APPLICATION OF THE DIFFERENCE MAP ALGORITHM TO PROTEIN FOLDING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ivan Christopher Rankenburg

August 2007

APPLICATION OF THE DIFFERENCE MAP ALGORITHM TO PROTEIN FOLDING

Ivan Christopher Rankenburg, Ph.D.

Cornell University 2007

This dissertation focuses on the application of a new search algorithm, the difference map, to the problem of protein structure prediction.

First a short review of protein structure is given to explain the terms and concepts used in the following chapters. A brief description of the current structure determination techniques (x-ray diffraction, NMR) are given, but the bulk of this dissertation is focused on ab initio structure prediction. Using ab initio methods, the native fold of a protein is assumed to be the global minimum of a high dimensional energy function. The current methods for minimizing this energy function are reviewed. The next chapter introduces a new search algorithm, the difference map. The difference map has been applied to many fields and problems where an exhaustive search is not feasible. A brief description of its historical development is given. Following this is an explanation of how the algorithm efficiently searches a high dimensional search space. A program called the difference map explorer was created to explore the effects of various difference map parameters on the search dynamics of the algorithm.

The application of the difference map to the problem of protein energy minimization is demonstrated in chapter 3. In this chapter, the rate at which the difference map produces low energy protein conformations is compared with that of a Monte Carlo based search algorithm, parallel tempering. It is shown that the difference map finds low en-

ergy protein conformations at a significantly higher rate than parallel tempering. The final chapter describes in detail NENA, the software implementation of the difference map folding algorithm.

BIOGRAPHICAL SKETCH

I was born to Bernhold and Yolanda Rankenburg in 1978. They named me Ivan. It is an unusual name in America, and it has turned me into an unusual person. I spent my first five years in Berkeley California, where my fondest memories to this day are lying in cool, shady, ivy patches, with the smell of earth surrounding me.

In 1983 we moved to Napa California, into the house where my parents still live. I had a rough childhood: I broke my arm on a playground, I fell off a bridge and hurt my back (an injury I carry still), and I had a tooth knocked out. I challenged my elementary school teachers constantly, and as a result I spent almost as much time in the principal's office as in the classroom. High school was challenging. The classes were interesting, absorbing, and trivially easy. It was a challenge to cohabitate with the other adolescents. Everybody wanted to be unique, but nobody had the courage to be different. I turned away from social interaction, to a large degree, and enjoyed math puzzles. My math puzzles kept me company; they were always available, and always entertaining. The interaction of numbers was predictable, the interactions of people were confusing.

I carried this personal affinity for math (and fear of social interaction) into college at U.C. Davis. There I immediately decided on math and physics for my two majors. Over the following 5 years I learned people are often superior company to math problems. I also learned that though I enjoyed math more than physics, physics came easier to me.

Finding the Elser group at Cornell was a Godsend. In this group I was able to ostensibly do physics, but be involved with a wide range of interesting math problems. My fellow colleagues in the group are some of the nicest, kindest people I have ever known. For the last three years I have had good company, both with fine math problems and with fine people. I am sure I will remember this period in my life as one my happiest.

To my Parents who gave me life, my God Thor who gave me strength, and Veit, who gave me knowledge.

ACKNOWLEDGEMENTS

So many people have helped me, it is absolutely impossible to remember and thank them all. I shall try.

Let me begin with thanking my parents, for giving me this fine body, and making me the person I have grown up to be. Let me thank my older brother for policing my bad behavior, and apologize to my little brother for the same bad behavior, and thank him for his forgiveness.

I must thank my inspirational high school teachers: Mr. Domizio (math), Mr. Scrivner (physics), Mr. Leasat (chemistry); each of whom molded me more than they can possible know.

At U.C. Davis, I first must thank Dr. Rena Zieve for her guidance, tolerance, and advice. Also, I thank Dr. Randy Harris for his friendship and support. Additional professors at U.C. Davis who helped me by giving me advice, or friendship, are: Dr. Erickson, Dr. Reid, Dr. Fuchs.

At Cornell, I cannot thank Veit Elser enough for giving me his friendship, interesting math problems, and a stimulating environment to thrive in. While working with Veit, both Duane (Nete) Loh and Pierre Thibault helped in ways beyond measure. In fact, it is safe to say that without their help this thesis would be far from begun. To my colleagues — I am very grateful.

This thesis is also the result of stimulating conversations with Simon Gravel, Buz Barstow, Nozomi Ando, Harold Scheraga, and Tomas Arias. Finally, I must acknowledge my friends who supported me in many ways for the past few years. I thank Eleni, Garnet, Andy, Arend, Lisa, Albert, and James. Thank you all for your friendship.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The solution to a computationally difficult problem can often be found by breaking the problem into two tractable sub-problems. The cost of this approach is that the two comparatively simple sub-problems need to have the same solution. In other words, a computationally difficult problem can be phrased so its that solution is located in the intersection of two constraint sets, where each constraint set is defined as the set of solutions to the corresponding sub-problem.

For example, consider the binary sequence, $\{0, 1, 1, 0, 1, 0, 1\}$. The magnitudes of its Fourier transform are (to six digits of precision)

$$\{1.51186, 0.209754, 0.303104, 0.849278, 0.849278, 0.303104, 0.209754\}.$$

If given these magnitudes, and the knowledge that they are derived from binary data, how hard is it to find the binary sequence that generated these magnitudes? This problem is indeed very difficult for longer sequences. However, like many difficult problems, this problem has the notable property that it can be phrased as a set intersection problem. One constraint set is the set of all binary vectors of length seven. This is not a large set. It has 2^7 discrete elements; for example $\{1, 1, 1, 1, 1, 0, 1\}$ and $\{0, 1, 0, 0, 0, 0, 1\}$. The other constraint set is the set of all real vectors of length seven, whose Fourier transform agrees with the given magnitudes. This set is continuous, rather than discrete. Like the binary constraint set, it is also easy to find an element in this set. To find an element, consider any set of arbitrary antisymmetric phases,[1] for example $\{0, 1.3, 2.7, -2.2, 2.2, -2.7, -1.3\}$. These phases can be assigned to the Fourier magni-

[1]The zero frequency phase must be zero. Also, the rest of the phases must be antisymmetric about the zero frequency. This is to ensure the sequence generated from these phases and the given magnitudes is real valued.

tudes, followed by an inverse Fourier transform, and the result,

$$\{0.0288832, 0.974077, 1.02533, -0.0281738, 1.00444, 0.000785885, 0.994657\}$$

will be both real valued, and its Fourier transform will agree with the given magnitudes.

In this example problem, elements of both constraint sets are easy to find,[2] but finding a vector of length seven that is an element of both constraint sets is challenging. For problems that can be reduced to a set intersection problem (like the above example), the *difference map* algorithm (DM) has been found to be very effective[15].

The DM is a generalization of Fienup's hybrid input-output phase retrieval algorithm[18] and the Douglas-Rachford algorithm.[5] Like its predecessors, the DM is an iterative algorithm. A complete explanation of its effectiveness at solving problems like the example above is currently mathematically unexplained. Even so, The dynamics of the DM, and analysis of how it works, is the subject of chapter 2.

A problem amenable to being split into two easier sub-problems is protein structure prediction. This problem is currently one of the most challenging subfields of computational structural biology and theoretical chemistry. As its name suggests, the goal is to predict, from the protein's sequence of amino acids, the three dimensional structure of the protein.

The three dimensional structure of a protein is becoming increasing more useful for highly targeted drug design, and therefore there is tremendous incentive to be able to determine the native structure of a protein. As an example of the utility of knowing a protein's structure, consider the recently developed drug Imatinib.[9] Imatinib is a new drug designed to fight chronic myelogenous leukemia. Imatinib was designed based on the protein structure of bcr-abl kinase, a protein essential for the spread of the disease.

[2]Indeed, even the *closest* element of a constraint set to a given input is easy to find. See chapter 2.1.2 for the significance of the closest element.

Imatinib selectively binds to the protein bcr-abl kinase, while not binding to other proteins. Once bound, the protein bcr-abl kinase no longer functions, and the detrimental effects of the disease are inhibited. Imatinib is one of the first drugs to be designed to target a specific harmful protein, while not affecting other essential proteins. Contrast this with chemotherapy. Typically, chemotherapeutic drugs target quickly dividing cells. Thus, they are effective against cancerous tumors, but they also destroy healthy normal cells, such as those responsible for hair growth, or for the replacement of the intestinal epithelium (intestinal lining). The side effects of chemotherapy can be significant, because unlike Imatinib, chemotherapeutic drugs don't target cancerous cells selectively enough.

A variety of techniques have been developed to determine the 3D structure of proteins. A complete review of these techniques, their effectiveness and applicability, and future prospects, is the subject of chapter 1.

Currently, the most successful technique for protein structure determination is x-ray crystallography, which accounts for about 36,000 out of the 42,000 determined structures in the Protein Data Bank (PDB).[3] By coaxing a protein to form a high quality crystal, x-ray diffraction is usually effective in determining the protein's structure. However, there are a large number of proteins that resist efforts to be crystallized. For these proteins, an alternative experimental technique, NMR spectroscopy, often is able to determine the protein's structure. Both methods rely on the presence of high concentrations of extremely pure protein samples. For many important proteins, obtaining a high enough concentration of pure protein is not yet possible. For these difficult (and very numerous) proteins, there is currently no method for structure determination. There is, however, a developing field of protein structure prediction.

[3]These numbers were obtained from the PDB website *http://www.rcsb.org/pdb/statistics/holdings.do* .

There are a variety of protein structure prediction techniques, and most of the well established ones will be described and explained in chapter 1.3. In chapter 3, we will demonstrate the effectiveness of the DM as a new and powerful technique for protein structure prediction. The native fold of a protein is found by splitting the problem of protein structure prediction into two easily solvable sub-problems. The DM is then used to find an atomic configuration that simultaneously solves both sub-problems.[4] The DM has already had significant success when applied to simple protein models,[14] and the data shown in chapter 3 suggest the DM is very effective for realistic protein models as well.

The application of the DM to the problem of protein structure prediction was conducted in the programing language C. To make the program easy to use for the general protein structure prediction community, a graphical user interface was developed in the programming language Python. These two programming languages are made compatible via a third programming language, Pyrex, and the complete package is called NENA. The complete source code for NENA can be found in appendix C.

Making precompiled C functions accessible by Python is not trivial. To demonstrate how it is done in NENA, a simpler program titled the difference map explorer (DMX) was developed. The two programs are structurally the same, though NENA is much more complicated. A thorough explanation of how DMX works is given in appendix B. This explanation is intended to give insight into how NENA is structured and operates.

It is our expectation that after I graduate from the Elser group, a new graduate student will continue the work we have begun. There is a lot of work still to be done, and preliminary results are very encouraging. Chapter 4 is a thorough explanation of how NENA works. It is meant to aid the student who adopts this project in understanding the

[4]The two sub-problems are explained in detail in appendix A.

precise details of how the program functions. Chapter 4 provides a detailed explanation of the current state of the software package, and proposes future additions to its current functions.

Chapter 1

Protein folding

1.1 What is a protein?

It is completely appropriate to begin with the etymology of the word *protein*. The word comes from the Greek word $\pi\rho\omega\tau\alpha$ ("prota"), which means "first" in modern Greek, and "of primary importance" in ancient Greek. The word "protein" was first used in 1838 by the Dutch chemist Gerhard Johan Mulder working with Friherre Jöns Jakob Berzelius (the father of modern chemistry). They identified and labeled this class of biological molecules by their nearly constant ratios of constituent elements.

A protein is a large biological molecule, composed of linked amino acids. The amino acids emerge from the ribosome joined together into a linear chain, linked by peptide bonds between the carboxyl group of one amino acid, and the amino group of the next. Genes in the cell nucleus encode the sequence of amino acids that constitute a particular protein. There are twenty types of naturally occurring amino acids and many synthetic, unnatural ones. A specific protein is identified by its sequence of amino acids, and so a protein can be thought of as a word composed of twenty different types of letters.[1] The smallest functioning proteins are around forty amino acids long, while the longest can be several thousand. The average sized protein consists of around three hundred amino acids. In 1955 the first protein (insulin) was sequenced by Sir Frederick Sanger. For this accomplishment, he won the Nobel Prize in Chemistry in 1958. Today, sequencing the

[1] For example, the staphylococcus aureus A protein (B domain) is given by the amino acid sequence TADNKFNKEQQNAFYEILHLPNLNEEQRNGFIQSLKDDPSQSAN-LLAEAKKLNDAQAPKA, where each letter stands for a specific amino acid (for example "A" represents alanine, "P" for proline ...). This sequence was obtained from the protein data bank.

amino acids that comprises a protein is considered a trivial problem.

All amino acids share the same basic structure: chirally bonded to a C_α atom is an amino group, a carboxyl group, and a distinguishing sidechain. The sidechain is what differentiates one amino acid from another, as can be seen in figure 1.1. The N, C_α, and C atoms common to all proteins link together and form the protein backbone. The only exception to this general amino acid structure is proline, also shown in figure 1.1. The sidechain of proline forms a bond to the protein backbone. This bond has important consequences for the internal degrees of freedom available to proline, as described below.

Many proteins aid in biological reactions as a catalyst (e.g. enzymes), while other proteins link together and form a rigid support network to maintain the cellular shape (e.g. actin). Still other proteins aid in a variety of essential cellular functions, such as cellular identification, cellular adhesion, and cellular division. For all of their purposes though, the function of a protein is completely determined by its 3D geometrical structure. The 3D structure of a protein is usually determined by its sequence of amino acids.[2]

For many years, it was thought that proteins did not have a well defined 3D structure, but instead were randomly coiled molten globules. Before 1938 the protein crystals scientists grew were not properly hydrated, and x-ray diffraction experiments regularly destroyed the crystals. Because of this, the innate 3D structure of a protein was obfuscated. However, perhaps due to not reading the literature, John Bernal and Dorothy Hodgkin tried the diffraction experiment with properly hydrated insulin crystals, and

[2]There are some exceptions to this rule. Often, the folding of a protein is done reversibly, and is self-driven. However, some proteins require the aid of chaperon proteins in the environment to fold successfully. Furthermore, the solvent is vitally important. A protein that is designed to fold in water will cease to fold correctly if immersed in a non-polar solvent.

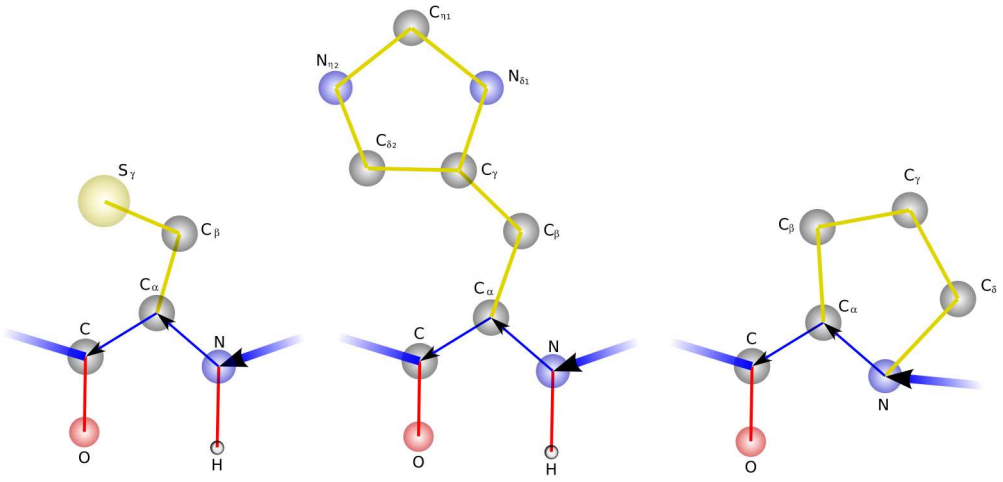

Figure 1.1: Carbon is shown as a gray sphere, nitrogen as a blue sphere, oxygen as red, sulfur as yellow, and hydrogen as white. Blue bonds represent the protein backbone, and yellow bonds are used in the amino acid's sidegroup. There are 20 types of natural amino acids. Shown here are cysteine, histidine, and proline. Cysteine is one of the two amino acids with sulfur in them. Histidine is one of the five amino acids with loops in their sidegroup. Proline is the only amino acid where the sidegroup bonds to the backbone. Glycine (not shown) has no sidegroup.

found a spot diffraction pattern for the first time, indicating a well defined shape for the protein. After 35 years of improving her experimental setup, in 1964 Hodgkin received the Nobel Prize in Chemistry for the structure of insulin.[3] Just before she received her award, in 1958 Max Perutz and Sir John Cowdery Kendrew determined the 3D structure of the first proteins, hemoglobin[38] and myoglobin[26], and for this they both won the Nobel Prize in Chemistry in 1962.[4] Today, there are several methods for determining

[3] "The Nobel Prize Internet Archive" , maintained by the Royal Swedish Academy of Sciences, says the prize was given, "for her determinations by x-ray techniques of the structures of important biochemical substances".

[4] "for their studies of the structures of globular proteins."

protein structure, and these techniques are reviewed in section 1.2.

1.1.1 Protein structure

All proteins begin as long strings of amino acids. Due to various inter-atomic driving forces (e.g. hydrophobic interactions with water, hydrogen bonds, ionic interactions, and van der Waals forces) the linear protein folds itself into a compact structure. Determining this 3D structure is often necessary to understanding the exact function (or functions) of a protein. As an example of the utility of knowing the structure of a protein, consider the development of the leukemia drug Imatinib.[9] Recently, Imatinib was developed to fight chronic myelogenous leukemia. This disease is caused by a DNA error that produces a protein called bcr-abl kinase. The protein speeds up cell division and inhibits DNA repair, making the cell more susceptible to further DNA mutations. Imatinib was designed, based on the structure of bcr-abl kinase, to bind to this particular protein and not significantly effect any other proteins. Once bound, bcr-abl kinase no longer functions, and the detrimental effects of the disease are inhibited. This is one of the first of a new kind of medicines, one designed to specifically target a harmful protein.

There are four distinct levels of protein structure, and it is believed a protein progresses through these four levels sequentially. Evidence of this temporal progression can be seen in many protein folding simulations (for example, see Liwo's 2006 work[44]). The first level of protein structure is called the primary structure. The primary structure is simply the linear chain of amino acids. This is the form the protein is assembled into by the ribosome. The gene for the protein is encoded in a segment of DNA, where it is transcribed to mRNA. The mRNA is read by the ribosome, which assembles the amino acids into a linear chain. The amino acid sequence for a protein is usually determined

by directly reading the DNA.

The secondary structure of a protein consists of local sub-structures, such as alpha helices and beta sheets. Alpha helices are shown in figure 1.2 and beta sheets are shown in figure 1.3. In 1951, using nothing except known information about bond lengths and angles, and a tremendous intuition, Linus Pauling predicted that protein sequences would form alpha helices. The alpha helices and beta sheets are held together by hydrogen bonds, and are fairly robust. Indeed, these structures are two ways the amino acid chain can saturate all the hydrogen bond donors and acceptors in the protein backbone. These structures only depend on properties of the protein backbone; the sidechains are irrelevant. For this reason, these structures are common to all proteins.

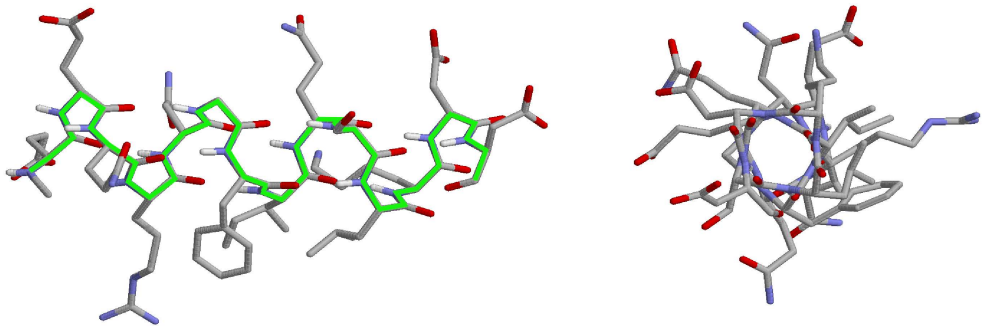

Figure 1.2: Shown here are two views of the same alpha helix. In the left view, the protein backbone is highlighted. The formation of a helix saturates most of the protein's hydrogen bond donors and acceptors, and it is for this reason that helices are very common to all proteins.

The tertiary structure is usually the main structure of the protein. This third level of structure is the assembly of the various secondary structures into a compact 3D geometry. The folding of the various secondary structures into the tertiary structure is driven largely by the hydrophilic/hydrophobic interactions of various sidegroups and water, and as a result a completely folded protein often has a hydrophobic, oily core. Additionally,

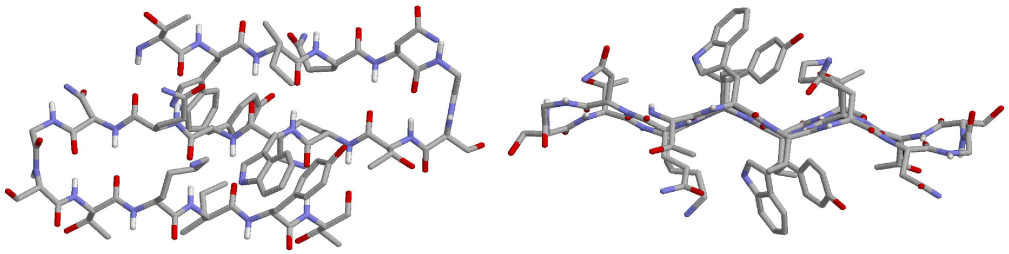

Figure 1.3: Shown here are two views of the same beta sheet. The left view shows how the hydrogen bonds line up, while the right view shows the profile of the structure; this shape is often called a corrugated sheet. Like the formation of an alpha helix, the formation of a beta sheet is another way the protein can saturate most of its hydrogen bond donors and acceptors.

further hydrogen bonding among the sidegroups, ionic interactions, or disulfide bonds help stabilize the protein. An important observation here is that while the formation of secondary structures has little to do with the constituent sidegroups, the assembly of the secondary structures into the tertiary structure is completely dependent on the exact sidegroups. Sometimes, if one amino acid in a protein is replaced by a different one, the formation of secondary structures will be essentially the same, but the assembly into the tertiary structure can be greatly affected; the protein may cease to fold into a functioning conformation.[5]

For some proteins, tertiary structures are further assembled into multi-protein complexes, called the quaternary structure. Typically the quaternary structure is stabilized by the same bonding forces that stabilized the tertiary structure: hydrophilic/hydrophobic interactions, hydrogen bonding among the sidegroups, ionic interactions, and disulfide bonds. A fine example of quaternary structure is hemoglobin. Hemoglobin is composed

[5]Though the mutation of one amino acid into another may possibly affect the final 3D structure, often it has little effect. The reason for this has much to do with evolution, and is described below in section 1.3.1.

of four proteins, two copies of hemoglobin α and two copies of hemoglobin β.[6] Each of these four proteins has a complicated tertiary structure (which in turn is composed of secondary structures, in this case mostly alpha helices), and these four proteins link together to form a functional protein complex, hemoglobin, as can be seen in figure 1.4.

