, T. A. and Opella, S. J. (1983). "Protein Structure by Solid State NMR." Journal of the American Chemical Society **105**: 306-308.

Engh, R. A. and Huber, R. (1991). "Accurate bond and angle parameters for x-ray protein structure refinement." Acta Cryst. **A47**: 392-400.

Fields, C. G., Fields, G. B., Noble, R. L. and Cross, T. A. (1989). "Solid phase peptide synthesis of ^{15}N-gramicidins A, B, and C and high performance liquid chromatographic purification." International Journal of Peptide and Protein Research **33**(4): 298-303.

Fields, G. B., Fields, C. G., Petefish, J., Van Wart, H. E. and Cross, T. A. (1988). "Solid-phase peptide synthesis and solid-state NMR spectroscopy of [Ala_3-^{15}N][Val_1]gramicidin A." Proceedings of the National Academy of Sciences of the United States of America **85**(5): 1384-8.

Fletterick, R. J., Tsai, C.-C. and Hughes, R. E. (1971). "The crystal and molecular structure of L-alanine-L-alanine." J. Phys. Chem. **75**: 918-922.

Fuller, G. H. (1976). J. Phys. Chem. Ref. Data **5**: 835.

Gippert, G. P., Yip, P. F., Wright, P. E. and Case, D. A. (1990). "Computational methods for determining protein structures from NMR data." Biochem. Pharmacol. **40**: 15-22.

Griffiths, J. M. and Griffin, R. G. (1993). "Nuclear Magnetic Resonance Methods for Measuring Dipolar Couplings in Rotating Solids." Anal Chim Acta **283**: 1081-1101.

Gullion, T. and Schaefer, J. (1989). "Rotational-Echo Double-Resonance NMR." Journal of Magnetic Resonance **81**: 196-200.

Hahn, E. L. (1950). "Spin Echoes." Phys. Rev. **80**: 580-594.

Harold, F. M. and Baarada, J. R. (1967). "Gramicidin, valinomycin, and cation permeability of Stretococcus faecalis." J. Bact. **94**: 53-60.

Hartmann, S. R. and Hahn, E. L. (1962). "Nuclear double resonance in the rotating frame." Phys. Rev. **128**: 2042-2053.

Havel, T. F. (1991). "An evaluation of computational strategies for use in the determination of protein structure from distance constraints obtained by nuclear magnetic resonance." Prog. Biophys. Mol. Biol. **56**: 43-78.

Havel, T. F. and Wüthrich, K. (1985). "An evaluation of the combined use of nuclear magnetic resonance and distance geometry for the determination of protein conformations in solution." J. Mol. Biol. **182**: 281-294.

Hotchkiss, R. D. (1944). "Gramicidin, tyrocidin and tyrothricin." Advan. Enzymol. **4**: 153-199.

Hu, W., Lee, K. C. and Cross, T. A. (1993). "Tryptophans in membrane proteins: indole ring orientations and functional implications in the gramicidin channel." Biochemistry **32**(27): 7035-47.

Jeffrey, G. A. and Saenger, W. (1994). Hydrogen bonding in biological structures, Springer-Verlag.

Jordan, P. C. (1987). "Microscopic approaches to ion transport through transmembrane channels: the model system gramicidin." J. Phys. Chem. **91**: 6582-6591.

Katsaras, J., Prosser, R. S., Stinson, R. H. and Davis, J. H. (1992). "Constant helical pitch of the gramicidin channel in phospholipid bilayers." Biophysical Journal **61**(3): 827-30.

Katz, E. and Demain, A. L. (1977). "The peptide antibiotics of Bacillus: chemistry, biogenesis, and possible functions." Bacteriol. Rev. **41**: 449-474.

Ketchem, R. R., Hu, W. and Cross, T. A. (1993). "High-resolution conformation of gramicidin A in a lipid bilayer by solid-state NMR." Science **261**(5127): 1457-60.

Killian, J. A., Nicholson, L. K. and Cross, T. A. (1988). "Solid-state ^{15}N-NMR evidence that gramicidin A can adopt two different backbone conformations in dimyristoylphosphatidylcholine model membrane preparations." Biochimica et Biophysica Acta **943**(3): 535-40.

Kirkpatrick, S., Gelatt Jr., C. D. and Vecchi, M. P. (1983). "Optimization by simulated annealing." Science **220**: 671-680.

Kurahashi, K. (1981). "Biosynthesis of peptide antibiotics." Antibiotics **4**: 325-352.

Kvick, A., Al-Karaghouli, A. R. and Koetzle, T. F. (1977). "Deformation electron density of α-glycylglycine at 82K. I. The neutron diffraction study." Acta Crystallogr. sect. B **33**: 3796-3801.

Langs, D. A., Smith, G. D., Courseille, C., Precigoux, G. and Hospital, M. (1991). "Monoclinic uncomplexed double-stranded, antiparallel, left-handed beta 5.6-helix (increases decreases beta 5.6) structure of gramicidin A: alternate patterns of helical association and deformation." Proceedings of the National Academy of Sciences of the United States of America **88**(12): 5345-9.

Laskowski, R. A., MacArthur, M. W., Moss, D. S. and Thornton, J. M. (1993). "PROCHECK: a program to check the stereochemical quality of." Journal of Applied Crystallography **26**: 283-291.

Lazo, N. D., Hu, W. and Cross, T. A. (1995). "Low-temperature solid-state ^{15}N NMR characterization of polypeptide backbone librations." Journal of Magnetic Resonance. Series B **107**(1): 43-50.

Lazo, N. D., Hu, W., Lee, K. C. and Cross, T. A. (1993). "Rapidly-frozen polypeptide samples for characterization of high definition dynamics by solid-state NMR spectroscopy." Biochemical and Biophysical Research Communications **197**(2): 904-9.

Lee, K. C. and Cross, T. A. (1994). "Side-chain structure and dynamics at the lipid-protein interface: Val_1 of the gramicidin A channel." Biophysical Journal **66**(5): 1380-7.

Lee, K. C., Huo, S. and Cross, T. A. (1995). "Lipid-peptide interface: valine conformation and dynamics in the gramicidin channel." Biochemistry **34**(3): 857-67.

Lipmann, F. (1980). "Bacterial production of antibiotic polypeptides by thiol-linked synthesis on protein templates." Adv. Microb. Physiol. **21**: 227-266.

Logan, T. M., Zhou, M.-M., Nettesheim, D. G., Meadows, R. P., Van Etten, R. L. and Fesik, S. W. (1994). "Solution structure of a low molecular weight protein tyrosine phosphatase." Biochemistry **33**: 11087-11096.

LoGrasso, P. V., Moll, F. d. and Cross, T. A. (1988). "Solvent history dependence of gramicidin A conformations in hydrated lipid bilayers." Biophysical Journal **54**(2): 259-67.

LoGrasso, P. V., Nicholson, L. K. and Cross, T. A. (1989). "N-H bond length determination and implications for the gramicidin channel conformation and dynamics from ^{15}N-^{1}H dipolar interactions." J. Amer. Chem. Soc. **111**: 1910-1912.

Lomize, A. L., Orekhov, V. and Arseniev, A. S. (1992). "Utochnenie prostranstvennoi struktury ionnogo kanala gramitsidina A. [Refinement of the spatial structure of the gramicidin A ion channel]." Bioorganicheskaia Khimiia **18**(2): 182-200.

Mackerell, A. D. J., Bashford, D., Bellot, M., Dunbrack, R. L., Field, M. J., Fischer, S., J., G., Guo, H., Ha, S., Joseph, D., Kuchnir, L., Kuczera, K., Lau, F. T. K., Mattos, C., Michnick, S., Nguyen, D. T., Ngo, T., Prodhom, B., Roux, B., Schlenkrich, B., *et al.* (1992). "Self-Consistent Parameterization of Biomolecules for Molecular Modeling and Condensed Phase Simulations." Biophys. J. **61**: A143.