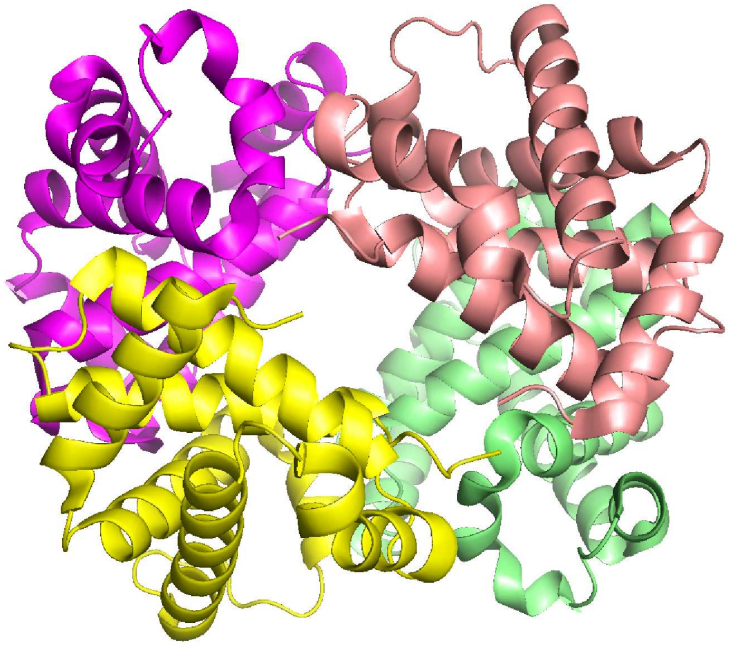

Figure 1.4: Only the backbone is displayed here; the sidegroups are not shown. The quaternary structure of hemoglobin consists of four tertiary structures: two copies of hemoglobin alpha (yellow and pink), and two copies of hemoglobin beta (purple and green). The tertiary structures are in turn comprised of alpha helices. The alpha helices are the only secondary structures of both hemoglobin alpha and hemoglobin beta. The primary structure of hemoglobin alpha consists of a linked chain of 141 amino acids, while hemoglobin beta is 146 amino acids long.

Many large proteins are composed of several nearly disjoint subregions. Though still part of the same threading backbone, these subunits are structurally self-stabilizing,

[6]hemoglobin α and hemoglobin β are two different, though similar, proteins. They should not be confused with alpha helices or beta sheets.

and are believed to fold independently from each other. These subregions are referred to as *domains*. Because they are small and fold independently, domains are typically the target of protein folding simulations. For example, the B domain of the staphylococcus aureus A protein is a small domain that protein folding simulations often focus on. The B domain of this protein is typically considered to be amino acid numbers 10 to 55 in the main staphylococcus aureus A protein, while the whole protein is several hundred amino acids long. The B domain folds independently of the rest of the protein, and forms, on its own, a compact and rigid structure.

Along the protein backbone, there are three dihedral angles, called ϕ, ψ, and ω, which are defined in figure 1.5.

The peptide bond ω exists in a resonance form and has a partial double bond nature, which constrains ω to either be in the cis or trans orientation. Furthermore, due to steric repulsion, the cis orientation is vastly unfavorable, and in proteins the trans orientation is preferred roughly 1000 to 1. For this reason, in the protein model of chapter 3, ω is assumed to be the trans configuration.

The dihedral angles ϕ and ψ are called *Ramachandran angles*, and are considered to be each amino acid's two basic internal degrees of freedom. These angles are ideally free to take any values, but due to steric repulsion there are some forbidden combinations of these angles. For example, if ϕ and ψ both equal 0, the amino acid's backbone oxygen will significantly overlap with the amino acid's backbone nitrogen. The distribution of naturally occurring Ramachandran angles is shown in figure 1.6. This plot is commonly called a *Ramachandran plot*, and large regions of it are not populated in natural proteins. Since glycine has no side group, its Ramachandran angles are less constrained, and glycine can access more of the Ramachandran plot than other amino acids. On the other hand, since the sidegroup of proline is bonded to the protein's backbone, proline has one

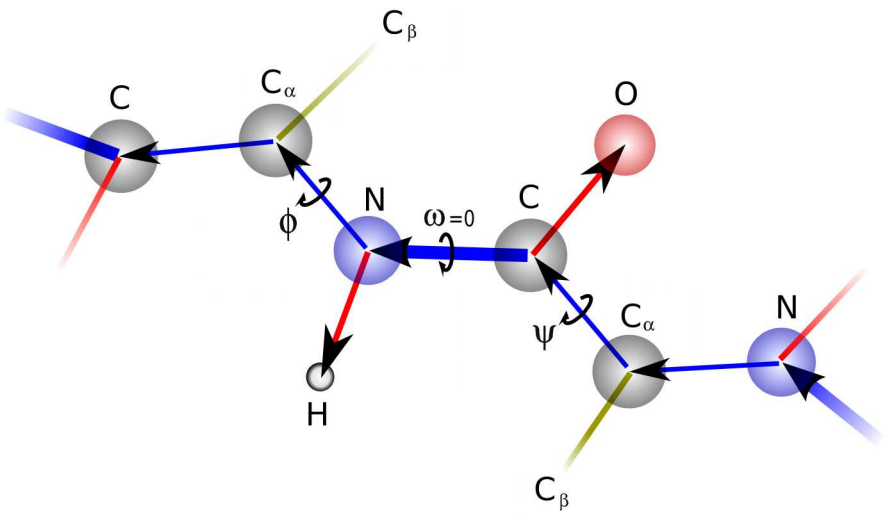

Figure 1.5: In this figure are two amino acids linked together. The blue bonds indicate the protein backbone. The backbone dihedral angles, ϕ, ψ, and ω, are defined as shown. Due to a double bond, ω is almost always in the orientation shown. Since $\omega = 0$, the six atoms C_α, C, N, C_α, O, and H all lie in a plane. The two dihedral angles ϕ and ψ are called *Ramachandran angles*, and ideally are free to assume any value. However, due to steric repulsion, some combinations of these angles are forbidden.

less degree of freedom than the other amino acids. For proline, ϕ is constrained to be $-75°$.

On a Ramachandran plot, two regions are notable since they frequently occur in nature. These two regions are indicated in figure 1.6. One is centered at $(\phi, \psi) = (-60°, -45°)$. This is the (ϕ, ψ) combination corresponding to alpha helices. The other region, corresponding to beta sheets, is around $(\phi, \psi) = (-120°, 115°)$.

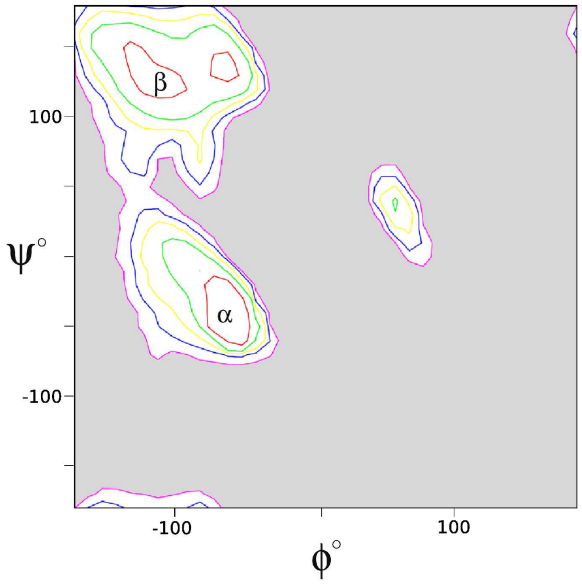

Figure 1.6: This figure shows the region of the Ramachandran plot populated by natural proteins. The data are taken from Kleywegt and Jones.[28] In less than 2% of all amino acids in the determined structures in the PDB, the Ramachandran pair lies in the gray region. 98% of all residues lie within the magenta contour, 95% within the blue contour, 90% within the yellow, 80% within the green, and 50% of all residues lie within the red contour. The regions on this plot corresponding to ideal alpha helices and beta sheets are indicated.

1.2 Methods of protein structure determination

There are two main methods of protein structure determination, x-ray diffraction and nuclear magnetic resonance spectroscopy (NMR). Currently, x-ray diffraction accounts for around 36,000 out of the 42,000 determined structures in the Protein Data Bank (PDB). NMR accounts for another approximately 6000 of the known structures, with the remaining structures being determined by less general techniques.[7]

[7]These numbers were obtained from the PDB website *http://www.rcsb.org/pdb/statistics/holdings.do* .

1.2.1 X-ray diffraction

X-ray crystallography (also called x-ray diffraction) is an old and extremely useful technique for determining protein structure. It uses the diffraction data produced by shining coherent x-rays through a crystalline protein sample to measure the magnitudes of the Fourier transform of the protein's electron density. The diffraction data obtained has significant encoded information regarding the crystalline structure of the protein. X-ray diffraction can be used to reconstruct the structure of many molecular structures, whether they are inorganic, DNA, or proteins. A large single crystal domain is ideal, but failing this, multiple small crystalline domains (e.g. powder diffraction) will also produce useful, though less complete data. It should be noted that it is the electrons in the molecules that interact with the x-rays, and not the atomic nuclei.

A single protein does not, by itself, have sufficient scattering power to collect useful diffraction data.[8] If one has an aqueous solution of proteins (consider 10^{23} proteins), all the proteins are in random orientations, and the scattered diffraction data is an incoherent mixture of all possible scattering orientations. This data is essentially useless. However, if the protein is crystallized, then there is a large number of proteins all in the same orientation, each constructively contributing diffraction data. In real space, the protein crystal can be considered a convolution of a periodic lattice, and the protein structure. Thus in Fourier space, the diffraction pattern will be a product of the Fourier transform of the crystal lattice, and the Fourier transform of the protein's electron density. Because of this, the diffraction Bragg peaks of the crystal lattice are modulated by the structure factor amplitude of the protein's Fourier transform. Thus by measuring the diffraction

[8]Technically, given an extremely weak x-ray source (so radiation damage to the molecule is irrelevant) and enough time (to gather a sufficient signal to noise ratio) a single protein *could* produce a useful diffraction pattern. Furthermore, since the Fourier data would be continuous, and not limited to Bragg peaks, phasing the Fourier data would be much easier.

amplitude at the Bragg peaks, the amplitudes of the the Fourier transform of the protein can be sampled on a regular grid.

The principle of crystalline diffraction is demonstrated in figure 1.7. In figure 1.7, the top left box is an example 2D electron density. The squared magnitude of its Fourier transform is displayed next to it. In the bottom left picture, the electron density has been convolved with a square lattice to simulate a “cubic” crystal of the electron density in 2D. The squared magnitude of the Fourier transform of the crystal is shown in the lower right. Notice that the Fourier amplitudes can only be sampled on a grid.

Once the amplitudes of the Fourier transform of the protein’s electron density are obtained, phases need to be assigned to the amplitudes. The phasing problem was once computationally challenging, but several methods have been developed to sometimes solve this problem.[13, 17, 36] In theory, any arbitrary set of phases will yield an electron density consistent with the measured Fourier amplitudes. However, large regions of the unit cell should be devoid of electron density, and the electron density should predominately lie in a small, compact region. Furthermore, the electron density needs to be positive everywhere. These additional constraints on the electron density are sometimes powerful enough to determine a unique set of phases. If a unique set of phases is determined, a simple inverse Fourier transform will yield the electron density of the protein, which leads to the protein’s structure. The existence of a unique set of phases consistent with the diffraction data is dependent on the availability of very high resolution diffraction data (measuring high frequency Fourier amplitudes), which in turn is dependent on a very high quality crystalline sample.

The first protein crystals were discovered by accident. It was observed that crystals formed spontaneously in the dried pools of blood on the decks of whaling boats.[25] These easily obtained crystals were actually sperm whale myoglobin, but very poor

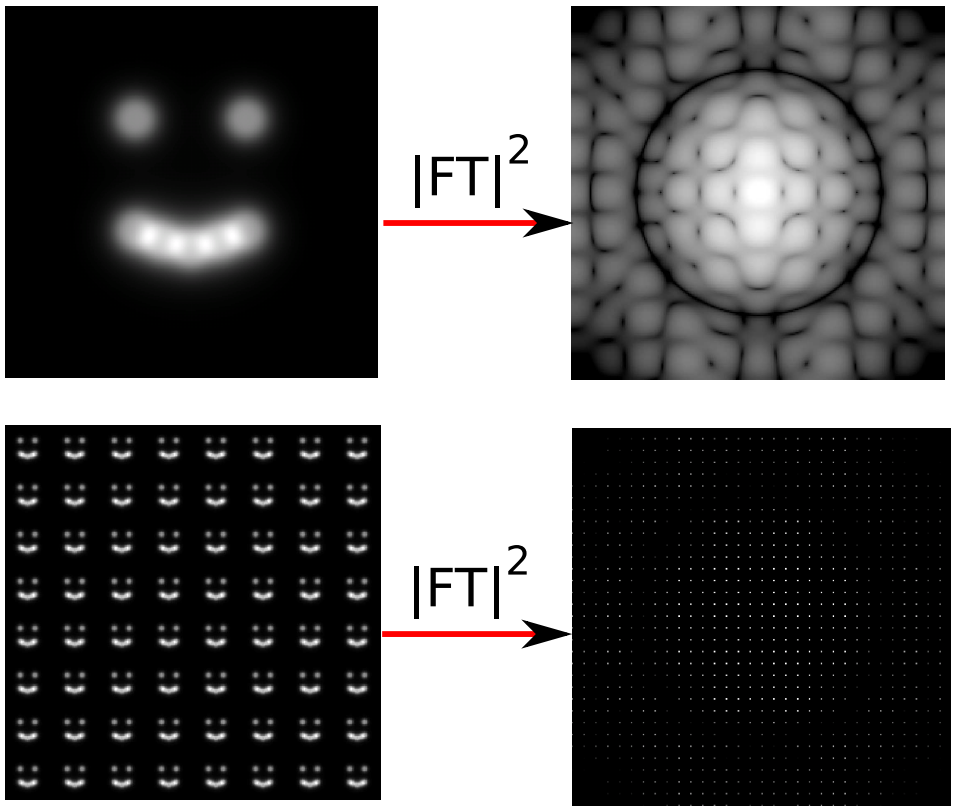

Figure 1.7: The upper left picture is an example 2D electron density. The magnitudes squared of its Fourier transform are displayed next to it. In the lower left, the same electron density has been formed into a square lattice. Next to the lattice are the magnitudes squared of its Fourier transform. Notice the Fourier transform of the crystallized electron density is just the Fourier transform of the lattice modulated by the Fourier transform of the electron density.

quality crystals. The quality of Max Perutz's early myoglobin diffraction data is too poor to solve for the structure of myoglobin. However, over the next 20 years Perutz worked with gradually higher quality crystals, until eventually the first protein structure (myoglobin) was determined in 1958 by Perutz and Kendrew. Since then, many proteins have been crystallized, and their structures determined via x-ray diffraction.

Today, if given a very high quality crystal, determining the structure of the pro-

tein has become a routine operation with x-ray diffraction. Unfortunately, it seems the vast majority of proteins do not easily crystallize, or fail to produce crystals of sufficient quality. For protein structure determination via x-ray diffraction, the problem has shifted from phasing the diffraction data, to the challenge of growing high quality protein crystals. To improve crystal quality, growth chambers with different geometries have been developed, allowing for thermal convection patterns that are conducive to growing high quality crystals. Alternately, one of the important applications of the International Space Station is to grow protein crystals in a zero gravity environment, where thermal convection will not be an issue at all.

1.2.2 Protein NMR

Nuclear magnetic resonance spectroscopy (NMR) also has a long history as an experimental technique, and a long history of Nobel prizes. The principle was first suggested by Rabi in 1938[42], who was investigating the nature of the strong force. For his discovery of the NMR method, he was awarded the Nobel Prize in Physics in 1944.[9] In 1946, Felix Bloch and Edward Mills Purcell further developed the method, for which they eared a Nobel prize in physics in 1952.[10] The specialized application to protein structure came much later. In 1968, Kurt Wüthrich at Bell Labs began studying protein structure using NMR, and for his remarkable success with this new technique for structure determination, he shared the Nobel Prize in Chemistry in 2002. [11]

[9] "For his resonance method for recording the magnetic properties of atomic nuclei." From "The Nobel Prize Internet Archive", maintained by the Royal Swedish Academy of Sciences.

[10] "for their development of new methods for nuclear magnetic precision measurements and discoveries in connection therewith."

[11] Kurt Wüthrich shared the prize with John B. Fenn and Koichi Tanaka for, "the development of methods for identification and structure analyses of biological macromolecules."

When a nucleus with a net nuclear spin is immersed in a strong magnetic field, the nucleus absorbs RF radiation at characteristic resonant frequencies. Furthermore, the absorbed frequencies shift slightly depending on the exact local environment of the nucleus, and the shift is proportional to the magnetic field strength. For example, a proton bonded to an oxygen will absorb slightly different frequencies than a proton bonded to a carbon. Hence, the absorption spectrum yields significant structural information.

Unfortunately, NMR requires a net nuclear spin for the nucleus to be detected. The majority of natural proteins are composed of naturally occurring isotopes of oxygen, carbon, hydrogen, and nitrogen. The most commonly occurring isotopes of oxygen (^{16}O) and carbon (^{12}C) have no net nuclear spin. While the most commonly occurring isotope of nitrogen (^{14}N) does have a net nuclear spin, it has a large quadrupole moment, which makes data collected from this isotope difficult. Because of this, for natural proteins only the hydrogens (lone protons) yield significant NMR data.[49]

The less common isotopes, ^{13}C and ^{15}N, are well suited for NMR data collection. To infuse a protein with these isotopes, first glucose is synthesized with ^{13}C, and ammonium chloride is synthesized with ^{15}N. Next, the DNA sequence encoding the protein is inserted into the DNA of *E. coli* (the most commonly used prokaryote), or yeast (the most commonly used eukaryote). The *E. coli* or yeast is then fed the modified glucose and ammonium chloride, and expresses the protein with the rare isotopes. The protein is finally harvested and purified. Unfortunately, many interesting proteins cannot be grown in sufficient quantities, or at all, by using yeast or *E. coli*. This is a problem not only for NMR, but for x-ray diffraction as well, since both methods require a large amount of very pure protein samples. For proteins that *E. coli* or yeast do not grow in sufficient quantities, the RIKEN Yokohama Research Institute is currently developing a cell-free protein synthesis method.

Data collection is typically a slow process, sometimes taking several days to obtain a usable signal to noise ratio.[49] The time required depends on the intensity of the magnetic field (hence there is tremendous incentive to develop stronger magnetic fields), and the concentration of the protein sample. Usually the first data set gathered is the 15N-HSQC data set (^{15}N heteronuclear single quantum correlation spectrum). This data set is relatively quick to collect, and is an effective test of the protein sample quality. The 15N-HSQC data set can be collected from natural proteins (with ^{14}N), but it either requires a much higher protein concentration, or a longer time to collect data. For this data set, one resonance peak is expected for every hydrogen bonded to a nitrogen. Because of this, a resonance peak is expected for every amino acid except proline (see figure 1.1), with a few additional signals from nitrogens in various sidegroups (e.g. histidine). Since protein sequencing is quick and easy to do, this NMR data set is a quick check to see if the protein sample is pure enough, or possibly in multiple folded states. Passing this check indicates whether or not it will be worthwhile to collect longer, more expensive data sets.

The various protein NMR data sets yield many different types of information about the folded protein. The most common types of information are distance restraints and angle restraints. For example, if two nuclei are sufficiently close to each other (between 1.8Å and 6Å), their individual resonance frequencies will be affected in relation to their proximity. Additionally, the exact local geometry around an amino acid's C_α affects the resonant frequencies of the C_α (if ^{13}C is used). Because of this effect, researchers can estimate the backbone torsional angles ϕ and ψ.

Once a large number of distance restraints and angle restraints have been measured, the data is used as input for the structure calculation process. There are many software programs available for this purpose, and converting the distance restraints and angle

restraints in a protein structure is considered essentially a solved problem.

Usually the multitude of proteins in the solution have strongly constrained parts that are all nearly identical, and weakly constrained parts that are highly variable. The NMR data yields a comparatively strong signal for the parts of the protein that are structurally identical in all the proteins, and very little restraint data for the parts of the protein that are weakly constrained. Because of this, reconstruction algorithms often accurately reproduce the strongly constrained parts of the proteins, while inaccurately describe the weakly constrained parts. An example of this can be seen in figure 1.8.

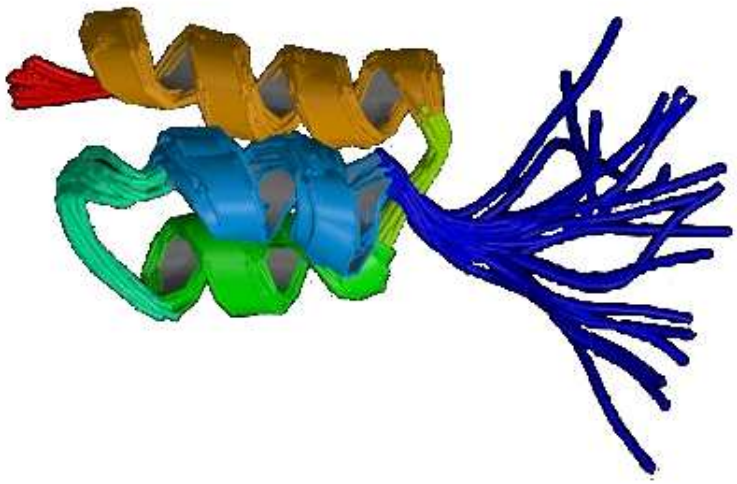

Figure 1.8: This protein (PDB code 1gab) was reconstructed 20 times based on the same NMR data. Some parts of the protein are strongly constrained, while the end of the protein (blue) is weakly constrained. For the weakly constrained part of the protein, NMR reconstruction algorithms produce many different protein conformations consistent with the NMR data. The data for this image is from the protein data bank.

One advantage of protein NMR over x-ray crystallography is that the protein is in

solution, rather than crystallized. For crystallized proteins, it is quite likely that the protein structure is slightly perturbed due to the fact that the protein is densely packed into a lattice. Crystallized proteins are quite rigid; they do not change shape significantly over time. On the other hand, since the NMR proteins are in a solution, they are free to take on the natural conformation they would assume in a living cell. However, since NMR experiments take a long time, the protein conformation data obtained from NMR is an average of the conformations sampled by the protein during this period. For example, hemoglobin comes in two conformations. The R state is for transporting oxygen, while the T state is for transporting CO_2. Both states have been resolved via x-ray diffraction. The structure of the R state is only slightly structurally different from that of the T state. Recently, however, the structure of hemoglobin was solved via NMR, and the reconstructed structure is an average of the R and T states.[33] The NMR reconstructed structure is misleading: it doesn't actually exist in nature. This illustrates a potential problem when interpreting NMR reconstructions.

1.3 Methods of protein structure prediction

Protein structure prediction is quite different from protein structure determination in that the predicted structure is not based on NMR or x-ray diffraction data; the only given data for protein structure prediction is the sequence of amino acids that constitute the protein. Instead of determining how a protein *does* fold based on quantitative measurements, the goal here is to predict how a protein *will* fold based on its sequence. There are two main branches of structure prediction methods: comparative protein modeling, which uses a large data bank of already determined protein structures, and *ab inito* energy minimization, which uses an interatomic potential whose global minimum represents the native fold. Since 1994, various methods of protein structure prediction have been

tested and compared in the semiannual CASP experiment [29].[12]

Currently, there are many more proteins with known sequences (as a result of the Human Genome Project) than with determined structures. X-ray crystallography and NMR spectroscopy are both slow and expensive processes, and so a reliable structure prediction method would be a tremendous asset to the protein structure community.

There are several significant reasons why protein structure prediction is a very difficult endeavor. One of the main problems is that for many proteins, the formation of the tertiary structure is dependent on the action of chaperon proteins. However, the effect of these chaperon proteins is not modeled in the structure prediction methods. Furthermore, many proteins can take on multiple conformations depending on their local environment. For these proteins, their structure depends strongly on the solvent, which is often not explicitly taken into account. Also, our understanding of the forces responsible for the stability of a folded protein is currently incomplete. And finally, there is Levinthal's paradox. In 1968, Cyrus Levinthal[31] calculated that if a protein has 150 amino acids (a relatively small protein) and each amino acid has two internal rotational degrees of freedom (ϕ and ψ), and each of these angles is able to take on three discrete values,[13] then there are 3^{300} different protein conformations for the small protein. If the protein sampled all of these conformations, even on a picosecond scale, the protein would never find the lowest energy state. Although nobody believes that proteins randomly sample all possible conformations, this simple calculation shows that there must be a funnel-like energy landscape for the protein to fold along.

Despite these hurdles, predicting the structure of small protein domains is currently

[12]The CASP organizers stress that the CASP experiment is not a competition, though despite this the experiment is sometimes referred to as the CASP competition.

[13]The arbitrary choice of three discrete values is a gross underestimation of the actual phase space the ϕ and ψ angles can explore.

quite achievable. There are several software packages that perform this calculation, and their various successes and failures can be seen in the latest CASP experiment.[29]

1.3.1 Comparative protein modeling

In addition to the sequence of amino acids, comparative protein modeling uses already determined structures in the protein data bank (PDB) to predict the structure of new protein sequences. Although the exact number of different protein sequences is enormous, it is estimated that there are actually a fairly small number of protein folds, perhaps on the order of 2000 largely distinct folds.[6] Thus, there are many similar sequences of amino acids that yield the same, or nearly identical, structures. In fact, approximately two thirds of all high resolution PDB structural domains can be assigned to about 1400 families of similar domain folds.[40] Furthermore, it has been calculated that for a given single-domain protein below 200 residues, there is more than a 90% chance that there is at least one solved structure (usually several) in the PDB that is structurally similar (RMSD[14] is less than 4Å).[50]

The two most successful comparative protein modeling techniques are homology modeling and protein threading. Each compares the sequence of amino acids in the undetermined structure against known structures in different ways. Homology modeling compares the undetermined sequence against sequences for which the native fold is known (which is usually determined via x-ray crystallography or NMR spectroscopy). If a similar sequence is found, the structural motifs from the known protein fold are assumed to be inherited in the undetermined sequence. Protein threading "threads" the sequence of the unknown protein into known protein structures and tries to find a protein

[14]RMSD stands for the root mean squared deviation. When comparing two different structures, the RMSD is a measure of how similar the structures are.

fold that is optimally compatible.

Note that all forms of comparative protein modeling rely heavily on the availability of similar template sequences whose structures have been determined. For large classes of proteins, such as membrane proteins, there is a dearth of available templates for comparison, since very few membrane proteins have had their structures determined. For such proteins, comparative protein modeling currently offers little promise. However, as more and more structures are determined via x-ray diffraction and NMR spectroscopy, comparative protein modeling will become a more powerful predictive technique.

Homology modeling

Homology modeling is based on the idea that as DNA mutates and evolves, the sequence of amino acids composing a protein can only change in slight and predictable ways. Thus, a sequence that encodes a particular protein can slowly evolve into a different sequence, creating essentially the same fold. Because of this, if an undetermined sequence of amino acids (called the “target”) is compared against sequences for which the protein structure is known, there is a chance that a similar sequence will be found (called the “template”), and there is a large probability that the target sequence will fold into a shape similar to that of the template.[35] It has been found that by changing less than half of the amino acids in a protein sequence, the protein can fold into a completely new conformation[7, 8]. However, such a drastic structural mutation is unlikely, since if the mutated protein ceases to function properly, there is a large chance the organism will not survive. Because of this evolutionary survival constraint, the functional aspects of a protein (and hence its 3D structure) must be maintained through the inevitable occurrence of sequence modifications.

If the target sequence has more than 70% of its amino acids in common with a tem-

```
Template Sequence   TAANK-PKEQ------QNAFYEILHLPNLNEEKKLNDALDA
                    ||-||--|||------||-||||----|-|||----||-||
Target Sequence     TADNKFNKEQSLKDDPQNKFYEII---NVNEE----DANDA
```

Figure 1.9: The target sequence is similar to the template sequence, but there is a large section present in the target sequence that is absent in the template sequence. Homology modeling is unreliable for predicting the structure of this inserted section.

plate protein structure, usually the target's fold can be predicted to within 2Å. If around 50% of the target protein's sequence is shared by a template protein structure, homology modeling is prone to error. Below 30% sequence similarity, structure prediction via homology modeling is very unreliable.[15] Furthermore, sometimes a target protein shares many of its amino acids with a template protein, except for a long inserted sequence somewhere in the middle of the target sequence. If this long inserted sequence is more than 10 amino acids long, it is likely to be an inserted protein domain present in the target protein, and absent in the template structure. Homology modeling is usually inaccurate for these inserted domains. An example of two similar sequences, along with substitutions and insertions, is shown in figure 1.9. In figure 1.9, the target sequence TADNKFNKEQSLKDDPQNKFYEIINVNEEDANDA is compared to the template sequence TAANKPKEQQNAFYEILHLPNLNEEKKLNDALDA. Some amino acid substitutions are evident, and there is a large section of the target sequence that is absent from the template sequence.

When the target sequence and template sequence are highly correlated (as in figure

[15]Below 30% sequence similarity is often referred to as the "twilight zone". In this regime, homology modeling can be wildly inaccurate, and sometimes perfectly correct.

1.9), homology modeling produces very reliable results. For this reason, recently there has been a structural genomics consortium created to try to find representative proteins for all families of protein structures.[48] It is widely believed that the main source of error in homology modeling lies in the critical initial step, finding a sufficiently similar template sequence in the PDB to a given target, and successfully aligning the target sequence against the optimal template.[50][23] A complete collection of every protein fold family, and many sequence representatives for each, will aid greatly in finding similar template sequences for any given target sequence.

The complete homology modeling method is basically three sequential steps. For each step, there are a wide variety of techniques available, and a large body of literature evaluating the various techniques.

First, a set of candidate template proteins is selected by comparing the target's sequence against templates stored in the protein data bank (PDB). While suitable templates are searched for, the target sequence is typically aligned against the potential templates.[16] Often there are several different template sequences in the PDB to which the target sequence can be aligned. Since evaluation of a structural model is more reliable than an evaluation of an alignment, a 3D model is typically generated for all promising alignments, and the best 3D model is selected in the final step, based on a suitable scoring function.

Second, for each alignment of template and target, a 3D protein model is constructed. There are several methods proposed for this construction.[4] For sections of the target's

[16]Sequence alignment is a very challenging problem, since there can be small insertions in either the target sequence or the template sequence. Consider for example the two sequences explored in figure 1.9. Discovering the shown alignment is quite challenging. There are many techniques and software packages available to construct an optimal alignment. Improper alignment is widely believed to be the main source of error in homology modeling.

sequence that are well matched by the template's sequence, this step is reliable. Typically these matched sections are well constrained core regions of the protein, or large secondary structures common to both the template and the target sequence.

Sections of the target sequence not well matched by the template (or not there at all) are commonly highly variable loops connecting secondary structures. These loop sections are often very important for the function of the protein, and no generally reliable method is available for constructing loops longer than five residues.[35] Even though homology modeling may often correctly predict the overall structure, the specific structure of these loop sections is far less reliable.

Finally, of all the predicted structures based on multiple promising sequence matches, one is selected based on an interatomic scoring function. For this function, statistical potentials based on the frequency of residue-residue contacts in known protein structures are often used, such as the statistical potential DOPE (Discrete Optimized Protein Energy).[46] Alternately, a physical chemistry based energy calculation can be used. A widely used energy force field, based on energy parameters used in CHARMM, is the EFF (Effective Force Field).[30] For a more detailed description of the various scoring functions, see the section on *ab initio* below.

Protein threading

Threading (also called remote homologue design) is a protein structure prediction technique that is similar to homology modeling in that the target sequence is compared against a database of known protein structures. The difference is that while homology modeling searches for sequence similarity between the target sequence and a template sequence, threading searches for a known protein fold already present in the protein database that is compatible with the target sequence. For example, if a target sequence

is assumed to have the fold of the staphylococcus aureus A protein (B domain (10-55)), then the validity of this assumption can be tested by various methods. With this assumed fold, if many hydrophilic residues are in the core of the protein, while many hydrophobic residues are on the exterior of the protein, then it can be safely concluded this is not the correct fold for the sequence. In this way, a small set of possible folds can be determined from the protein databank, and the most likely fold is determined by evaluating a suitable scoring function.

Evaluating the quality of a possible fold can be measured in many ways. An important measure, illustrated in the above example, is the location of hydrophobic/hydrophilic residues. Also, various amino acids are prone to be in a certain secondary structure. For example, alanine is frequently found in alpha helices, and proline is frequently found at the end of a helix. If a possible fold has amino acids in highly unlikely secondary structures, then the fold can be discarded. Furthermore, from the known structures in the PDB, statistics have been derived regarding the frequency of amino acids contacting each other. For example, isoleucine is often close to valine and rarely close to arginine. A possible fold yields a 2D matrix of inter-residue distances, and these distances can be used to determine the likelihood of the fold being correct.[34, 37] These various methods for determining the likelihood of a fold are used as a scoring function. The sequence is threaded through many structures to find one that maximizes the degree for which environment and adjacency preferences are satisfied.

In this way, protein threading has traits of both homology modeling and *ab inito* structure prediction. Like homology modeling, the target sequence is compared to already determined protein structures. But like *ab inito* structure prediction, the quality of a fold is evaluated based on an interatomic, or inter-residue energy.

1.3.2 *Ab Initio* energy minimization

Ab initio methods do not rely on the known structures in the PDB. Instead, these methods define an atomic interaction energy function and attempt to minimize the function. The idea was first put forth by Anfinsen in 1973.[1] For real proteins, the native fold is thermodynamically stable, and is thus a deep minimum of an energy landscape, perhaps (though not necessarily) the global minimum. Though the precise energy function that a folding protein explores is unknown, there have been many successful approximations to the energy function. Indeed, the energy functions of CHARMM and AMBER have already successfully predicted the native fold of many small proteins.[24]

Typically the protein is modeled as a complete collection of atoms[44, 47], or united atoms [22, 27] to aid computation efficiency. The formation of a hydrophobic core seems to be a key driving force in the assembly of the tertiary structure from the secondary structure. Because of this, many atomic simulations simulate the presence of water molecules surrounding the protein. As the studied proteins grow in size, the number of degrees of freedom of the simulation grow very quickly. Currently, to find the global minimum of the energy function for a moderately sized protein (300 amino acids), either new minimization algorithms or much faster computers are needed.

The most common energy minimization algorithms currently used are modifications to the general Monte Carlo technique. Two of these modifications are described below: energy landscape paving (ELP), and parallel tempering (PT). All Monte Carlo based algorithms suffer from three significant problems. First, to find the global minimum, the entire folding pathway needs to be modeled. Though this pathway is an interesting result in-and-of itself, if the goal is to find the global minimum, then modeling the folding pathway is a tremendously computationally costly endeavor. Second, the energy landscape is typically very rugged, with numerous deep local minima. Monte Carlo

based methods frequently get trapped in these minima, and significant computational resources are wasted while the Monte Carlo search attempts to free itself from the manifold minima. And finally, all Monte Carlo techniques only consider small modifications to the protein conformation. Every step is only a slight perturbation from the last. This considerably limits the rate at which Monte Carlo searches progress.

Recently, a new energy minimization algorithm, the difference map, has been applied to the problem of *ab initio* protein structure prediction. The algorithm has already demonstrated that it can find the global minimum of a rugged protein energy landscape for a very simple protein model[14], and the application of the algorithm to a realistic protein model is the subject of chapter 3.

The difference map overcomes the main barriers faced by Monte Carlo simulations. First, The difference map does not model the folding pathway. Immediately it begins searching for low energy protein conformations, with no regard for how the conformation folded. Second, the difference map was developed to avoid local minima in the energy landscape, and can efficiently navigate the rugged energy landscape until the global minimum is found. And finally, the difference map typically makes large global modifications to the protein, and because of these modifications quickly explores large regions of the energy landscape.

Many different energy functions are used. Some are statistical potentials, derived from known structures in the PDB. The statistical potentials are calculated based on the frequency of various types of atoms, or types of sidegroups, being in contact in the known structures. For example, if the sidegroup of alanine is never near the sidegroup of threonine, the statistical potential uses a repulsive “force” between these sidegroups. A commonly used statistical potential is DOPE (Discrete Optimized Protein Energy)[46], though several others have been developed.[11] Alternately, there are physical chemistry

based potentials. These simulate the interatomic interactions that come from van der Waals and electrostatic forces. To facilitate computation, often the interaction of water with the protein is modeled with an implicit solvation term that simulates the various hydrophobic-hydrophilic interactions of sidegroups with water. Both CHARMM and AMBER use physical chemistry based potentials[30], and with both software packages the user can choose to model water implicitly or explicitly.[17]

Parallel tempering

Parallel tempering is a modification to the basic Monte Carlo method that aims at facilitating the searching properties of a Monte Carlo energy minimization search. In the proceedings of the 23rd Symposium on the interface of computing science and statistics, Geyer seems to be the first to suggest the method in 1991. Since then, the use of PT has successfully minimized the energy of several small proteins.[10, 45]

If the Monte Carlo temperature is very low, the iterate[18] is trapped for many Monte Carlo iterations due to the low probability of the iterate overcoming the high energy barriers. On the other hand, if the temperature is too high the iterate can hop over barriers, but in this case there is a low probability the iterate will explore a deep energy minimum. Parallel tempering uses the conformational searching power of high temperature simulations and the energy minimizing power of low temperature simulations at the same time.

The method uses the standard Metropolis update to evolve the Monte Carlo iterate. First a small perturbation to the protein conformation is calculated, and then the new

[17]Explicitly modeling the water molecules always slows down the computer simulation tremendously. There are many different ways the effect of water can be implicitly simulated.

[18]Monte Carlo is an iterative algorithm. While the algorithm iterates a protein conformation, the current atomic configuration will be referred to as the *iterate* in this work.

configuration's energy is calculated. The step is either accepted or rejected, based on the probability of acceptance,

$$P = \mathrm{e}^{-\frac{E_{new}-E_{old}}{k_b T}} .$$

If P is greater than 1 (this implies $E_{new} < E_{old}$), the step is automatically accepted. If P is between 0 and 1, then P is the probability of accepting the step. Typically a step acceptance rate of 50% is ideal, and the magnitude of the random perturbation is adjusted to maintain this acceptance rate.

If two different Monte Carlo simulations (starting from the same initial condition) are run with slightly different temperatures, the simulations will inevitably create similar energy histograms[19], which implies that there will be moments when both iterates have the same energy. Since each simulation has no memory, when the two simulations have the same energy, swapping the atomic configurations of the two simulations will not interfere with the detailed balance condition of each Monte Carlo simulation. Furthermore, if the two simulations are not at the exact same energy, a swap can still satisfy each detailed balance condition if the reverse swap is equally likely. The probability of a swap between two Monte Carlo simulations is usually,

$$\text{Prob} = \min\left(1, \mathrm{e}^{(\frac{1}{k_b T_i}-\frac{1}{k_b T_j})(E_i-E_j))}\right) .$$

Note that if a hotter simulation has a lower energy than a colder simulation (for example, $E_i > E_j$ and $T_i < T_j$) then a swap is guaranteed.

Usually, far more than two simulations are computed simultaneously. Previous studies[45] have used as many as 30 "clones" concurrently evolving via Monte Carlo steps. Every clone begins with the same initial protein conformation, and with every Monte

[19]The energy histogram produced by a Monte Carlo simulation is both a function of the energy landscape and the Monte Carlo temperature. The slight different temperatures will produce similar histograms.

Carlo step the probability of a swap is calculated among every pair of clones. The temperatures used must span a range such that the coldest clone will greedily roll down the energy landscape, and the hottest clone can traverse the highest barriers. For protein simulations, this implies the temperatures must fall in a range between approximately 2K and 600K. The purpose of the hottest clones is to scale the energy barriers that may trap the coldest clones, while the purpose of the coldest clones is to languish in the local minima, effectively preserving the lowest energy conformation yet found. In this way, all of the advantages of a Monte Carlo simulation are accentuated, and the weaknesses are minimized.

PT has been used to find the global minimum of several tiny proteins, but it is computationally demanding. To simulate the folding dynamics of a sequence consisting of forty amino acids (a tiny protein), with 20 clones, on a single computer, would require about 13 seconds per iteration. To completely fold the protein, it would take roughly 60 3GHz computers two weeks to complete.

Energy landscape paving

Energy landscape paving (ELP) is another variant of the standard Monte Carlo algorithm. In this variation, the energy landscape is locally flattened to make local minima no longer confining. This is done by adding a "penalty" to the energy function that is derived from the histogram of any suitably chosen order parameter.[3] A common choice of this order parameter is the protein conformation's energy. The histogram of the Monte Carlo's search energy is updated at every Monte Carlo step, and is used to create a new energy function,

$$\tilde{E}_t = E + f(H(E_t)) \, .$$

Here, E is the original energy function, $\tilde{E}_t$ is the effective energy function (the subscript t indicates this is a function of "time", or iteration number), and $f(H(E_t))$ is a function of the energy histogram (the histogram is also a function of "time"). If the Monte Carlo iterate languishes in a local minimum for an extended period of time, the energy landscape is effectively modified to make this local minimum slowly increase in energy. Eventually the iterate escapes the local minimum and continues searching the energy landscape. In this way, the energy landscape evolves over time, and this method therefore violates detailed balance. Because of this, ELP is inappropriate for uncovering statistical thermodynamic properties of the protein. However, the technique's main purpose is to discover the global minima of the energy landscape, not to describe the protein's conformational statistical distribution.

The technique has had some success in protein structure prediction for simple protein models[2] and small proteins[21], but for all of its successes it remains a slow algorithm. The algorithm does nothing to avoid the trapping of deep local minima, and it requires costly computation time to deform the energy landscape to make the trapping minima escapable. Even for the smallest independently folding protein,[20] it took the algorithm around 25,000 Monte Carlo iterations to find the global minimum in the energy landscape.[21]

The difference map

The difference map algorithm is significantly different from Monte Carlo methods. The main difference, and the reason for its efficacy, is in the embedding search space. In all Monte Carlo searches, the atoms of the folding protein are continually bound together to form a protein molecule. Monte Carlo based algorithms only make slight modifications

[20]The vilin headpiece, PDB code HP-36, is only 36 amino acids long.

to this molecule to try to locate the global minimum of the energy landscape. The difference map frees the atoms from their bonds and allows them to move independently. It searches for an atomic configuration that is both bonded correctly (as dictated by the primary sequence), and has a low non-bonded energy. The search space is indeed much larger, but this is not necessarily a bad thing. The manifold minima of the energy landscape that normally trap Monte Carlo simulations are now easily escapable due to the higher dimensionality of the search space used by the difference map.

Knots provide a good example of how increasing the search dimension can make the normally formidable energy landscape less daunting. In three dimensions, there are an infinite number of inequivalent non-trivial knots. Each of these cannot be undone because the rope is constrained by not being able to pass through itself. However, in four dimensions, there is only one knot, the trivial knot. In four dimensions, any knot can be undone because the steric barrier of passing rope through itself can be circumvented. To untangle a knot in four dimensions, the rope does not need to pass *through* itself, the high dimensionality allows the rope to go *around* itself. Similarly for protein folding, energy landscape minima that would normally trap a Monte Carlo search can be easily escaped by the difference map. These local minima in the energy landscape are only confining if the protein molecule is restricted to a rigid bond geometry. Free of this restriction, the atomic configuration can easily tunnel out of (or through) the energy landscape minima.[21]

As described in Chapter 2, the difference map seeks an element that is simultaneously in two constraint sets. In the context of protein folding, the "element" is an atomic configuration, and the constraint sets are defined by: (1) the atomic configuration must

[21] Note that this analogy is not perfect. In the knot example, the higher dimensionality trivializes the problem of untangling the knot. In the context of protein folding, the higher dimensionality makes the search easier, but by no means trivial.

have the correct bond geometry present in the native protein; and (2) the atomic configuration must to have a non-bonded energy below a predefined target energy. With these constraints the difference map has already been very successful in finding global minima of a simple protein model.[14] A detailed explanation of the two constraint spaces, as applied to realistic proteins, can be found in appendix A.

In chapter 3, an application of the difference map to a realistic protein model is thoroughly explored. The results indicate that this relatively new algorithm finds low energy protein conformations far faster than a popular Monte Carlo variant, parallel tempering. The significant gains in computational efficiency are due, in part, to the omission of simulating the folding pathway. While parallel tempering gradually minimizes the energy by constructing secondary structures, and eventually tertiary structure from these, the difference map searches immediately for low energy protein conformations, and finds them at an unprecedented rate.

Chapter 2

The difference map

2.1 Definitions

The difference map (DM), comes from a long line of precursor algorithms. These original algorithms (described in section 2.2.1 below) were explained in their original papers in a manner that was only later realized to be fundamentally geometric in nature. Because of this, they will not be described here with their original form, but rather in the more intuitive language of constraint sets and projections.