Mai, W., Hu, W., Wang, C. and Cross, T. A. (1993). "Orientational constraints as three-dimensional structural constraints from chemical shift anisotropy: the polypeptide backbone of gramicidin A in a lipid bilayer." Protein Science **2**(4): 532-42.

Mandl, J. and Paulus, H. (1985). "Effect of linear gramicidin on sporulation and intracellular ATP pools of Bacillus brevis." Archives of Microbiology **143**(3): 248-52.

Mehring, M. (1983). High resolution spectroscopy in solids.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. and Teller, E. (1953). "Equation of state calculations by fast computing machines." J. Chem. Phys. **21**: 1087-1092.

Moll, F. and Cross, T. A. (1990). "Optimizing and characterizing alignment of oriented lipid bilayers containing gramicidin D." Biophysical Journal **57**(2): 351-62.

Momany, F. A., McGuir, R. F., Burgess, A. W. and Scheraga, H. A. (1975). "Energy parameters in polypeptides. VII. Geometric parameters, partial atomic charges, nonbonded interactions, hydrogen bond interactions, and intrinsic torsional potentials for the naturally occurring amino acids." J. Phys. Chem. **79**: 2361-2378.

Nicholson, L. K. and Cross, T. A. (1989). "Gramicidin cation channel: an experimental determination of the right-handed helix sense and verification of beta-type hydrogen bonding." Biochemistry **28**(24): 9379-85.

Nicholson, L. K., Moll, F., Mixon, T. E., LoGrasso, P. V., Lay, J. C. and Cross, T. A. (1987). "Solid-state ^{15}N NMR of oriented lipid bilayer bound gramicidin A'." Biochemistry **26**(21): 6621-6.

Nicholson, L. K., Teng, Q. and Cross, T. A. (1991). "Solid-state nuclear magnetic resonance derived model for dynamics in the polypeptide backbone of the gramicidin A channel." Journal of Molecular Biology **218**(3): 621-37.

North, C. L. (1993). Peptide backbone librations of the gramicidin A transmembrane channel as measured by solid state nuclear magnetic resonance. Implications for proposed mechanisms of ion transport. Institute of Molecular Biophysics. Tallahassee, The Florida State University.

North, C. L. and Cross, T. A. (1993). "Analysis of Polypeptide Backbone T_1 Relaxation Data Using an Experimentally Derived Model." Journal of Magnetic Resonance **101B**: 35-43.

North, C. L. and Cross, T. A. (1995). "Correlations Between Function and Dynamics: Timescale Coincidence for Ion Translocation and Molecular Dynamics in the Gramicidin Channel Backbone." Biochemistry **34**: 5883-5895.

Opella, S. J., Stewart, P. L. and Valentine, K. G. (1987). "Protein structure by solid state NMR spectroscopy." Q. Rev. Biophys. **19**: 7-49.

Pascal, S. M. and Cross, T. A. (1992). "Structure of an isolated gramicidin A double helical species by high-resolution nuclear magnetic resonance." Journal of Molecular Biology **226**(4): 1101-9.

Pascal, S. M. and Cross, T. A. (1993). "High-resolution structure and dynamic implications for a double-helical gramicidin A conformer." Journal of Biomolecular NMR **3**(5): 495-513.

Peticolas, W. L. and Kurtz, B. (1980). "Transformation of the ϕ-ψ plot for proteins to a new representation with local helicity and peptide torsional angles as variables." Biopolymers **19**: 1153-1166.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. (1992). Numerical Recipes in C, Cambridge University Press.

Prosser, R. S., Davis, J. H., Dahlquist, F. W. and Lindorfer, M. A. (1991). "^{2}H nuclear magnetic resonance of the gramicidin A backbone in a phospholipid bilayer." Biochemistry **30**(19): 4687-96.