2.1.1 Constraint sets

A *constraint set* is a subset of a Euclidean space, whose elements satisfy a certain property. Some simple examples include:

Positivity Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $x_i > 0\, \forall\, i$
Magnitude Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $\vec{x} \cdot \vec{x} < M$
Support Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $i \in S$ implies $x_i = 0$
Integer Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $x_i \in \mathbf{Z}\, \forall\, i$
Binary Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $x_i \in \{0, 1\}\, \forall\, i$
Fourier Constraint Set	:	$\forall\, \vec{x} \in \mathbf{R}^n$ s.t. $\parallel FT(\vec{x})_i \parallel = M_i\, \forall\, i$

The *magnitude constraint* is dependent on a given parameter, M. The *support constraint* is dependent on a set, S. The *binary constraint* sometimes constrains the vector values to be either -1 or 1, rather than 0 or 1 as written above. The *Fourier constraint* is dependent on a set of given Fourier magnitudes.

The first three constraints here define convex constraint sets. For a convex constraint

set, if $\vec{x}_1$ and $\vec{x}_2$ both satisfy the constraint, then so does $a\vec{x}_1 + (1-a)\vec{x}_2$ for all $a \in [0,1]$. The last three constraint sets are non-convex since they fail to satisfy this property.

The DM, and its predecessor algorithms, were designed to locate elements that simultaneously exist in two constraint sets. For example, it is easy to find a vector that is simultaneously in the positivity constraint set and the integer constraint set (e.g. $\{1,2,0,10,2\}$). However, it is typically very difficult to find a simultaneous member of the binary and Fourier constraint sets, as will be demonstrated below.

The Fourier constraint deserves some special attention, since satisfying it (subject to a variable second constraint) has been the main driving force for the development of the DM. To illustrate the topology of the Fourier constraint set, consider the following:

A vector $\vec{x} \in \mathbf{R}^n$ is Fourier transformed,[1] and the magnitudes of the Fourier coefficients are made known. Any arbitrary set of phases[2] can be assigned to these magnitudes, and the inverse Fourier transform will produces a vector $\vec{x} \in R^n$ satisfying the Fourier constraint. It is the vectors produced by an assigned set of phases which constitute the Fourier constraint set.

Given an arbitrary assignment of phases[2], let the i^{th} Fourier coefficient be $M_i \, \mathrm{e}^{i\,\phi_i}$. Call the inverse Fourier transform of this assignment of phases $\vec{x}$. Then the j^{th} component of the vector $\vec{x}$ (note $\vec{x}$ has n components) is given by,

$$x_j = \frac{1}{\sqrt{n}} \Big(M_0 + M_1 \, \mathrm{e}^{i\left(2\pi \frac{j}{n} + \phi_1\right)} + M_2 \, \mathrm{e}^{i\left(2\pi \frac{2j}{n} + \phi_2\right)} + \ldots \\ \ldots + M_{n-2} \, \mathrm{e}^{i\left(2\pi \frac{(n-2)j}{n} + \phi_{n-2}\right)} + M_{n-1} \, \mathrm{e}^{i\left(2\pi \frac{(n-1)j}{n} + \phi_{n-1}\right)} \Big) \; .$$

[1] All Fourier transforms mentioned in this work will be assumed to be discrete Fourier transforms.

[2] Since the vector that produced the Fourier magnitudes is real, the Fourier coefficients have an inherent symmetry. The i^{th} coefficient is the complex conjugate of the $(n-i)^{th}$ coefficient, and the zero frequency coefficient is automatically real. Hence, an "arbitrary set of phases" is understood here to mean an arbitrary set of phases respecting this symmetry.

Since the vector $\vec{x}$ is real, the Fourier components must come in conjugate pairs.[2] Using this, the j^{th} component of the vector $\vec{x}$ can be rewritten as,

$$\begin{aligned}
x_j &= \frac{1}{\sqrt{n}}\Big(M_0 + M_1\, \mathrm{e}^{i\left(2\pi\frac{j}{n}+\phi_1\right)} + M_2\, \mathrm{e}^{i\left(2\pi\frac{2j}{n}+\phi_2\right)} + \ldots \\
&\qquad \ldots + M_2\, \mathrm{e}^{i\left(2\pi\frac{(n-2)j}{n}-\phi_2\right)} + M_1\, \mathrm{e}^{i\left(2\pi\frac{(n-1)j}{n}-\phi_1\right)}\Big) \\
&= \frac{1}{\sqrt{n}}\Big(M_0 + M_1\, \mathrm{e}^{i\left(2\pi\frac{j}{n}+\phi_1\right)} + M_2\, \mathrm{e}^{i\left(2\pi\frac{2j}{n}+\phi_2\right)} + \ldots \\
&\qquad \ldots + M_2\, \mathrm{e}^{i\left(2\pi\frac{-2j}{n}-\phi_2\right)} + M_1\, \mathrm{e}^{i\left(2\pi\frac{-j}{n}-\phi_1\right)}\Big) \\
&= \frac{1}{\sqrt{n}}\Big(M_0 + M_1\, \left(\mathrm{e}^{i\left(2\pi\frac{j}{n}+\phi_1\right)} + \mathrm{e}^{i\left(2\pi\frac{-j}{n}-\phi_1\right)}\right) + \\
&\qquad + M_2\, \left(\mathrm{e}^{i\left(2\pi\frac{2j}{n}+\phi_2\right)} + \mathrm{e}^{i\left(2\pi\frac{-2j}{n}-\phi_2\right)}\right) + \ldots\Big) \\
&= \frac{1}{\sqrt{n}}\left(M_0 + 2\, M_1\, \cos\left(2\pi\frac{j}{n}+\phi_1\right) + 2\, M_2\, \cos\left(2\pi\frac{2j}{n}+\phi_2\right) + \ldots\right) .
\end{aligned}$$

Call the vectors whose j^{th} element is given by $\cos\left(2\pi\frac{k\,j}{n}\right)$ and $\sin\left(2\pi\frac{k\,j}{n}\right)$, $\vec{C}_k$ and $\vec{S}_k$ respectively. Explicitly,

$$\vec{C}_k = \begin{bmatrix} \cos\left(2\pi\frac{k\,0}{n}\right) \\ \cos\left(2\pi\frac{k\,1}{n}\right) \\ \cos\left(2\pi\frac{k\,2}{n}\right) \\ \ldots \\ \cos\left(2\pi\frac{k\,(n-1)}{n}\right) \end{bmatrix} \quad \text{and} \quad \vec{S}_k = \begin{bmatrix} \sin\left(2\pi\frac{k\,0}{n}\right) \\ \sin\left(2\pi\frac{k\,1}{n}\right) \\ \sin\left(2\pi\frac{k\,2}{n}\right) \\ \ldots \\ \sin\left(2\pi\frac{k\,(n-1)}{n}\right) \end{bmatrix}$$

Notice that $\vec{C}_k \cdot \vec{C}_m = 0$ and $\vec{S}_k \cdot \vec{S}_m = 0$ unless $k = m$, and $\vec{C}_k \cdot \vec{S}_m = 0$ for all k and m. With this definition, the vector $\vec{x}$ can be written as,

$$\begin{aligned}
\vec{x} &= \frac{1}{\sqrt{n}}\,\Big(M_0\,\vec{C}_0 + 2\,M_1\,\Big(\cos(\phi_1)\,\vec{C}_1 - \sin(\phi_1)\,\vec{S}_1\Big) + \\
&\qquad +2\,M_2\,\Big(\cos(\phi_2)\,\vec{C}_2 - \sin(\phi_2)\,\vec{S}_2\Big) + \ldots\Big)
\end{aligned}$$

From this it can be seen that the Fourier constraint set is the Cartesian product of $\frac{n-1}{2}$ orthogonal circles.[3] The circles have radii given by the Fourier magnitudes, and are

[3] If n is even, $\frac{n}{2} - 1$ circles. The derivation of this result is nearly identical.

parameterized by the assigned phases. Thus, the Fourier constraint defines a hypertorus in real space, and it will be referred to in this work as the Fourier constraint hypertorus.

2.1.2 Projections

In the context of the above constraint sets, a *projection* is a map $\mathbf{R}^n \rightarrow \mathbf{R}^n$, that minimally modifies a given vector to produce an element of a constraint set. Specifically, a projection of the vector $\vec{x}$ to constraint set C is denoted

$$P_C[\vec{x}] = \vec{x}_c \, ,$$

where the vector $\vec{x}_c$ is a closest element in the constraint set C to the vector $\vec{x}$. Obviously, the result of a repeated projection is unchanged, or $P_C[P_C[\vec{x}]] = P_C[\vec{x}] = \vec{x}_c$. A graphical example of projections can be seen in figure 2.1. Here, the red and the blue contours are two 2D constraint sets, two example points (black) are shown projected onto the constraint sets. For the green dots, there is no "closest" point on the red contour. Similarly, for the orange dot there is no "closest" point on the blue contour. Points such as these, where there is no unique projection, almost always comprise a set of measure zero. For this reason, these points do not effect the search dynamics of any of the algorithms below.

Both the projection to the binary constraint and the projection to the Fourier constraint will be used in the sections below, and so these projections will be explained in detail here.

Projections to the binary constraint are easy to compute. Let the projection of $\vec{x}$ to the binary constraint be called $\vec{b}$. To compute the projection, we thus seek to find a vector $\vec{b}$ in the binary constraint, closest to the vector $\vec{x}$. Thus, we seek to minimize,

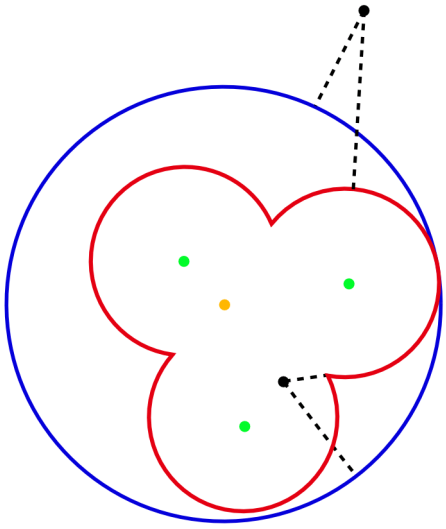

Figure 2.1: Here, the red and blue contours represent two different constraint sets. Two black dots are shown projected onto these constraints. Note the projection of a point is defined as the closest point in the constraint set. For the green dots, there is not a unique "closest" point in the red contour, and for the orange dot there is no "closest" point in the blue contour. For these special points, there is not a unique projected point defined.

$D = \| \vec{x} - \vec{b} \|$. Alternately, we can minimize D^2, and,

$$D^2 = \sum_i (\vec{x}_i - \vec{b}_i)^2 \,,$$

which is minimized only when each of $\| \vec{x}_i - \vec{b}_i \|$ is minimized independently. Thus,

$$P_B[\vec{x}] = \vec{b} \quad \text{where} \quad b_i = \begin{cases} 1 & : \quad x_i > 0 \\ -1 & : \quad x_i < 0 \end{cases}$$

Essentially, this produces a corner of a n-dimensional hypercube, closest to the vector $\vec{x}$. For this reason the binary constraint set is sometimes referred to as the binary constraint hypercube.

The Fourier constraint projection is tougher to conceptualize. Given the vector $\vec{x}$, we seek a vector $\vec{f}$ that is a member of the Fourier constraint, and also the closest to the

vector $\vec{x}$. Let $\vec{d} = \vec{f} - \vec{x}$. Hence, we seek to minimize $\vec{d}$ subject to the condition that $\vec{x} + \vec{d}$ is a member of the Fourier constraint set. Since the discrete Fourier transform is a unitary operator, we can choose to minimize the magnitude of $\vec{d}$ in Fourier space, where it is more convenient. In Fourier space, we seek to minimize $\tilde{d}$ subject to the condition that each component of $\tilde{x} + \tilde{d}$ has the correct Fourier magnitude.[4] Notice that in Fourier space, the components of $\tilde{x} + \tilde{d} = \tilde{f}$ can be dealt with independently. The closest $\tilde{f}_i$ to $\tilde{x}_i$ that has the correct Fourier magnitude is just a rescaling of the magnitude of $\tilde{x}_i$ to M_i, while keeping the phase of $\tilde{x}_i$. In other words, the projection of $\vec{x}$ to the Fourier magnitude constraint is done by,

$$P_F[\vec{x}] = \vec{f} \quad \text{where} \quad \tilde{f}_i = \begin{cases} \frac{M_i}{\|\tilde{x}_i\|}\,\tilde{x}_i & : \quad \| \tilde{x}_i \| > 0 \\ 0 & : \quad \| \tilde{x}_i \| = 0 \end{cases}$$

This is shown graphically with the blue contour of figure 2.1 (the red contour has no relevance to the current topic). In that figure, the blue contour can be considered as the set of complex numbers a distance M_i from the origin. The black points represent two possible $\tilde{x}_i$'s. Notice the closest point on the blue contour to $\tilde{x}_i$ has the same phase as $\tilde{x}_i$. The projection simply amounts to a rescaling of the magnitude of $\tilde{x}_i$ to M_i. Since this projection is distance minimizing for each component in Fourier space, it is then also distance minimizing in real space as well.

2.2 The difference map

Like its predecessors, the DM algorithm was initially developed for the purpose of phase retrieval. Phase retrieval problems arise in a variety of fields and applications, such as speckle interferometry in astronomy,[19] x-ray diffraction, or recovering the phases of

[4] Here, the symbol $\tilde{x}$ means the Fourier transform of $\vec{x}$, $\tilde{d}$ means the Fourier transform of $\vec{d}$, etc.

a quantum mechanical wavefunction from measured probability distributions.[41]

Typically one measures Fourier intensities ($\parallel F(k) \parallel$) related to an image ($f(x)$) by,

$$F(k) = \sum_{x=0}^{N-1} f(x) \mathrm{e}^{-i2\pi k \frac{x}{N}} ,$$

where the image $f(x)$ is subject to various constraints (for example, positivity, reality, or atomicity).

A relevant example of this phase problem is protein x-ray diffraction. When coherent x-rays are diffracted through a protein crystal, the magnitudes of the Fourier transform of the protein's electron density are measured [5]. Assuming a large amount of diffraction data is measured, the electron density can be modeled as being tightly localized around the atoms. This implies the function $f(x)$ is nonzero only at a small set of discrete points (and always positive). Given enough diffraction data, this constraint on $f(x)$ makes the retrieval of the phases of $F(k)$ computationally feasible by many modern algorithms.

A simplified model of the x-ray diffraction problem can be constructed in the following way. Consider a 1D "crystal" of point like atomic scatterers. The electron density would look like, for example,

$$\ldots 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, \ldots$$

where the "unit cell" $\{0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0\}$ is infinitely repeated. Here, a 1 represents an atom. In this example, there is a rather coarse sampling of the unit cell into "atom" and "no atom"; there are only eleven bins to describe 5 atoms. In this model, the analog of collecting x-ray diffraction data is performing a discrete Fourier transform of the unit cell, and interpreting the Fourier magnitudes as the diffraction data. In this

[5] This is explained in more depth in chapter 1.2.1.

case, it would be,

$$\{1.50756, 0.611664, 0.572813, 0.150811, 0.306426, 0.73807,$$
$$0.73807, 0.306426, 0.150811, 0.572813, 0.611664\} .$$

A more precise representation of the same unit cell is,

$$\{0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0\} \quad .$$

Note this is the same unit cell, just represented with a finer sampling. And in this case, the collected Fourier magnitude data would be,

$$\{0.870388, 0.317104, 0.451094, 0.255148, 0.189331, 0.367474, 0.161381, 0.500648,$$
$$0.323209, 0.203428, 0.189912, 0.348155, 0.349246, 0.768841, 0.144825, 0.447264,$$
$$0.259345, 0.259345, 0.447264, 0.144825, 0.768841, 0.349246, 0.348155, 0.189912,$$
$$0.203428, 0.323209, 0.500648, 0.161381, 0.367474, 0.189331, 0.255148, 0.451094,$$
$$0.317104\} .$$

Notice that in this example, collecting more Fourier magnitudes in Fourier space corresponds to a more precise sampling of the unit cell in real space. For protein x-ray diffraction, if enough Fourier magnitudes are measured, the electron density can be reconstructed on a fine grid, and the density can be therefore modeled as distinct, point like scatterers. This constraint on the electron density in real space (along with positivity) is typically a strong enough constraint to determine a unique set of phases in Fourier space.[6]

To demonstrate the effectiveness of the following algorithms when applied to x-ray diffraction phase retrieval, they will be tested on this simple 1D model of a protein crystal. For all of the tests, the correct atomic distribution will be the binary vector,

$$\{0,0,1,0,0,1,0,0,0,1,0,1,1,0,0,0,0,1,1\} ,$$

[6]This is explained in more depth in chapter 1.2.1.

with corresponding Fourier magnitudes,

$$\{1.60591, 0.0891748, 0.425296, 0.676124, 0.291878, 0.459069, 0.951737, 0.148424,$$
$$0.449914, 0.371969, 0.371969, 0.449914, 0.148424, 0.951737, 0.459069, 0.291878,$$
$$0.676124, 0.425296, 0.0891748\} .$$

These magnitudes will be interpreted as the x-ray diffraction "data", and the following algorithms will try to reconstruct the corresponding "electron density".

2.2.1 Precursor algorithms

In their original published form, both the error reduction algorithm and the hybrid input-output map did not use the convenient language of constraint sets and projections. Here they will be explained in these terms. Their effectiveness at 1D phase retrieval will also be demonstrated.

The error reduction algorithm

The error reduction algorithm (ER) is a slight generalization of the Gerchberg-Saxton algorithm.[20] The Gerchberg-Saxton algorithm was developed to solve problems where one has Fourier magnitude data for both $F(k)$ and $f(x)$ (both are complex), and $F(k)$ is related to $f(x)$ by,

$$F(k) = \sum_{x=0}^{N-1} f(x) \mathrm{e}^{-i2\pi k \frac{x}{N}} .$$

This situation arises in the fields of electron microscopy and in wavefront sensing, where typically two intensity measurements are taken, one for $F(k)$ and $f(x)$ each. The generalization of the Gerchberg-Saxton algorithm to problems where the intensity of $f(x)$ is no longer measured, but instead assumed to be, for example, real or positive, is called the error reduction algorithm.[18]

Both the Gerchberg-Saxton algorithm and ER are iterative algorithms, where the iterate is alternately projected onto two constraint sets. For this reason, the algorithm is sometimes referred to as the alternating projections algorithm.[7] For the Gerchberg-Saxton algorithm, these constraint sets are both Fourier constraint sets. For ER, one constraint is typically a Fourier constraint, while the other constraint is variable (such as positivity, or a support). For the 1D crystal model, the second constraint set is a binary constraint in real space.

For the application of ER to the 1D crystal model, the iterate is evolved by,

$$\vec{x}_{n+1} = P_F[P_B[\vec{x}_n]] \ .$$

Note the final projection is to the Fourier constraint. Hence, after every iteration, the iterate is real valued, though not necessarily binary, and its Fourier transform has the correct magnitudes.

ER is a promising algorithm because it can be proven that with every iteration, the iterate moves less than the previous move, and thus the algorithm converges very quickly. Unfortunately, though the algorithm converges very quickly to a fixed point, usually the fixed point does not correspond to an intersection of the constraint sets. If the iterate is sufficiently close to an intersection of the constraint sets, the algorithm converges to the solution in just a few iterations. On the other hand, if the iterate is not close to the intersection, this algorithm is quite prone to stagnation.[8]

The reason for the stagnation problem of ER is shown in figure 2.2. In this figure, a 2D planar cross section of the Fourier constraint hypertorus (blue contour) is shown along with four corners of the binary constraint hypercube (red dots). Clearly the top

[7]The alternating projections algorithm is demonstrated in the context of protein structure prediction in chapter 3.2.2.

[8]Stagnation occurs when the iterate finds a fixed point, but the fixed point has no relation to the solution.

fixed point of the ER algorithm is not an intersection of the constraint sets, yet further iteration will not free the iterate from this fixed point. However, the bottom fixed point in the figure is where the constraint sets actually intersect, and ER finds this point in just one iteration.

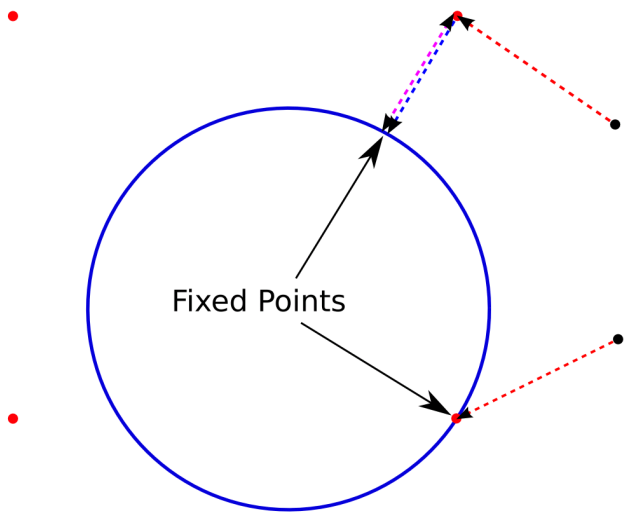

Figure 2.2: In this figure, the Fourier constraint hypertorus is represented by a 2D circle (blue), and the binary constraint hypercube is represented by the four red dots. Projections to the binary constraint are shown with the red dotted lines, and projections to the Fourier constraint is shown with a blue dotted line. The black dots represent initial iterate to the ER algorithm. The top black dot finds a fixed point of the algorithm after one iteration of $\vec{x}_{n+1} = P_F[P_B[\vec{x}_n]]$. Subsequent iteration of this fixed point does not change it. The bottom black dot finds the set intersection after one iteration.

The effectiveness of the ER algorithm, and the severity of its stagnation problem, is demonstrated by the following test. A vector of 19 real numbers, chosen randomly from a uniform distribution from [0,1], is evolved via ER. The number of iterations before a fixed point is found is recorded. This test is repeated 10,000 times.

The average number of iterations before the iterate finds a fixed point is found to be 3.09. Of these 10,000 trials, the solution was found 3.12% of the time.

Alternately, the projections can be conducted in the opposite order. For the ER algorithm,

$$\vec{x}_{n+1} = P_B[P_F[\vec{x}_n]] \ ,$$

the algorithm finds a fixed point after 2.18 iterations on average, and out of 10,000 trials the algorithm found the solution 3.6% of the times.

The hybrid input-output map

The hybrid input-output map (HIO) was designed to remedy the stagnation problem of ER.[17] Rewritten in the language of projections, the original HIO algorithm is,

$$\vec{x}_{n+1} = \vec{x}_n + \beta \ P_S[\left(1 + \frac{1}{\beta}\right) \ P_M[\vec{x}_n] - \frac{1}{\beta}\vec{x}_n] - \beta \ P_M[\vec{x}_n] \ ,$$

where P_S is the projection to a support constraint[9], and P_M is the projection to a Fourier magnitude constraint.

The first applications of HIO were to problems where a 2D image was to be reconstructed based on its Fourier magnitudes and a support constraint. It was experimentally determined that β's near 1 seem to have the best searching properties.[18] Originally, researchers would iterate HIO until a fixed point was found. The fixed point was observed to always be near the intersection of the constraint sets. To find the intersection, the ER algorithm was applied to the fixed point of HIO.

It was later realized the use of the ER algorithm to locate the intersection of the constraint sets is unnecessary. Actually, the fixed point of HIO yields the solution by simply applying the Fourier magnitude projection to the iterate. This can be seen by

[9]Pixels outside a central "support" region are constrained to be zero.

considering the implication of a fixed point of HIO, $\vec{x}_{fp}$. Since this is a fixed point, then,

$$P_S[\left(1+\frac{1}{\beta}\right)\ P_M[\vec{x}_{fp}]-\frac{1}{\beta}\vec{x}_{fp}]=\ P_M[\vec{x}_{fp}]\ .$$

Thus, $P_M[\vec{x}_{fp}$ is a member of both constraint sets.

Replacing the support projection of the original HIO with a binary projection, HIO becomes,

$$\vec{x}_{n+1}=\vec{x}_n+\beta\ P_B[\left(1+\frac{1}{\beta}\right)\ P_M[\vec{x}_n]-\frac{1}{\beta}\vec{x}_n]-\beta\ P_M[\vec{x}_n]\ ,$$

Alternately, another form of HIO is,

$$\vec{x}_{n+1}=\vec{x}_n+\beta\ P_M[\left(1+\frac{1}{\beta}\right)\ P_B[\vec{x}_n]-\frac{1}{\beta}\vec{x}_n]-\beta\ P_B[\vec{x}_n]\ ,$$

In this work we shall refer to these two forms as HIO$_{BM}$ and HIO$_{MB}$, where the subscript distinguishes the order of projections.

Because all of the fixed points of HIO yield solutions, HIO solves the stagnation problem of ER. This is demonstrated for both forms of HIO in figure 2.3. Shown in this figure is the same 2D cross section of the Fourier constraint hypertorus and four corners of the binary constraint hypercube. The near-intersection of the constraint sets do not trap the iterate. The algorithm detects the constraint sets do not actually intersect at the near-intersection, and escapes. The actual intersection is also shown to be locally attractive to both algorithms.

To measure the performance of this adaptation of HIO to the 1D crystal model, both forms of HIO were tested 1000 times. For each test, the initial iterate was a vector of 19 random numbers between 0 and 1, taken from a uniform distribution. For HIO$_{BM}$, a fixed point (and hence a binary vector with the correct Fourier moduli) was found on average in 74.0 iterations (the longest instance took 589 iterations to find the solution). For HIO$_{MB}$ the average was 55.2 iterations to find a solution, and after 1000 iterations the solution was found 98.7% of the time.

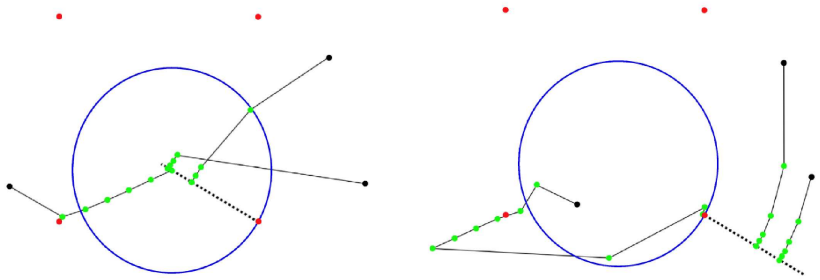

Figure 2.3: HIO$_{BM}$ is shown on the left and HIO$_{MB}$ is shown on the right. The Fourier constraint is represented with the blue circle, and the binary constraint is shown as the red dots. In each picture, three initial points (black dots) are iterated via HIO, and their trajectory is shown with the green dots. The stagnation problem of ER is remedied; the near solutions that trap ER do not trap either form of HIO. In both pictures, the dotted black line shows the manifold of fixed points. When a fixed point is reached, the fixed point always produces a solution.

Figure 2.4 shows the evolution of a sample iterate by HIO$_{MB}$. Here, a random vector of 19 real numbers is evolved via HIO, and each iterate is shown in gray scale. The iteration begins at the bottom of the figure, and progresses upwards.

As can be seen at the top of the figure, the HIO$_{MB}$ search finds a fixed point. The arrival at this fixed point is indicated by unchanging color in the figure. The iterate at the end of the search is,

$$\vec{x}_{fp} = \{0.121215, 0.32438, 0.291787, 0.289053, 0.825247, 0.813577, -0.0210301, -0.0969179, 0.746002, 0.141885, 0.217831, 1.11991, 0.420916, -0.0141976, 0.293388, 0.562162, 0.00325036, 0.721625, 0.773661\}\,.$$

Notice this fixed point is not a member of either constraint set, but it necessarily produces a solution. Both $P_B[\vec{x}_{fp}]$ and $P_M[2P_B[\vec{x}_{fp}] - \vec{x}_{fp}]$ should yield the same solution.

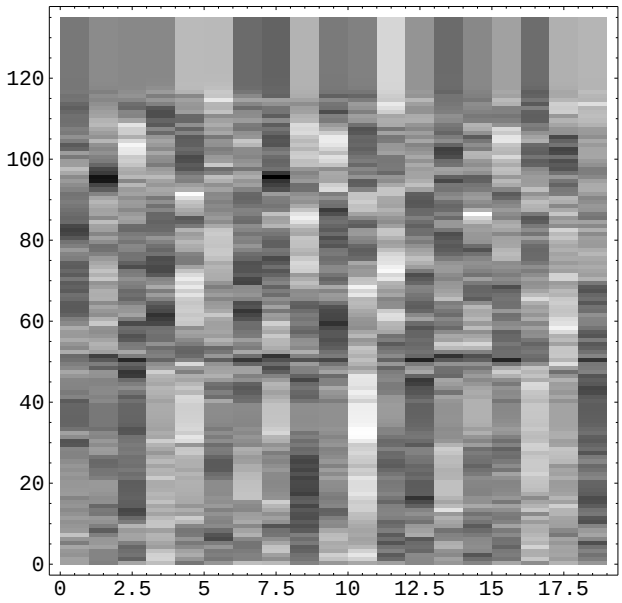

Figure 2.4: The iterate initially is random, and is displayed as the bottom row of the figure. As the iterate is evolved via HIO$_{MB}$, it is displayed in grayscale progressing upwards. The HIO$_{MB}$ explores a near-solution around iteration 38, but eventually leaves the region and continues searching elsewhere. Since the display is static beyond iteration 120, a fixed point (and thereby a solution) has been found.

In this case,

$$P_B[\vec{x}_{fp}] = \{0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1\}$$

$$P_M[2P_B[\vec{x}_{fp}] - \vec{x}_{fp}] = \{0., 0., 0., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 1., 1.\}\,.$$

Near iteration 38, the algorithm explores a near intersection of the constraints. Here, the constraints come close to each other, but don't intersect. This is the exact scenario that traps ER. However, HIO$_{MB}$ recognizes the sets don't intersect there and iterates away to explore other regions of the Euclidean space.

2.2.2 The difference map

In 2001, professor Elser attended a workshop at the University of California at Berkeley. There he learned about HIO, and its remarkable ability to recover the phases of a Fourier intensity pattern subject to real-space constraints. A natural geometer, his first reaction to learning HIO was to express it in the language of projections and constraints. Once expressed in this form, he devised a generalization of HIO that treated each constraint set equally, and reduced to either HIO$_{BM}$ or HIO$_{MB}$ with the appropriate choice of parameters.

The original DM formula,[12]

$$\begin{aligned} \vec{x}_{n+1} &= \vec{x}_n + \beta\ (P_M[F_B(\vec{x}_n)] - P_B[F_M(\vec{x}_n)]) \\ \text{where} \quad F_B(\vec{x}_n) &= (1+\gamma_B)\, P_B[\vec{x}_n] - \gamma_B\, \vec{x}_n \\ \text{and} \quad F_M(\vec{x}_n) &= (1+\gamma_M)\, P_M[\vec{x}_n] - \gamma_M\, \vec{x}_n\ , \end{aligned}$$

reduces to HIO$_{MB}$ when $\gamma_M = -1$, $\gamma_B = \frac{1}{\beta}$, and it reduces to HIO$_{BM}$ when $\gamma_M = \frac{-1}{\beta}$, $\gamma_B = -1$, and β is negative. In this way, HIO is a special case of the DM algorithm, with perhaps non-optimal parameters. If the constraint sets are assumed to be orthogonal in the vicinity of their intersection, the optimal γ's can be determined, as is shown below.

If $\vec{x}_{fp}$ is a fixed point of this map, then clearly,

$$P_M[F_B(\vec{x}_{fp})] = P_B[F_M(\vec{x}_{fp})]\ .$$

Even though the fixed point is typically not a member of either constraint set, both $P_M[F_B(\vec{x}_{fp})]$ and $P_B[F_M(\vec{x}_{fp})]$ produce a point common to both constraint sets. Like the HIO example above, the DM doesn't necessarily converge upon the true intersection of the constraint sets, but rather to a manifold of points that map to the intersection.

Typically, the sum of the dimensionality of the constraint sets is less than the embedding dimension. In the case of the 1D crystal problem explored above, the unit cell

is 19 bins long. This implies the Fourier constraint hypertorus is 9 dimensional,[10] embedded in a 19 dimensional Euclidean space. The binary constraint defines a point set (the corners of a hypercube) and is thus zero dimensional. Given a set of random (appropriately symmetric) Fourier magnitudes, the probability that the Fourier constraint hypertorus will intersect a corner of the binary constraint hypercube is zero. The fact that they *do* intersect in the example 1D crystal problem is a result of the fact that the Fourier magnitudes are not random, but were derived from an actual corner of the hyper cube. Hence the constraint sets intersect by construction.

To aid in the conceptualization of two constraint sets whose total dimensionality is less than the embedding dimension, we will here consider two orthogonal lines in 3D. Without loss of generality, we can assume one line is $L_1 = a\,\hat{x} + o_1\,\hat{z}$. This line is parameterized by a and o_1 is a constant offset from the $x-y$ plane. Similarly, let the second line be $L_2 = b\,\hat{y} + o_2\,\hat{z}$, where this line is parameterized by b and has a z-offset of o_2. The lines are orthogonal, and will only intersect if $o_1 = o_2$.

A general point in 3D can be written as $\vec{x} = x_p\hat{x}+y_p\hat{y}+z_p\hat{z}$. Projections of this point to the two constraint sets (the lines) result in $P_1[\vec{x}] = x_p\,\hat{x}+o_1\,\hat{z}$ and $P_2[\vec{x}] = y_p\,\hat{y}+o_2\,\hat{z}$.

With these definitions, the general point $\vec{x}$ can be evolved via the DM for one iteration. To calculate this, first we will calculate the F_i's, then use this in the DM formula above. The F_i's are:

$$\begin{aligned} F_1 &= (1+\gamma_1)\,(x_p\,\hat{x}+o_1\,\hat{z}) - \gamma_1(x_p\,\hat{x}+y_p\,\hat{y}+z_p\,\hat{z}) \\ F_2 &= (1+\gamma_2)\,(y_p\,\hat{y}+o_2\,\hat{z}) - \gamma_2(x_p\,\hat{x}+y_p\,\hat{y}+z_p\,\hat{z})\,, \end{aligned}$$

[10] Not coincidently there are 9 undetermined phases.

And $\vec{x}$ is moved to,

$$\begin{aligned}
\vec{x}_{n+1} &= x_p\,\hat{x} + y_p\,\hat{y} + z_p\,\hat{z} + \beta\ (P_1[y_p\,\hat{y} + (o_2 + \gamma_2 o_2 - \gamma_2 z_p)\,\hat{z} - \gamma_2 x_p\,\hat{x}] - \\
&\qquad\qquad P_2[x_p\,\hat{x} + (o_1 + \gamma_1 o_1 - \gamma_1 z_p)\,\hat{z} - \gamma_1 y_p\,\hat{y}]) \\
&= x_p\,\hat{x} + y_p\,\hat{y} + z_p\,\hat{z} + \beta\ (-\gamma_2 x_p\,\hat{x} + o_1\,\hat{z} + \gamma_1 y_p\,\hat{y} - o_2\,\hat{z}) \\
&= (1 - \beta\,\gamma_2)\,x_p\,\hat{x} + (1 + \beta\,\gamma_1)\,y_p\,\hat{y} + (z_p + \beta(o_1 - o_2)\,)\hat{z}
\end{aligned}$$

Optimal convergence rates are found when $\gamma_1 = -1/\beta$ and $\gamma_2 = 1/\beta$. With this choice of parameters, after one iteration the general point $\vec{x}$ is iterated to,

$$\vec{x} = x_p\,\hat{x} + y_p\,\hat{y} + z_p\,\hat{z} \rightarrow (z_p + \beta(o_1 - o_2)\,)\hat{z} \quad .$$

If the lines do intersect ($o_1 = o_2$), then $\vec{x} = z_p\,\hat{z}$ is a fixed point of the map, and found after one iteration. Though this fixed point is not necessarily the intersection point, it maps to the intersection point.[11]

If the lines do not intersect ($o_1 \neq o_2$), then in one iteration the DM moves the iterate onto the z-axis, as before. Subsequent iterations move the iterate away at the constant rate $\beta(o_1 - o_2)$ every iteration. Thus, if two general constraint sets approach each other, but don't actually intersect, the DM will initially approach the near-intersection, then iterate away along the axis of the constraint sets' closest approach. This behavior is shown in figure 2.5 where the two constraint sets are shown as a blue and red contour respectively. Notice that the iterate moves along the axis of the constraint sets closest approach, and that it takes steps equal in size to the constraint sets' distance apart ($\beta = 1$ in this example).

[11] In this case, $P_1[z_p\,\hat{z}] = o_1\,\hat{z}$ while $P_2[z_p\,\hat{z}] = o_2\,\hat{z}$.

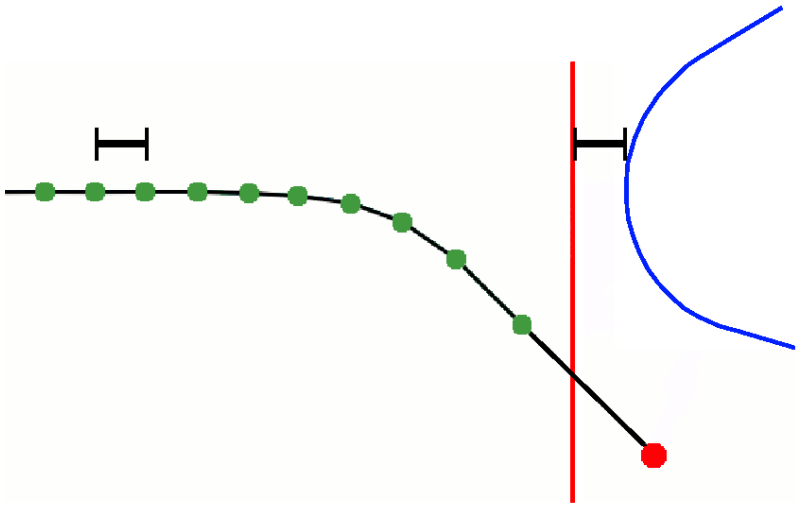

Figure 2.5: Here the two constraint sets are represented with the red and blue contours. The initial iterate of the DM is the red dot, and its subsequent evolution is shown with the green dots. For this example $\beta = 1$. Notice the iterate moves to the axis of the constraint sets' closest approach. Also, the iterate eventually leaves along that axis to continue searching elsewhere for an intersection of the constraint sets. As it moves away, the iterate takes steps equal in size to the constraint sets' closest approach.

With the optimal γ's, the DM becomes,

$$\begin{aligned} \vec{x}_{n+1} &= \vec{x}_n + \beta\ (P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)]) \\ \text{where} \quad F_1(\vec{x}_n) &= \left(1 + \frac{1}{\beta}\right) P_1[\vec{x}_n] - \frac{1}{\beta}\,\vec{x}_n \\ \text{and} \quad F_2(\vec{x}_n) &= \left(1 - \frac{1}{\beta}\right) P_2[\vec{x}_n] + \frac{1}{\beta}\,\vec{x}_n\ . \end{aligned}$$

Typically the progress of the DM is monitored by the norm of,

$$DM_{\text{error}} = \parallel P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)] \parallel\ .$$

This value is often referred to as the DM "error", and it goes to zero when a fixed point has been found.

Consider F_1 as β is varied. When parameterized by β, F_1 is the line connecting the point $\vec{x}_n$ and its projection to constraint set 1, or $P_1[\vec{x}_n]$. Similarly, F_2 can be viewed as the line going through $\vec{x}_n$ and $P_2[\vec{x}_n]$ parameterized by β. These two lines are shown in figure 2.6 as dotted black lines. The value of F_1 is shown for various β's as dark blue dots, and the value of F_2 is shown for various $\beta's$ as dark red dots. Also shown is $\vec{x}_n$ as a black dot.

In the left picture, the purple arrow is the value of the vector $P_2[F_1(\vec{x}_n)]-P_1[F_2(\vec{x}_n)]$, when $\beta = 1.5$. Notice that $\beta = 1.5$ times this vector is exactly equal to the green vector. Thus, $\vec{x}_n + \beta\ (P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)])$ would move the initial point onto the intersection of the constraint sets in one iteration. In the right picture, the purple arrow is the value of $P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)]$, when $\beta = -0.5$. Again, $\beta = -0.5$ times the purple arrow yields the green arrow. And so for $\beta = -0.5$ the initial point is again moved to the intersection of the constraint sets in one iteration.

For the case where the constraints are not orthogonal at their intersection, the DM still converges to a fixed point, though slower. The evolution of the iterate for the case where the constraint sets are not orthogonal is shown in figure 2.7. In this figure, the left picture shows the DM trajectory for $\beta = 0.7$, and the right picture shows the DM trajectory for $\beta = 1.4$ Negative β's are quantitatively the same; the DM iterate simply spirals in the opposite direction. These figures were created with the difference map explorer program (DMX). This program is explained in appendix B.

For non-orthogonal constraint sets, a different choice of the γ's would yield faster convergence, but there is usually no way of knowing *a priori* what the angle between the constraint sets is at their intersection. For this reason, the standard choice of $\gamma_1 = -1/\beta$ and $\gamma_2 = 1/\beta$ is usually used.

Note this final form of the difference map reduces to HIO$_{MB}$ for $\beta = 1$ and HIO$_{MB}$

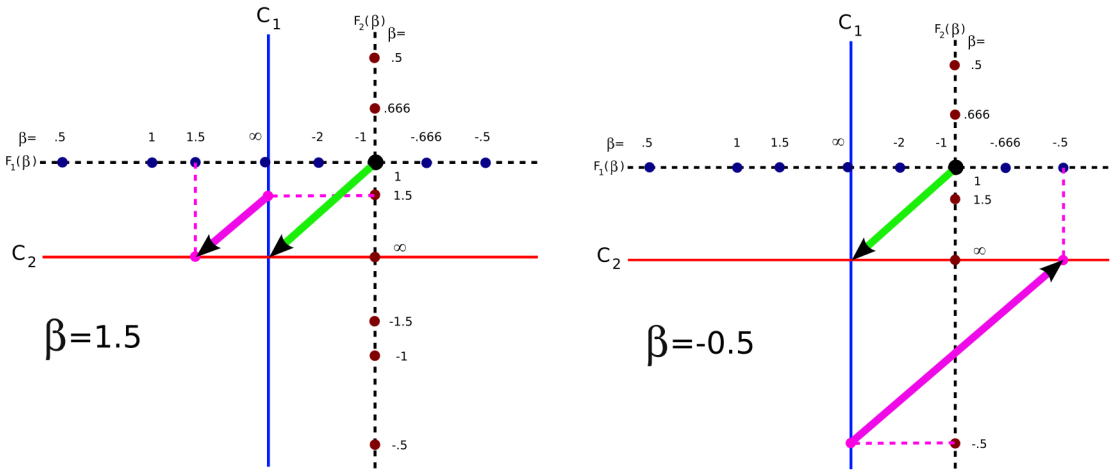

Figure 2.6: Here the value of F_1 is shown for various β's as dark blue dots, and the value of F_2 is shown for various $\beta's$ as dark red dots. The initial point is the black dot, constraint set 1 is the blue contour, and constraint set 2 is the red contour. In the left picture, the purple arrow is the vector $P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)]$ for $\beta = 1.5$, and in the right picture the purple arrow is the vector $P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)]$ for $\beta = -0.5$. In both cases, β times the purple vector yields the green arrow. Thus if $\beta\ (P_2[F_1(\vec{x}_n)] - P_1[F_2(\vec{x}_n)])$ is added to the initial point, the point is moved to the intersection of the constraint sets in one iteration.

for $\beta = -1$. For many problems, $\beta = \pm 1$ is not the optimal choice of β. For example, when applied to the 1D crystal problem, the optimal beta appears to be 0.85. The average number of iterations versus β is shown in figure 2.8.

For the 1D crystal example, the angle between the constraint set is not defined, since one constraint set is a point set. According to figure 2.8, the DM still converges for a wide range of β, but the choice of $\gamma_1 = -1/\beta$ and $\gamma_2 = 1/\beta$ may be far from optimal for this application. This is explored in figure 2.9. Here, $\beta = 1$, and a range of γ's are explored. Apparently, if $\gamma_1 = -1.3$ and $\gamma_2 = .9$, the DM converges slightly faster. With these choices for the γ's, the DM finds a fixed point (and hence a solution) in 45.3

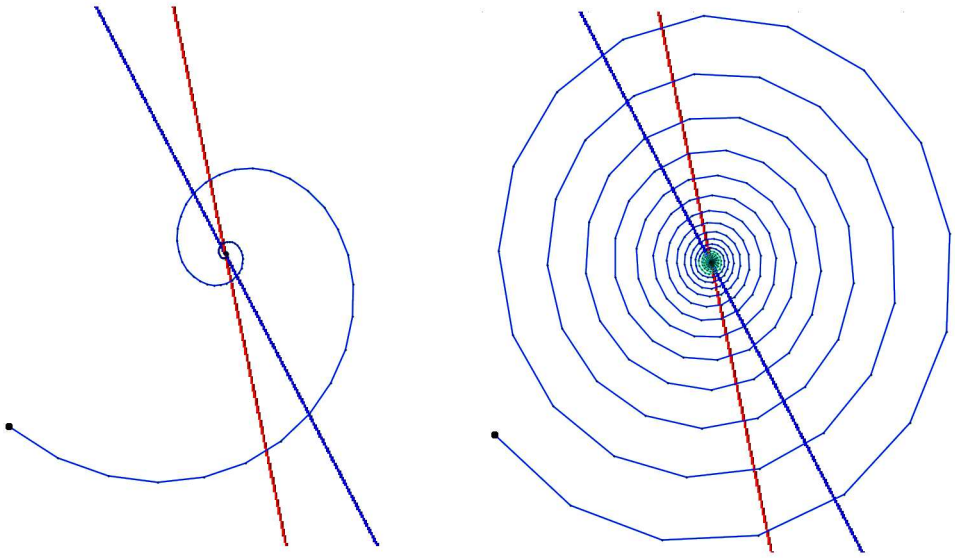

Figure 2.7: The constraint sets (clearly not orthogonal) are represented by a red and blue contour. In the left picture, the DM used $\beta = 0.7$. In the right picture, $\beta = 1.4$ was used. Negative β's perform qualitatively the same, the DM simply spirals in the opposite direction. These figures were generated with the difference map explorer program (DMX). This program is the subject of appendix B.

iterations on average, while with the standard choice for the γ's, it takes 55.2 iterations on average.