Ramachandran, G. N. and Sassiekharan, V. (1968). "Conformation of polypeptides and proteins." Adv. Prot. Chem. **28**: 283-437.

Roux, B. and Karplus, M. (1991a). "Ion transport in a gramicidin-like channel: dynamics and mobility." J. Phys. Chem. **95**: 4856-4868.

Roux, B. and Karplus, M. (1991b). "Ion transport in a model gramicidin channel. Structure and thermodynamics." Biophysical Journal **59**(5): 961-81.

Sanders, J. K. M. and Hunter, B. K. (1993). Modern NMR Spectroscopy A Guide for Chemists. New York, Oxford University Press.

Sarges, R. and Witkop, B. (1965). "Gramicidin A. V. The structure of valine- and isoleucine-gramicidin A." Journal of the American Chemical Society **87**: 2011-2020.

Sillescu, H. (1982). "Recent advances of ^{2}H NMR for studying molecules in solid polymers." Pure & Appl. Chem. **54**: 619.

Slichter, C. P. (1990). Principles of Magnetic Resonance. New York, Springer-Verlag.

Smith, S. O. (1993). "Magic Angle Spinning NMR Methods for Internuclear Distance Measurements." Curr. Opin. Struct. Biol. **3**: 755-759.

Spiess, H. W. (1983). "Molecular dynamics of solid polymers as revealed by deuteron NMR." Coll. & Polymer. Sci. **261**: 193-209.

Spiess, H. W. (1985). Deuteron NMR - a new tool for studying chain mobility and orientation in polymers. Adv. Polym. Sci. Vol. 66. New York, Springer-Verlag: 23-58.

Teng, Q., Iqbal, M. and Cross, T. A. (1992). "Determination of the ^{13}C Chemical Shift and ^{14}N Electric Field Gradient Tensor Orientations With Respect to the Molecular Frame in a Polypeptide." Journal of the American Chemical Society **114**: 5312-5321.

Teng, Q., Nicholson, L. K. and Cross, T. A. (1991). "Experimental determination of torsion angles in the polypeptide backbone of the gramicidin A channel by solid state nuclear magnetic resonance." Journal of Molecular Biology **218**(3): 607-19.

Urry, D. W. (1971). "The gramicidin A transmembrane channel: a proposed $\Pi_{(L,D)}$ helix." Proceedings of the National Academy of Sciences USA **68**: 672-676.

Wallace, B. A. and Janes, R. W. (1991). "Co-crystals of gramicidin A and phospholipid. A system for studying the structure of a transmembrane channel." Journal of Molecular Biology **217**(4): 625-7.

Wallace, B. A. and Ravikumar, K. (1988). "The gramicidin pore: crystal structure of a cesium complex." Science **241**: 182-187.

Waugh, J. S. (1976). "Uncoupling of local field spectra in nuclear magnetic resonance: determination of atomic positions in solids." Proceedings of the National Academy of Sciences of the United States of America **73**(5): 1394-7.

Weinstein, S., Wallace, B. A., Morrow, J. S. and Veatch, W. R. (1980). "Conformation of the gramicidin A transmembrane channel: A 13C nuclear magnetic resonance study of 13C-enriched gramicidin in phosphatidylcholine vesicles." Journal of Molecular Biology **143**(1): 1-19.

Wüthrich, K. (1989). "Protein structure determination in solution by nuclear magnetic resonance spectroscopy." Science **243**: 45-50.

BIOGRAPHICAL SKETCH

Randal R. Ketchem was born on July 26, 1965, in Spokane, Washington to Fred and Shareen Ketchem. He then moved to Montana, Idaho, South Carolina, the country of Panama, Florida, Georgia and back to Florida where he has lived ever since. He completed a B.S. in Chemistry/Biochemistry at the University of West Florida (where he also met his wife, Paula Ketchem) the Summer of 1989. The following Fall he began his Ph.D. in Molecular Biophysics at the Florida State University.