Recently a program was developed titled the difference map explorer (DMX). The operation of the DMX is explained in appendix B. The program enables the user to set up many different constraint set geometries in 2D, and explore the convergence rate of the difference map for a given geometry. For example, consider the constraint sets shown in figure 2.10. Here, the two sets are shown as red and blue contours.

The DMX program considers every pixel in a user defined region, and iterates the pixel via the DM. The program records how long it takes each pixel to find a fixed point, and colors the pixel accordingly. For the constraint geometry displayed in figure 2.10,

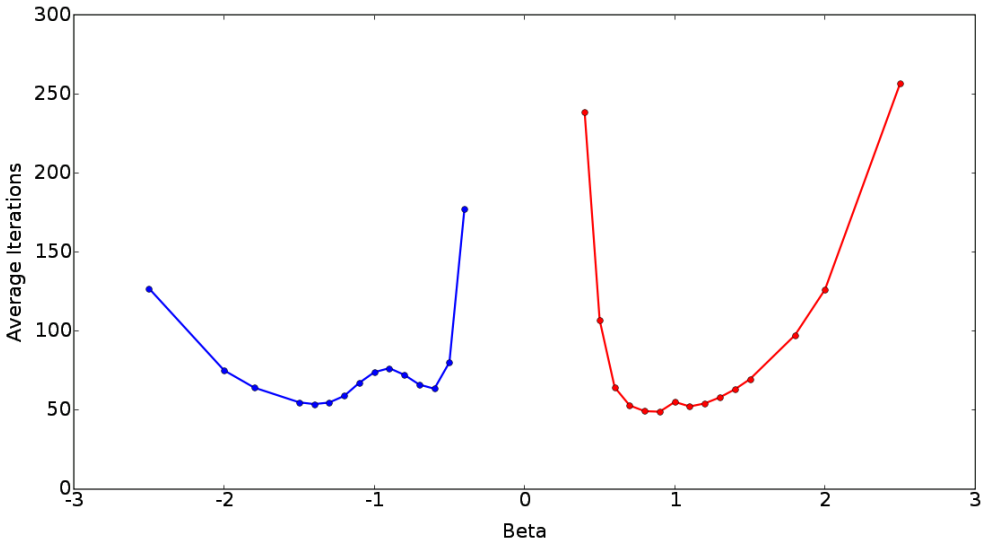

Figure 2.8: In this figure, the DM is applied to the 1D crystal problem. The average number of iterations is plotted for many different β's. The data points are an average of 80,000 DM searches. Negative β's are shown in blue, and positive β's are shown in red. The optimal β seems to be about 0.85 .

the convergence rate for the nearby pixels is shown in figure 2.11. In this figure, the left picture is the convergence rate, per pixel, with $\beta = 1$ and the right picture uses $\beta = -1$. The constraint sets are also shown overlayed. In each picture, the dotted line is the set of fixed points for the DM.

Though it was originally developed for the purpose of retrieving obfuscated phases in diffraction experiments, the DM has been found to be a very general and effective search algorithm in other contexts as well.[15] It is effective at solving 3-SAT problems, discovering Ramsey numbers, solving sudoku puzzles, finding solutions to Diophantine equations, and finding low energy protein conformations (as will be shown in chapter 3). All of these applications have a common theme; they are problems that can be split

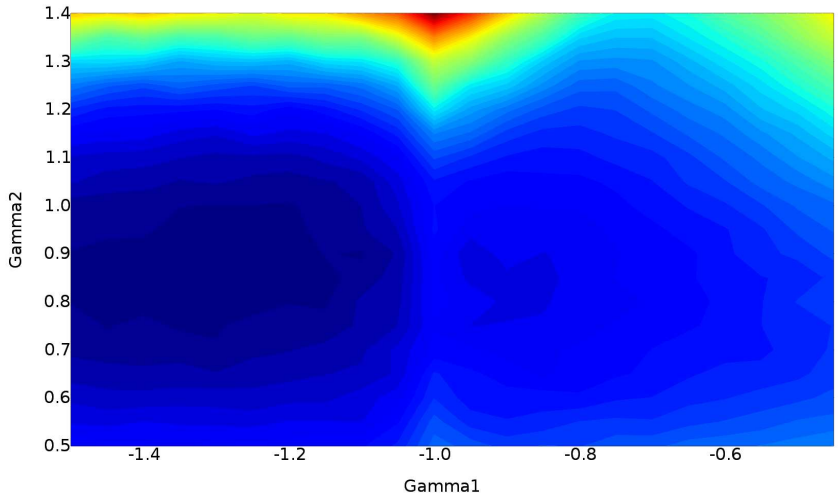

Figure 2.9: In this figure, the DM is again applied to the 1D crystal problem. The average number of iterations is displayed for $\beta = 1$ and a range of γ's. The data points are an average of 16,800 DM searches. For orthogonal constraint sets, $\gamma_1 = -1/\beta$ and $\gamma_2 = 1/\beta$ are optimal. For the 1D crystal problem though, $\gamma_2 = 0.9$ and $\gamma_1 = -1.3$ seems optimal, yielding an average of 45.3 iterations to converge. For the traditional choice of parameters, $\gamma_1 = -1$ and $\gamma_2 = 1$, the DM takes 55.2 iterations on average.

into two simultaneous sub-problems, each of which is trivial to solve alone.

There has been a lot of work trying to adapt the DM to accommodate more than two constraints. For example, consider a problem that can be split into three subproblems, each of which is easy to solve alone. Furthermore, assume any two of the subproblems can be solved easily together, but solving all three simultaneously is challenging. At the moment, there is no effective generalization of the DM that can accommodate such a problem.

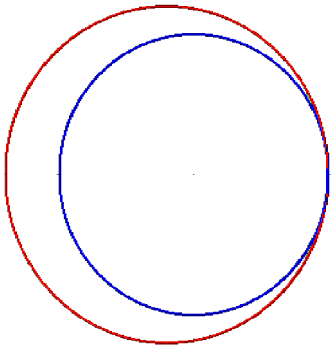

Figure 2.10: These constraint sets were arranged using the DMX program, which is described in appendix B. The two constraint sets are represented with a red and blue contour respectively.

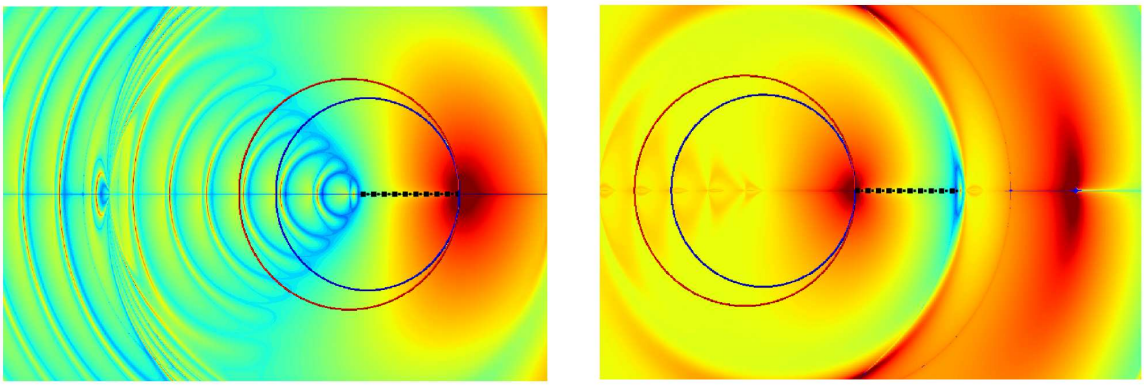

Figure 2.11: In this figure, each pixel is iterated via the DM, and the color of the pixel is proportional to how long the pixel takes to find a fixed point. Dark blue pixels find a fixed point quickly, while dark red pixels iterate a long time before finding a fixed point. The left picture iterates the DM with $\beta = 1$, and the right picture uses $\beta = -1$. In each picture the set of fixed points is shown as a black dotted line.

Chapter 3

The DM applied to protein folding

The bulk of this chapter is based on a paper currently in preparation.[43]

3.1 Introduction

As explained in chapter 1, the three dimensional structure of a protein largely determines its function. For proteins that form high quality crystals, x-ray crystallographic methods have long enabled researchers to determine the protein's structure. For proteins that fail to form high quality crystals, NMR spectroscopy is a time consuming and expensive alternative.[1] Currently, there are many proteins with known sequences and unknown structure. As an alternative to x-ray crystallography and NMR spectroscopy, a reliable structure prediction method would be a tremendous asset to biological research. The various structure prediction methods are compared at the semi-annual CASP experiment.[29] In the past CASP experiments, the most successful structure prediction method has been homology modeling. In recent years though, *ab initio* methods have started to become competitive.

The method of homology modeling[2] is based on the observation, that when two proteins have similar amino acid sequences, they also usually have similar structural properties.[35] Using this method, a protein's structure is determined first by comparing its amino acid sequence against other determined structures in the Protein Data Bank, and finding similar sequences.[50] For example, if a particular subsequence of amino acids almost always forms an alpha helix, then if found in the undetermined protein's

[1] Both x-ray diffraction and protein NMR are explained in chapter 1.2.
[2] Homology modeling is more thoroughly explained in chapter 1.3.1.

sequence, the structure of this sub-sequence can be safely guessed. In this way, the structure is piece-wise determined, and subsequently assembled. [4] This technique relies heavily on the availability of similar template sequences whose structures have been determined. For large classes of proteins, such as membrane proteins, there is a dearth of templates for comparison. For such proteins, homology modeling currently offers little promise.

With *ab initio* structure prediction,[3] the protein is modeled as a collection of atoms[44, 47], or united atoms [22, 27], and the native structure is assumed to be the global minimum of an appropriate energy function.[1] Because the actual energy function proteins navigate is difficult to precisely calculate, there are numerous approximations in use. Finding the global minimum of the energy function is itself a very challenging endeavor,[39] and many different methods have also been developed for this. All of the modern energy minimization algorithms require great computational resources, and *ab initio* methods have been limited to small proteins (approximately fifty amino acids).

In this work we consider a very simple energy function (explained in appendix A) that calculates an energy for any protein conformation. We propose a new method for finding the global minimum of the non-bonded energy function. The proposed method, the *difference map* (DM),[4] operates in a very different manner than previous search algorithms used to minimize the conformational energy. Most energy minimization methods are based on a Monte Carlo exploration of the protein conformation's energy landscape. For these methods, the "iterate" is an evolving protein conformation. Contrasting this, the DM "iterate" is not a protein conformation, but an atomic configuration. Since a polypeptide has on average about three degrees of freedom per amino acid, and an atomic configuration has three degrees of freedom per atom, the iterate of the DM

[3] *Ab initio* structure prediction is explained more fully in chapter 1.3.2.

[4] The difference map is explained in chapter 2.

searches a much larger space than that explored by Monte Carlo methods. Searching this larger space is not necessarily a liability: deep local minima in the energy landscape that would trap a Monte Carlo iterate are easily escaped by the DM, since the DM can evolve the iterate in directions not accessible to the Monte Carlo iterate.

The DM overcomes three fundamental deficiencies of all Monte Carlo based search techniques. First, in all Monte Carlo based searches, the entire folding pathway needs to be simulated in order to find the lowest energy configuration. The DM overcomes this by immediately searching for the lowest energy state, without regard to the folding pathway. Second, Monte Carlo search methods have a tendency of getting stuck in deep local minima of the energy landscape. There have been many modifications to the basic method to overcome this problem[3, 45]. However, though lessened, the problem remains to a degree. Contrasting this, the DM escapes even deep local minima in the energy landscape, and spends very little time exploring them. And finally, Monte Carlo search methods update the iterate by local modifications to the protein conformation, thus limiting the rate by which the protein conformation can evolve. The DM typically makes large modifications to the iterate prior to finding a fixed point. Every fixed point of the DM algorithm is locally attractive, and corresponds to a low energy protein conformation.

We have applied the DM algorithm to an all-atom protein model (sidechain hydrogens have been omitted, though backbone hydrogens are included for the purpose of hydrogen bonding). The performance of the DM is compared to that of a popular Monte Carlo method, parallel tempering (PT).[5] To make the comparison meaningful, the two algorithms are each run on the same computer, running the same amount of time. The atomic potential used is as simple as possible, involving only hydrophobic-hydrophilic

[5]Parallel tempering is explained in chapter 1.3.2.

interactions, hydrogen bonds, and steric repulsion.[6] Though simple, this potential is able to correctly reproduce the general structure of the native fold of the staphylococcus aureus A protein (B domain (10-55)). In this chapter we will refer to this protein as "protein A".

The DM is applied to the problem of minimizing the protein energy by using the program NENA. The source code for NENA can be found in appendix C. Details about how the program functions can be found in chapter 4.

3.2 Theory

3.2.1 Constraints and projections

The difference map (DM) is an iterative algorithm where the iterate (an atomic configuration) is evolved by means of *projections* onto two constraint sets. The first constraint set is the set of atomic configurations that have a valid peptide geometry. A member of this constraint set has all bond lengths, bond angles, and left handed versus right handed orientations correct (the bond lengths and angles are taken from Engh 1991[16]). This is the set of the rotamer configurations. Most contemporary Monte Carlo searches have the folding protein always a member of this constraint set; this defines what is commonly understood as the folding landscape. The second constraint set used by the DM is the set of atomic configurations whose non-bonded energy is less than a predefined target energy. When freed of the peptide geometry constraint, it is easy to find a member of this constraint set. It is clear that when an atomic configuration is found that is a member of both constraint sets simultaneously, the problem has been solved. In this case, an atomic configuration that has both a valid peptide geometry, and a sufficiently

[6] Details about the potential can be found in appendix A.

low energy, has been found. The two constraint sets are described in detail in appendix A.

We represent an atomic configuration by $\vec{\mathbf{R}} = \{\vec{\mathbf{r}}_1,\ \vec{\mathbf{r}}_2,\ \ldots\}$, where $\vec{\mathbf{r}}_i$ is the 3D coordinate of atom i. For both constraints, the projection to that constraint set is defined as the closest atomic configuration that satisfies the constraint. In this chapter, $P_G\left[\vec{\mathbf{R}}\right]$ denotes the projection to the peptide geometry constraint, while $P_E\left[\vec{\mathbf{R}}\right]$ denotes the projection to the energy constraint.

For the geometry constraint, the projection is accomplished by minimizing a penalty function (defined in appendix A) via an adaptive step-size steepest descent algorithm (the algorithm is explained in chapter 4.4.1). This projection performs a minimal modification to the atomic configuration that yields a member of the geometry constraint set. The result of this projection has a valid peptide geometry, but non-bonded atoms are allowed to overlap, and in general there is no bias toward a low energy atomic configuration.

To compute the projection to the energy constraint, the energy function defined in appendix A is minimized until the non-bonded energy is below a predefined target energy. Though the result of this projection is a low energy atomic configuration, the configuration in general does not have a valid peptide geometry. For a typical member of this constraint set, bond and angle constraints are usually not satisfied. While computing this projection, the protein behaves as a liquid of independent atoms, rather than as a linked chain of amino acids.

3.2.2 Difference map algorithm

As a simple pedagogical step toward understanding the DM algorithm, first consider the following alternative algorithm, called *alternating projections* (AP):

$$\vec{\mathbf{R}}_{n+1} = P_G\left[P_E\left[\vec{\mathbf{R}}_n\right]\right]$$

For AP, the iterate is projected to the energy constraint, followed by a projection to the geometry constraint. With the projections in this order, the iterate is perpetually a member of the geometry constraint set. This algorithm greedily minimizes the distance between the two constraint sets, and quickly evolves toward a fixed point, where the distance between the two constraint sets has a local minimum. This is essentially Fienup's error reduction algorithm,[20] described in chapter 2.2.1.

To contrast AP and the DM, they are both applied to a 2D example problem in figure 3.1. In this example, the two constraint sets are the red and blue curves, DM iteration is shown as green dots, and AP iteration is shown as the gold dots. If the initial iterate is close to an actual intersection of the constraint sets (the top red dot in figure 3.1 for example) then AP will converge to the intersection. However, AP is prone to stagnating at places where the distance between the constraint sets is locally minimized (the bottom trajectory in figure 3.1, for example). Finally, note that the iterate of AP is always a member of the blue constraint set.

The DM iterate is updated by $\vec{\mathbf{R}}_{n+1} = \vec{\mathbf{R}}_n + \vec{\mathbf{d}}$, where

$$\vec{\mathbf{d}} = P_E\left[2P_G[\vec{\mathbf{R}}_n] - \vec{\mathbf{R}}_n\right] - P_G\left[\vec{\mathbf{R}}_n\right] \ .$$

Clearly a fixed point has been found when $\vec{\mathbf{d}} = \vec{\mathbf{0}}$. If $\vec{\mathbf{R}}^*$ is a fixed point of the DM, the corresponding solution ($\vec{\mathbf{R}}_{sol}$) is given by

$$\vec{\mathbf{R}}_{\text{sol}} = P_E\left[2P_G[\vec{\mathbf{R}}^*] - \vec{\mathbf{R}}^*\right] = P_G\left[\vec{\mathbf{R}}^*\right] \ .$$

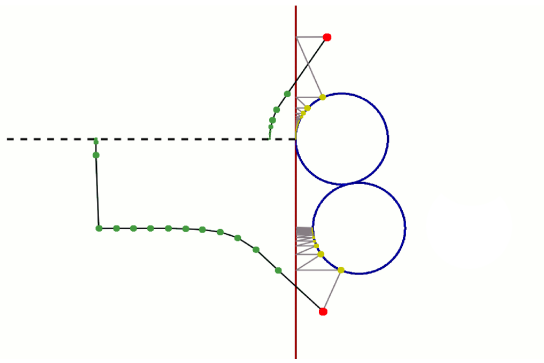

Figure 3.1: The two constraint sets are shown as red (vertical line), and blue (circles). Two initial points (red dots) are iterated via the DM (green dots, black line) and AP (gold dots, gray line). The dashed line is the set of fixed points of the DM. Every fixed point of the DM maps to the point common of both constraint sets. For the top initial point, both search algorithms find the solution. For the bottom initial point, AP stagnates at a near intersection of the constraint sets, while the DM is repelled by this near intersection, and eventually finds the actual intersection. The iterate of AP is always a member of the blue constraint set.

Since $\vec{\mathbf{R}}_{\text{sol}}$ is simultaneously equal to a projection to each constraint set, it both has the correct peptide geometry, and a sufficiently low energy. When the iterate is sufficiently near a fixed point, the attractive property of the DM leads to monotonic convergence to the fixed point. Since every fixed point yields a solution, the DM is not prone to the same stagnation problems AP suffers from.

The extent to which the native conformation of protein A is an attractive fixed point of the DM is shown in figure 3.2. Here, the initial iterate of the DM was chosen by adding random vectors, of given amplitude, to protein A's atomic coordinates. The DM then evolved the iterate, and terminated when the iterate converged upon a fixed point. The perturbation followed by DM iteration was tested 100 times, for many different magnitudes of the perturbation. The average number of iterations before a fixed point

was found is displayed. Given the same initial iterate, the convergence rate of the alternating projection map was tested in the same way, and is also shown.

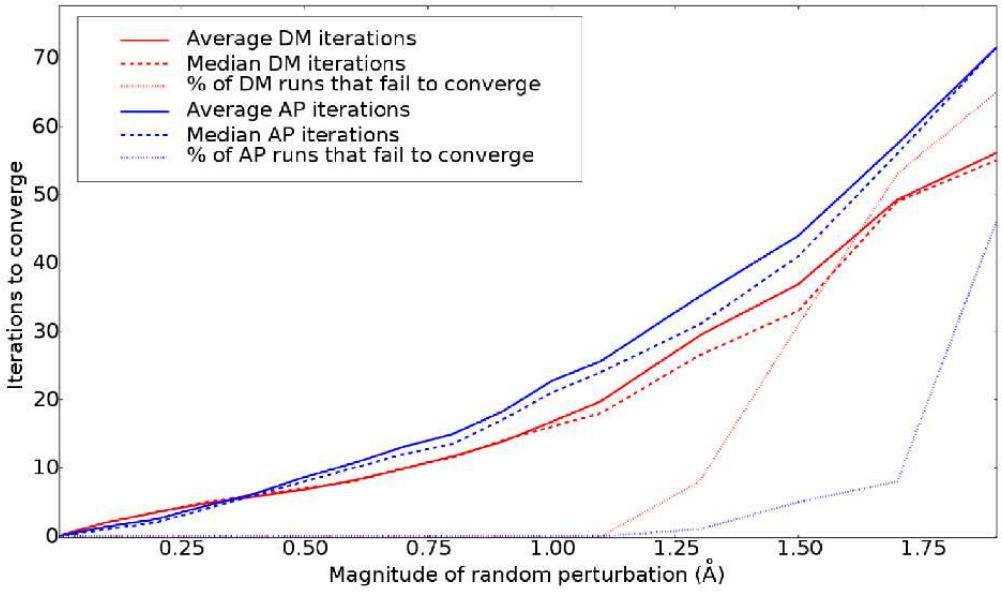

Figure 3.2: In this figure, both algorithms begin searching close to the native fold of protein A. If the atomic configuration is within 1 Å RMSD of the native fold, both search algorithms always converge upon the native fold, on average within 30 iterations. Above 1.25 Å RMSD, both algorithms occasionally fail to recognize the nearby intersection of the constraint sets.

Though prone to stagnation, AP is a useful algorithm for finding the nearest local minimum of the conventional energy landscape. This energy refinement is done by first projecting the atomic configuration to the geometry constraint (yielding a valid protein conformation) and evaluating the atomic configuration's non-bonded energy. Next, the atomic configuration is projected to the energy constraint with a projection target energy only slightly lower (1 unit of energy) than the current energy (this moves the iterate a small step in the downhill gradient direction). Finally, the atomic configuration is again

projected to the geometry constraint. These alternating projections, with the target energy continually being lowered, quickly lowers the energy of the protein conformation, and eventually finds a fixed point at a nearby local minimum in the energy landscape. These are the same local minima that could potentially trap a Monte Carlo search iterate.

To generate low energy protein conformations, the program NENA[7] iterated the DM for seven days on four parallel processors, each 3 GHz. Each processor operated independently of the others, and every search began with a configuration of random atom positions. The initial atom coordinates where chosen from inside a box with a uniform probability distribution.

Every three hundred DM iterations, the current iterate was refined via AP until a nearby fixed point of AP was found. The fixed point was the nearest local minimum of the conformational energy landscape, and represented the best estimate for an atomic configuration satisfying both the geometry constraint and the energy constraint. The energy and RMSD (all atoms) of these estimates are plotted in figure 3.4 (green dots).

After refining via AP, if the energy of an estimate was below the target energy of the energy constraint, the target energy was lowered to this new lower energy. The iterative DM search was then restarted with a new random initial atomic configuration. On the other hand, if after refining the energy of the estimate was above the cutoff energy, the DM iterate was replaced with the refined estimate, and DM iterations continued.

The progress of the DM search is monitored with the DM "error".[8] The DM error, in our protein problem, is the rms displacement of atoms in one DM iteration. When a fixed point is found the DM error goes to zero, since the atomic configuration no longer moves. The DM error is by no means monotonic. During the search the error can be at a low value, indicating the iterate is in a region where the constraint sets are close

[7]NENA is the subject of chapter 4.

[8]The DM error is defined in chapter 2.

together. If the constraint set's are close together but don't actually intersect, the DM moves the iterate away and continues searching elsewhere in configuration space.

An example error history can be seen in figure 3.3. As can be seen in the figure, every 300 iterations the DM refines the atomic configuration via AP. At iteration 300, the result of this refinement yielded an atomic configuration with an energy below the target energy. This atomic configuration was output to a file, and the DM was restarted with a lower energy target, and a new random iterate.

3.2.3 Parallel tempering algorithm

For the sake of comparison, the same computers were used to conduct a *parallel tempering* (PT) minimization of the proteins non-bonded energy.[9] PT has had significant success in folding small proteins.[10, 45] The method is a modified Monte Carlo search. For each search, there are several clones of the same initial atomic configuration. Each clone is evolved via Monte Carlo steps at a different temperature. At every iteration there is a probability of a swap between any two clones (a swap consists of switching their temperatures). The probability is a function of the clones' current energies and temperatures. Additionally, after a large number of Monte Carlo iterations, the atomic configuration of the lowest energy clone replaces the atomic configuration of the hottest clone.

The Monte Carlo step is computed by adding to the iterate's atomic coordinates, random vectors of a given magnitude. After this perturbation, the atomic configuration is projected to the geometry constraint set. The result of these two operations is a slightly perturbed atomic configuration that has a valid peptide geometry. After the perturbation and projection, the energy of the new protein conformation is calculated, and the proba-

[9]The theory behind parallel tempering is explained in greater detail in chapter 1.3.2.

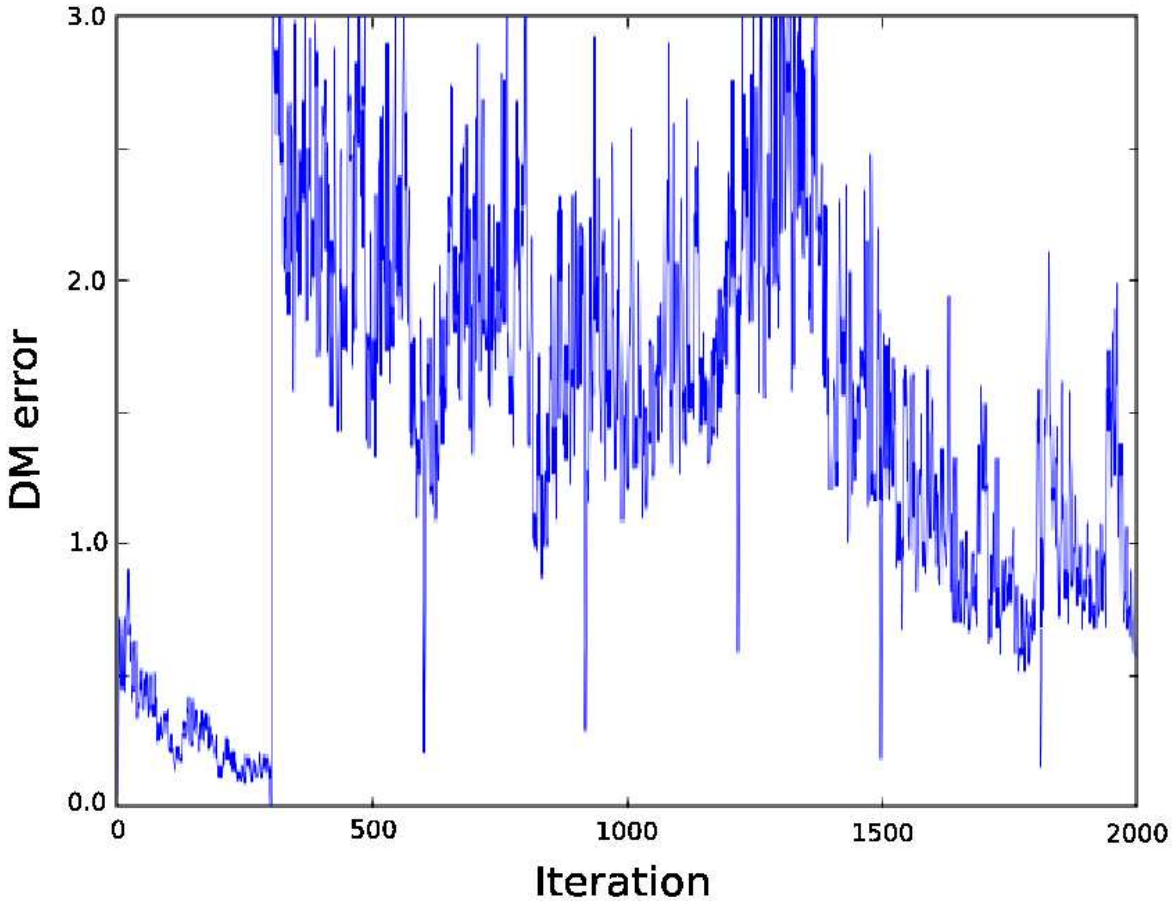

Figure 3.3: This DM error data was generated by NENA, the protein conformational energy minimizer program. The DM error is defined in chapter 2. The DM error is a measure of how far the iterate has moved during one DM iteration. When a fixed point is reached, the error goes to zero, as can be seen at iteration 300. When this happens, the protein conformation (which necessarily has an energy below the energy target) is output to a PDB file. The DM then restarts with a random iterate and a lower target energy.

bility of accepting or rejecting the test step is calculated. The magnitude of the random perturbation is adjusted for each temperature to maintain a step acceptance rate of 50%.

The PT simulation also searched for seven days on four processors, each 3 GHz. We

used four random initial configurations. The initial atomic coordinates were generated by first choosing atom positions from inside a box with a uniform probability distribution, and then projecting the atomic configuration to the geometry constraint. For each of the four simulations, we used fifteen clones, whose temperatures ranged from 2.92 to 0.01 (in the energy scale described in appendix A). A temperature of 2.92 was hot enough that the clone with this temperature quickly explored the energy landscape, and spent very little time in any one energy minimum. On the other hand, the clone with a temperature of 0.01 was essentially frozen: its energy was low, and fluctuated only very little. Each of the four simulations made consistent progress toward the solution. The progress of each of the four simulations was very similar to that found in previous studies[45].

With our PT code, we averaged 1.7 seconds per iteration, for a single computer, with fifteen clones. This is close to previously published iteration rates. In their 2005 paper[45], Schug *et. al.* averaged about one million iterations per fifteen clones using fifteen computers in one day, for a protein with five amino acids. This corresponds to about ten seconds per iteration, for a single computer, with fifteen clones, iterating a protein with forty amino acids. The fact that our PT iterations are faster is due to our comparatively simple potential.

3.3 Results

After one week of searching, each algorithm found many low energy atomic configurations. The RMSD (all atom) from the native fold versus the energy of these folded proteins is shown in figure 3.4. As can be seen from the figure, the PT simulations are still progressing towards lower energy conformations. Previous studies suggest the PT simulations will find the global minimum of the energy landscape when given enough

time.

In the same amount of computation time, the DM found many low energy conformations. The lowest energy conformation the DM found had little resemblance to the native fold. The conformation with the lowest RMSD (all atom RMSD= 4.4Å) found by the DM had an energy of -71.1. The native fold had an energy of -66.6. This protein conformation is shown in figure 3.5.

We do not claim that the lowest energy protein conformation found by the DM is the lowest possible, only that given the same amount of time and resources, the DM finds many more low energy states than PT, as is evident from figure 3.4.

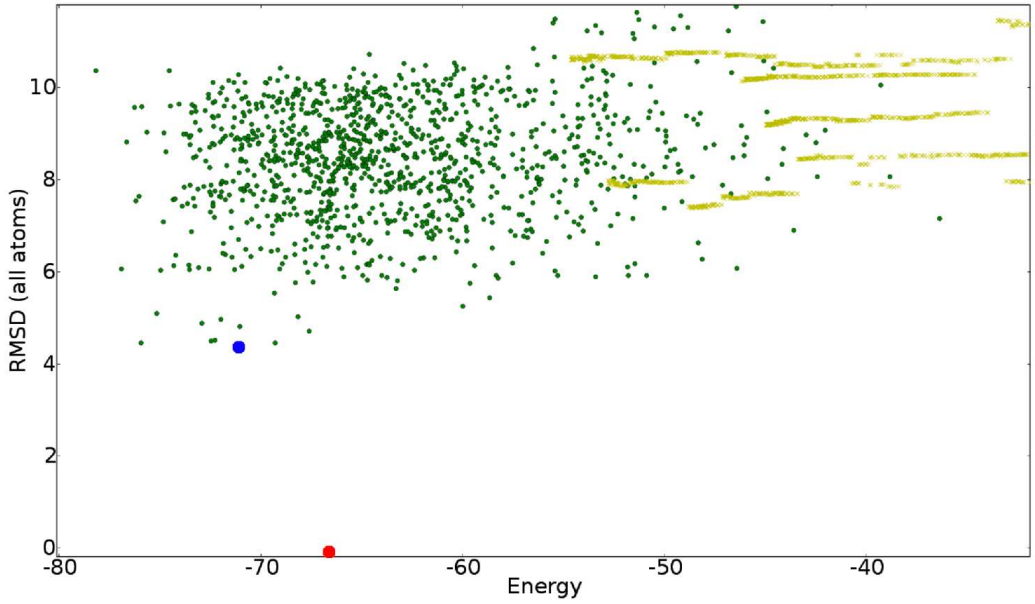

Figure 3.4: The green dots are the output configurations found by the DM, while the gold crosses are those found by PT. Clearly the PT simulations are still progressing toward lower energy. Both methods ran one week. The blue dot is the most native-like fold discovered, and the protein conformation is shown in figure 3.5. The red dot is the native fold, also shown in figure 3.5. Because of our simplistic potential, many protein conformations with lower energies than the native fold are discovered.

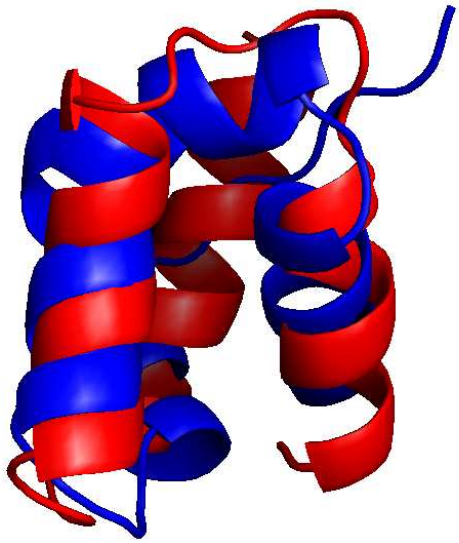

Figure 3.5: This protein conformation was found during one week of computation. It has an RMSD (all atom) of 4.4Å, and an energy of -71.1. The native fold (red) has an energy of -66.6.

There is really only one energy parameter in our potential: a relative energy scale between hydrogen bonding and the hydrophobic effect. For a given set of energy parameters, the native conformation was refined via AP to locate the nearest local minimum in the energy landscape. In this low energy conformation (only slightly different from the native conformation) the ratio of the hydrophobic energy (E_{HP}) and the hydrogen bonding energy (E_{HB}) was evaluated. We call this ratio $F_{HB} = \frac{E_{HB}}{E_{HB}+E_{HP}}$.

For the proteins 1bba, 1enh, 1gab, 1gjs, 1guu, 1vii, and 1ba5, the relative energy scale between hydrogen bonding and a hydrophobic core was adjusted so that F_{HB} ranged from about 0.90 (implying the total energy of the native conformation was mostly due to hydrogen bonding) to about 0.50 (implying an equal contribution to the energy of

the native conformation from hydrogen bonding and hydrophobicity). F_{HB} values are given in table 3.1. For every choice of energy parameters, NENA was run and the energy of the lowest energy protein conformation is displayed. Unlike the folding of protein A described earlier, which was allowed one week of searching, the results displayed in table 3.1 are based on only two days of searching.

3.4 Discussion

As can be seen in figure 3.4, the PT data points form an almost continuous trajectory. There were four different PT simulations, and there can be seen approximately four yellow streaks, each occasionally broken by a discontinuity. PT is limited by making small steps on the folding landscape. Because of this, the PT simulations in effect reconstruct a folding pathway. However, if the goal is to find the lowest energy conformation, then simulating the entire folding pathway is unnecessary. While the PT algorithm was simulating the folding pathway of the protein, the DM was searching for low energy conformations directly, with no regard for the folding pathway. This accounts, to a large extent, for the superior performance of the DM algorithm.

The most native like protein conformation produced by the DM is shown in figure 3.5. This protein conformation, like the native fold, has three helices, in roughly the same locations. The reason the native fold is not the lowest energy fold possible is due to our minimalistic potential. We used a computationally simple potential to show the effectiveness of a DM search as compared to PT. Many different choices of potential parameters were explored in the process of trying to find a potential with the property that the native conformation is the lowest energy fold, but for nearly every choice of potential parameters a conformation was found with an energy lower than the native fold. It seems a more realistic potential is necessary to have the ground state be the

Table 3.1: The fraction of the native conformation's energy that is due to hydrogen bonding is shown ($F_{HB} = \frac{E_{HB}}{E_{HB}+E_{HP}}$). This is adjusted by varying the energy scale between hydrogen bonding and hydrophobic effect. For every choice of F_{HB}, the native conformation's energy is given along with the best fold discovered by NENA. This data was gathered by allowing the DM to search for the time displayed.

protein (PDB code)	F_{HB}	native E	lowest E discovered	search time (3 GHz)
1bba	0.86	-33.2	-34.8	45 hours
	0.69	-50.4	-56.0	9 hours
	0.5	-56.7	-60.3	35 hours
1enh	0.85	-71.4	-80.0	45 hours
	0.62	-105.2	-112.5	9 hours
	0.38	-125.5	-126.6	35 hours
1gab	0.89	-61.2	-66.7	45 hours
	0.72	-88.6	-89.7	9 hours
	0.53	-101.0	-102.9	35 hours
1gjs	0.89	-64.1	-66.3	45 hours
	0.75	-92.1	-91.2	9 hours
	0.59	-106.4	-109.2	35 hours
1guu	0.87	-57.8	-65.1	45 hours
	0.70	-85.3	-86.4	9 hours
	0.48	-102.9	-103.7	35 hours
1vii	0.84	-43.6	-50.1	45 hours
	0.60	-67.1	-71.0	9 hours
	0.33	-82.7	-86.7	35 hours
1ba5	0.87	-62.6	-65.6	45 hours
	0.67	-95.4	-94.0	9 hours
	0.43	-111.0	-116.4	35 hours

native fold. We believe the superior performance of the DM algorithm over PT will extend to more realistic potentials as well.

In this chapter, we demonstrate a new search algorithm, based not on the physical pathway of the folding processes, but on the geometry of constraint sets. Our results show the DM algorithm can be successful in finding low energy states for a given potential. The algorithm is both easy to implement, and is easy to run in parallel. It is our hope that this new method for finding low energy atomic configurations will facilitate the development of more precise atomic potentials, since the most important feature of a useful potential is that the native fold has the lowest energy.

Chapter 4

NENA

Recently a powerful new program was written that efficiently minimizes the non-bonded energy of a protein conformation. Entitled NENA, the program's efficacy is demonstrated in chapter 3, where it is compared to a contemporary energy minimizing algorithm, parallel tempering. In the results of that chapter, the difference map search algorithm (which NENA uses) is found to produce many more low energy atomic configurations than parallel tempering in the same amount of time.

NENA has a convenient graphical user interface (GUI) written in Python, while most of the intensive computation is executed in C. The entire source code is reproduced in appendix C. The program's code structure is very similar to the simpler program, the difference map explorer (DMX). The code for DMX is thoroughly explained in appendix B. Before trying to decipher the NENA code (which has abundant comments in it), the reader is encouraged to read and understand the DMX code reproduced in appendix B.

In this chapter, the operation of NENA will be explained, as will the general principles of how the program minimizes the non-bonded energy of an atomic configuration.

4.1 Using NENA

In this section, the various buttons will be explained and referred to with *italics*, while the text boxes and their associated parameters will be referred to with **bold** font.

To run NENA, open a terminal in the directory containing the files in appendix C, and type "python nena.py". If everything works properly, the program should open an interface like figure 4.1.

To begin minimizing the non-bonded energy of a protein, type in the name of the

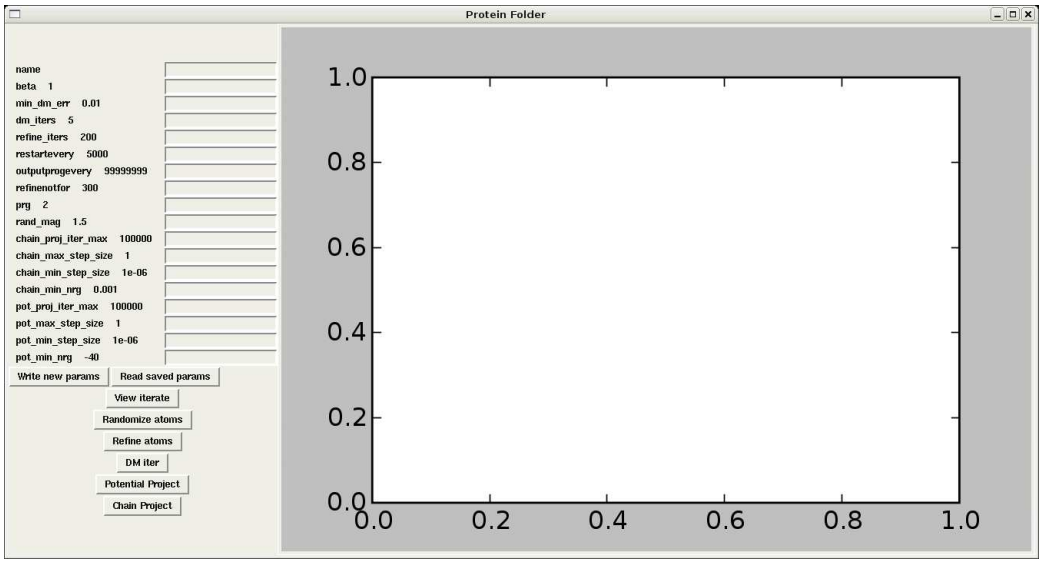

Figure 4.1: On the left are the various DM parameters and text boxes. On the right is the plotting area.

PDB structure file in the **name** text box, for example "1bdd.pdb".[1] Once done, press the *write new params* button. This will load the PDB file into the NENA program. To verify the protein was loaded correctly, press the *view iterate* button. This will open a Rasmol molecular viewer window, and you should be able to see the loaded protein. It should look like, for example, figure 4.2.

Once the protein is properly loaded, the various buttons have the following effects on the atomic configuration.

First, the *write new params* button saves newly input parameters into the NENA program. For example, to use a different DM β, [2] first the new value for β is input into the **beta** text box, then the *write new params* button is pressed. The new parameters are then displayed next to the appropriate variable names. A further effect of pressing the *write new params* button is the current parameters are stored in the "params.db" file in the

[1] Files of the type "*.pdb" are currently supported. Other file input types should be added later.

[2] See chapter 2 for an explanation of this parameter.

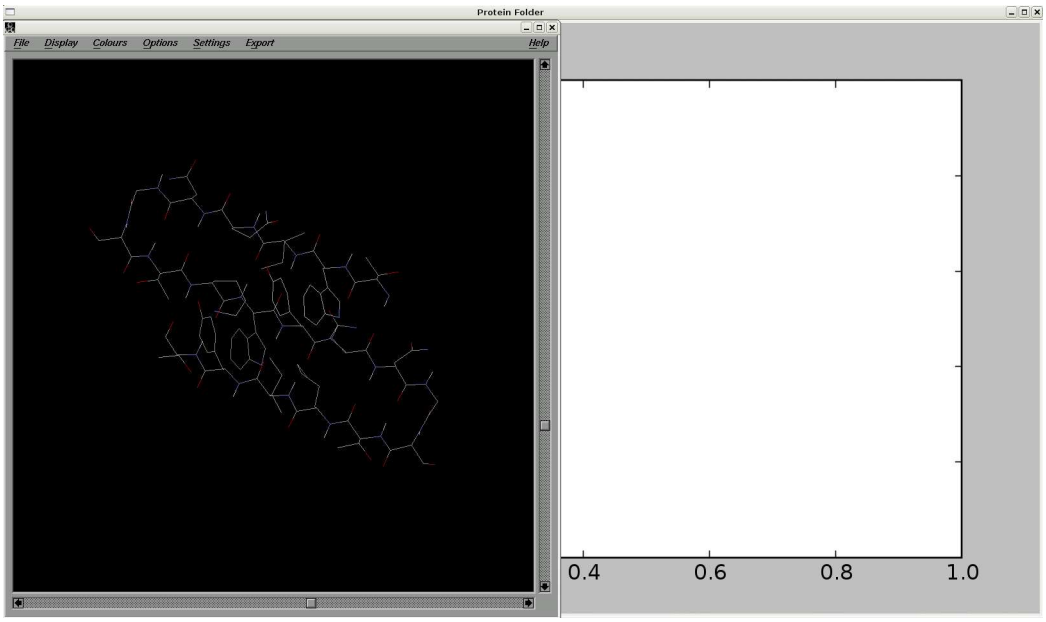

Figure 4.2: When a protein is loaded into NENA, pressing the *view iterate* button open a RASMOL window, and the protein can be inspected. During DM iteration, if the user presses the *view iterate* button, the DM iterate will be displayed. The DM iterate typically does not have a valid peptide geometry, and in this case RASMOL sometimes draws bonds where they do not belong.

local directory. Also, the current atomic configuration is written to the "atoms_tmp.db" file.

The *read saved params* button does the inverse of the *write new params* button. Pressing this button will load the local parameter file "params.db" created by pressing the "write new params" button. Furthermore, pressing this button will load the saved atom coordinates in the local "atoms_tmp.db" file. Hence, to run several simultaneous DM searches with the same parameters, the parameters need to be set up only once. Once setup, pressing the *write new params* will create a parameter file. This file needs to be placed in the directory of every simultaneous search. Finally, the *read saved params* button is pressed in every simultaneous copy of NENA running.

The *view iterate* button opens a Rasmol molecular viewer window, and displays the iterate. It should be noted that the DM iterate frequently does not have a valid peptide geometry, and so the molecule displayed by Rasmol often has some peculiar bonds displayed. The *view iterate* button is really only appropriate after an atomic refinement has been done (this is done with the *refine atoms* button). On the other hand, it is often interesting to view the DM iterate itself.

The *randomize atoms* button adds a random vector of magnitude **rand_mag** to each atom. If **rand_mag** is around 1000, then this button produces a very random initial atomic configuration for the DM to iterate.

The *refine atoms* button executes **refine_iters** alternating projections. The alternating projections algorithm is essentially Fienup's error reduction algorithm,[18] which is explained in chapter 2. This algorithm finds the nearest local minimum in the energy landscape for the current protein conformation. It minimizes the non-bonded energy similar to a very cold Monte Carlo simulation. The second projection of the alternating projection is a projection to the geometry constraint, and hence the output of this atomic refinement has the correct peptide geometry. As such, it is illustrative to view the iterate with the *view iterate* button after refining the atomic coordinates.

The *chain project* button executes a projection of the current atomic configuration to the geometry constraint.[3] The *chain project* button will minimize the penalty function described in appendix A until it is less than **chain_min_nrg**. During this minimization, **chain_proj_iter_max** is how many minimization iterations the minimizing algorithm is allowed to maximally take, **chain_min_step_size** is the shortest step the minimizer can take, and **chain_max_step_size** is the largest step the minimizer can take. A maximum step size of 1.0 yields faster projections, but less "distance minimizing".[4] A maximum

[3]See appendix A for a complete description of the geometry constraint.

[4]See chapter 2.1.2 for details on projections.

step size of 0.1 causes slower projections, but the DM seems to converge more quickly. The result of the projection to the geometry constraint is an atomic configuration with a valid peptide geometry, but it is quite likely that atoms will be overlapping; there is no steric repulsion in effect for this projection. Pressing the *view iterate* button after this projection will often confuse Rasmol, and Rasmol will draw many bonds in the protein that shouldn't be there.

The *potential project* button executes a projection of the current atomic configuration to the energy constraint.[5] This button minimally moves the atoms so that the non-bonded energy is less than **pot_min_nrg**. The projection is accomplished by minimizing the non-bonded energy until the energy is below the target. During the minimization, **pot_proj_iter_max** iterations are maximally allowed, **pot_max_step_size** is the largest minimizing step allowed, and **pot_min_step_size** is the shortest step allowed. If the maximum step size for this minimization, **pot_max_step_size**, is too large (1.0 for example), then the resultant configuration could have an energy slightly lower than the target, since the last step taken by the minimizing routine may overshoot the energy target. A maximum step size of 0.1 typically produces an atomic configuration with an energy very close to **pot_min_nrg**. The DM seems to converge faster with a small step size.

The final button, *DM iter*, begins the DM iteration. NENA will perform **dm_iters** iterations, outputting its progress to the screen every 10 iterations. The DM will use the parameter **beta** as the β defined in chapter 2. If the DM error[6] goes below **min_dm_err**, then NENA executes the *refine atoms* script described above. After the refinement, NENA outputs the refined atomic configuration as a PDB file in the "outputs" folder in the local directory. This output configuration necessarily has a non-bonded energy below **pot_min_nrg**, and so this new low energy becomes the new target energy for

[5] See appendix A for a complete description of the energy constraint.

[6] See chapter 2 for the definition of the DM error.

the potential projection. **pot_min_nrg** is automatically updated, and the DM begins searching again from a random atomic configuration. The random atomic configuration is the result of NENA executing the *randomize atoms* script described above.

As the DM searches for low energy configurations, every **refinenotfor** iterations the DM executes the *refine atoms* script described above. If the resultant protein conformation has an energy below **pot_min_nrg**, then NENA outputs the atomic configuration as a PDB file to the "outputs" folder. NENA then replaces **pot_min_nrg** with the new lower energy, and begins searching again from a random atomic configuration. On the other hand, if after performing **refine_iters** alternating projections, the resultant protein conformation energy is not below **pot_min_nrg**, then the DM continues searching from the refined atomic configuration. Additionally, the DM search automatically replaces the iterate with a random atomic configuration (essentially restarting) every **restartevery** iterations. Finally, the iterate of the DM can be automatically output to the "outputs" directory in the form of a PDB file every **outputprogevery** iterations.

The DM iteration can be interrupted by pressing control C in the terminal NENA was started in. Once interrupted, all the buttons have the above effects, and DM iteration can be resumed by pressing the *DM iter* button.

An example of NENA conducting the DM search can be seen in figure 4.3.

The progress of the DM iterate's Ramachandran angles can be seen as vertical colored stripes, and the DM error can be seen on the right. The iterate evolution begins at the bottom and progresses upwards. There is a two color vertical column for every amino acid (one color each for ϕ and ψ). The color key for ϕ and ψ is shown in figure 4.4.

The DM error displayed to the right of the Ramachandran data is a sideways graph of the DM error versus iteration. As can be seen in the figure, the DM error goes to

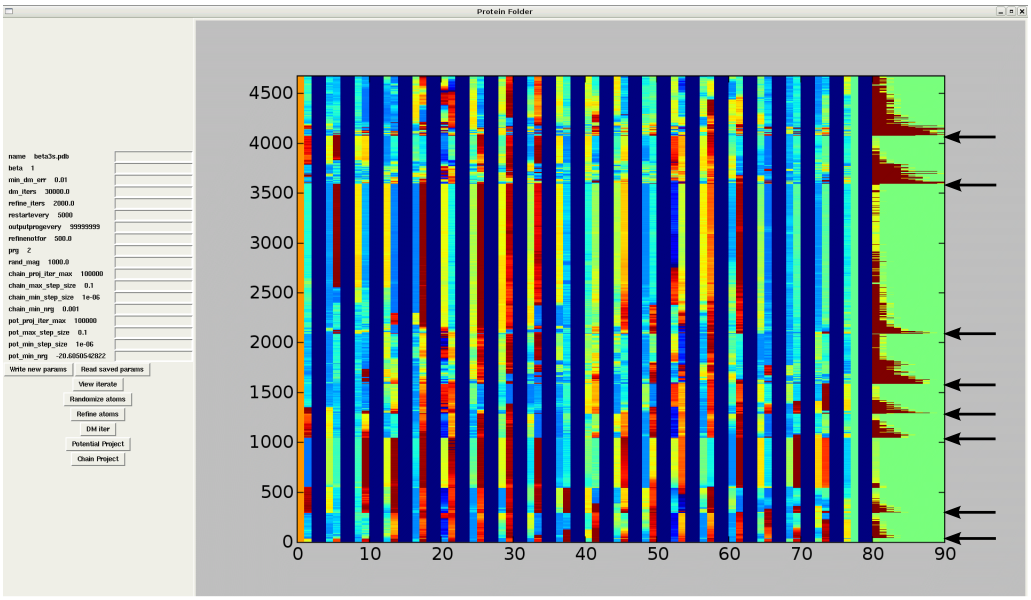

Figure 4.3: Here NENA attempts to minimize the non-bonded energy of a 20 amino acid long protein. Each of the two colored vertical bars (separated by dark blue bars) represents one amino acid. For the two colored bars, the left color encodes information regarding ϕ, and the right color shows ψ. As the DM searches, the Ramachandran angles of the iterate are displayed with these bars. The DM error (defined in chapter 2) is plotted (in red) to the right of the vertical bars. The arrows point to iterations where the DM error went to zero, and a fixed point of the DM was found. In such cases, the target energy was lowered, and the DM search was restarted with a random iterate. Just after a restart, the DM error can be seen to be very large.

zero at iteration 30, 290, 1100, etc.. Every time the DM error reached zero, a new low energy protein conformation is found. The conformation's atomic coordinates are then output into a PDB file, the target energy **pot_min_nrg** is lowered, and the DM search is restarted with a new random atomic configuration.

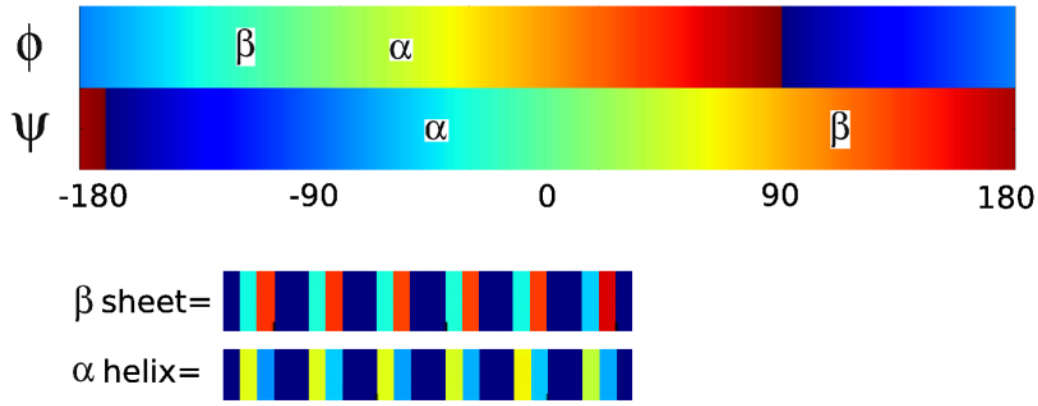

Figure 4.4: The top two color bars show the color scales for ϕ and ψ. On each color bar, the angles where alpha helices and beta sheets form are indicated. At the bottom of the figure, an example of a sequence of amino acids forming a beta sheet and an alpha helix is shown.

4.2 Principles behind NENA

The search algorithm operating behind the python GUI is the difference map (DM). The DM seeks to find an atomic configuration simultaneously in two constraint sets. The first constraint set is the set of atomic configurations that have the correct peptide geometry. The second is the set of atomic configurations that have a non-bonded energy below a given target energy. In NENA, this target energy is **pot_min_nrg**.

Details on how the DM searches for a simultaneous member of two constraint set are given extensively in chapter 2. For details on the specifics of the constraint set, refer to appendix A.

4.3 Constraints

The constraint set are exhaustively explained in appendix A, but how they are computationally implemented in NENA is explained here.

The atomic configuration is represented in C as an array of "atom" structures. The atom structure is defined as,

```
typedef struct {
  int aminoacidnum;    //which amino acid the atom belongs to
  fvector r;           //the 3D coordinate of the atom
  char *atomtype;      //C, N, O, H, or S
  char *genpotcode;    //atom type in Elber and Tobi 2000
  char *pdbcode;       //CA, CB, N, NE1, ...-- PDB atom codes
  char *aminoacidtype; //what type of amino acid is the atom in?
  int hydro;           //hydrophobicity code, used in HP energy
  double rscl;         //atom mass
  double radius;       //atomic radius
  double charge;       //atomic charge
  int num_nghbrs;      //how many neighbors does the atom have?
  int *nghbrs;         //list of the neighbors
  int num_bonded;    //how many atoms are a fixed distance away?
  int *bonded;       //list of atoms that are a fixed dist. away
} atom;
```

Clearly, each atom has a lot of information associated with it. The structure member *genpotcode* is a string that comes from a paper[11] that developed a useful interatomic potential. Though the potential is no longer used by NENA, this structure member is still useful in that it distinguishes, for example, a hydroxyl oxygen from a carboxyl oxygen.

4.3.1 The geometry constraint

The geometry constraint is determined by four lists of sub-constraints. For example, one list is a C array of bonds present in the native protein. These lists are comprised of different types of C structures, and each structure represents one constraint in the protein. As described below, there is a C structure to represent a bond, a C structure

to represent a bond angle, a structure to control four atom parallelepiped volumes (this distinguished left handed versus right handed orientations), and a structure to control the Ramachandran angles.[7]

The projection to the geometry constraint makes every sub-constraint in each list approximately satisfied. When every element of the four lists of sub-constraints is satisfied, the total penalty becomes approximately zero.

Constraint lists

The first list is a C array of "bond" structures. These structures represent an atomic bond in the protein polypeptide. There are approximately as many bond structures in the bond list as there are atoms in the protein. The bond structure is defined as,

```
typedef struct {
    int atom1;    //atom1 # in the atom list
    int atom2;    //atom2 # in the atom list
    double tar;   //how far apart the atoms SHOULD be
    double act;   //how far apart the atoms actually ARE
    char *type;   //what kind of bond?  backbone? sidegroup?
    double er;    //the penalty function error of this bond
    double ewt;   //the penalty function weight of this bond
} bond;
```

When the projection to the geometry constraint is complete, each bond structure in the array will have the members *tar* and *act* approximately equal, and the member *er* will be approximately zero. In equation (A.1) of appendix A, the variable $B_i = p_i \left(\vec{\mathbf{v}}_i \cdot \vec{\mathbf{v}}_i - b_i^2\right)^2$ can be identified here as the *er* member of the bond structure, p_i is here called *ewt*, and b_i is *tar*.

The second list of sub-constraints is a C array of "angl" structures. These structures represent a bond angle constraint in the protein. There are approximately two angle

[7] See appendix A for how each of these lists of structures contribute to the total chain penalty function.

constraints per atom in the protein. A bond angle is defined by the structure,

```
typedef struct {
    int atom1;   //atom1 # in the atom list
    int atom2;   //atom2 # in the atom list
    int atom3;   //atom3 # in the atom list
    double tar;  //What (a3-a2).(a2-a1) should be
    double act;  //What (a3-a2).(a2-a1) really is
    char *type;  //where is this angle? backbone? sidegroup?
    double er;   //penalty function error of this angle
    double ewt;  //penalty function weight of this angle
} angl;
```

Notice it takes three atoms to define an angle. These three atoms define two vectors, and it is the dot product of these vectors that is controlled with each angle constraint. The magnitudes of the two vectors is controlled by the bond constraint described above. In equation (A.1) of appendix A, the variable $A_i = p_i \left(\vec{\mathbf{v}}_{i,1} \cdot \vec{\mathbf{v}}_{i,2} - a_i\right)^2$ can be understood to be the *er* member of this structure. Furthermore, p_i is here called *ewt*, and the variable a_i is called here *tar*.

The third list of sub-constraints is a C array of "det" structures. These structures each have four atoms. These four atoms determine three vectors, which in turn determine a parallelepiped volume. There are roughly two of these constraints per amino acid: one for the chiral C_α, and one for the planar peptide bond. The "det" structure is defined as,

```
typedef struct {
  int atom1;   //atom1 # in the atom list
  int atom2;   //atom2 # in the atom list
  int atom3;   //atom2 # in the atom list
  int atom4;   //atom2 # in the atom list
  double tar;  //the target parallelepiped volume
  double act;  //the actual parallelepiped volume
  char *type;  //what kind of determinant? omega? CA?
  double er;   //penalty function error of this determinant
  double ewt;  //penalty function weight of this determinant
} det;
```

In the "det" structure, the *type* member is a string specifying if, for example, the specific determinant controls the four atoms defining the ω dihedral angle, or the four atoms

defining the chirality of the C_α atom. Both the D_i and the Ω_i terms of equation (A.1) of appendix A refer to a "det" structure.

The final list of sub-constraints is a C array of "rama" structures. These structures control the Ramachandran angles. For this reason each structure contains information about the 5 backbone atoms determining each Ramachandran (ϕ, ψ) pair. The "rama" structure is defined by,

```
typedef struct {
  int atom1;        //atom1 # in the atom list
  int atom2;        //atom2 # in the atom list
  int atom3;        //atom3 # in the atom list
  int atom4;        //atom4 # in the atom list
  int atom5;        //atom5 # in the atom list
  int aminoacidnum; //Which amino acid is this?
  char *type;       //first? last? proline? glycine?
  double phi;       //the value of phi
  double psi;       //the value of psi
  double sinphi;    //the value of sin(phi)
  double cosphi;    //the value of cos(phi)
  double Aphi;      //see description below
  double sinpsi;    //the value of sin(psi)
  double cospsi;    //the value of cos(psi)
  double Apsi;      //see description below
  double score;     //alpha helix like? beta sheet like?
  double er;        //penalty function error
} rama;
```

The precise penalty function controlling the allowed Ramachandran angles is given in appendix A. Basically, there are disallowed combinations of Ramachandran (ϕ, ψ) pairs, and the penalty function is calculated based on how much a current (ϕ, ψ) pair violates the disallowed region of the Ramachandran plot. The disallowed region of the Ramachandran plot is shown in figure 1.6

The *Aphi* and *Apsi* members of the "rama" structure record the values of two frequently calculated vector functions used in the calculation of the Ramachandran penalty function. The values are recorded so they only have be calculated once, rather than

multiple times.[8]

Each of the elements of these four lists represents a sub-constraint, such as a bond length, or an ω dihedral angle being 0°. The minimization routine described below minimizes all of these sub-constraints simultaneously. The minimization routine exits, and the projection is complete, when the sum of all of the *er* members is less than the NENA parameter **chain_min_nrg**.

Ramachandran guiding function

To encourage the formation of secondary structure, there is an additional guiding function gently effecting the atoms as the above penalty functions are minimized. The use of a guiding function is described in Elser and Rankenburg's 2006 application of the DM to a very simple model of proteins.[14]

The guiding function considers each of the amino acids sequentially, and determines if the amino acid should be an α helix or a β sheet. This decision is based on an average of the amino acid, and each of its two neighbor amino acids. For example, if both neighbors of an amino acid each have a Ramachandran (ϕ, ψ) pair precisely in the α helix region, then the center amino acid will be gently pushed toward the α helix region as well. The magnitude of this "gentle" push is approximately 10 times weaker than any of the bond, angle, determinant, or Ramachandran constraints listed above.[9]

Guiding functions are given this name because the atoms are guided while minimizing the above penalty functions. It is important to realize the geometry constraint is unchanged: it is still defined as having all the bond constraints, angle constraints, and

[8] *Aphi* and *Apsi* are used in the *ramaengrad* function inside "enrg_grad.c". See appendix C.

[9] The code for this guiding function can be found in the *ramaengrad* function inside "enrg_grad.c" in the labeled section "sequence continuity guiding function". See appendix C.

determinant constraints satisfied, and the Ramachandran angles are all in the allowed region of the Ramachandran plot. While minimizing the various penalty functions, the amino acids are concurrently encouraged to form secondary structure. In other words, the guiding function does not effect the quitting criteria of the geometry projection, but it does guide the Ramachandran (ϕ, ψ) pairs toward either α helices or β sheets while the actual penalty functions described above are minimized.

4.3.2 The energy constraint

The energy constraint is defined as an atomic configuration with an energy below the NENA parameter **pot_min_nrg**. The total non-bonded energy comes from hydrogen bonds between backbone hydrogens and backbone oxygens (only if they have at least 2 amino acids between them), hydrophobic interactions, and steric repulsion. To implement steric repulsion, every atom has a radius associated with it, stored in the *radius* member of the atom structure. If two atoms become too close together, they contribute a positive energy to the total non-bonded energy, and hence they repel each other. Every atom also has a hydrophobicity associated with it, stored in the *hydro* member of the atom structure. In general, hydrophilic atoms repel every atom type, while hydrophobic atoms attract each other to varying degrees.

The energy calculation is ostensibly done by calculating an interaction energy between every pair of atoms.[10] However, this naive approach is facilitated with a continually updated "neighbor list". The *nghbrs* member of an atom structure is a list of atoms that are close to the given atom. For calculating pairwise interaction energies, if suffices to consider only atoms that are close together, since atoms too distant don't contribute significant energies.

[10] See appendix A for a precise description of the interaction energy function.

Neighbor list

During the minimization of the non-bonded energy, the neighbor list is recalculated every ten minimization iterations.[11] The neighbor calculating procedure scales as the square of the number of atoms, and so is only recalculated as often as necessary. In general, if two atoms are within 7Å, they get placed on each other's neighbor lists. However, there are exceptions. If two atoms are constrained to be a fixed distance apart (such as two atoms bonded together), they shouldn't contribute to the atomic configuration's total non-bonded interaction energy. For this reason they are not placed on each other's neighbor lists. For a given atom, the atoms not eligible to be considered as an interaction neighbor are listed in the *bonded* member of the atom structure.

4.4 Projections

The two projections are performed by minimizing, for the potential constraint and the geometry constraint, the non-bonded energy and the total penalty function respectively. For the minimization routine described below, it doesn't matter what is being minimized. Let an atomic configuration be called $\vec{\mathbf{R}} = \{\vec{\mathbf{r}}_1,\ \vec{\mathbf{r}}_2,\ \ldots\}$, where $\vec{\mathbf{r}}_i$ is the 3D coordinate of atom i. All the minimizing routine needs is a function $f(\vec{\mathbf{R}})$ that returns a value to be minimized, and the gradient of the function, $\nabla_{\vec{\mathbf{r}}_i} f(\vec{\mathbf{R}})$.

In "enrg_grad.c", the function *chainengrad* returns the total penalty function, and fills in the passed gradient array at the same time. The function *potengrad* returns the total non-bonded energy, and also fills in the passed gradient array.

The minimization routine essentially follows the gradient direction, taking intelligently calculated step sizes. Note that the downhill gradient direction is the direction

[11] The function to calculate neighbor lists is the *nghbrmaker* function inside "enrg_grad.c". See appendix C for this code.

that decreases the function the fastest. For this reason, if the atomic configuration is very close to the constraint set, then following the downhill gradient will create the minimal change to the atomic configuration yielding an element of the constraint set. Thus, if the iterate is close enough to a constraint set, this minimization routine performs a true projection.

4.4.1 The minimizing routine

The minimization routine uses three pieces of information to calculate the step size to take. First, it calculates the current function value, and at the same time calculates the gradient direction and magnitude. Then it moves the atoms a small test step in the gradient direction, and calculates the new energy. With these two energy values, and the magnitude of the gradient, the minimization routine assumes a parabolic energy profile in the direction of the gradient and calculates how far to step in the downhill gradient direction..

Given the initial energy E_0, the energy E' after a test step d_0, and the magnitude of the gradient M, the minimum (or maximum) of the parabola is calculated to be a forward distance of,

$$d_{\text{step}} = \frac{\frac{d_0}{2}}{1 - \frac{E_0 - E'}{M\, d_0}} \, .$$

Note if $\frac{E_0 - E'}{M\, d_0} < 1$, then the parabola is concave up, and the minimum is found by the given formula. If $\frac{E_0 - E'}{M\, d_0} > 1$, then the parabola is concave down, and a step size forward of $1.5\ d_0$ is taken.[12] The step size taken is then used for the next iterations test step. This calculation is shown in figure 4.5.

[12] If the parabola is concave down, then gradually increasing the step size is warranted. Hence, the step size gets 50% larger every iteration.

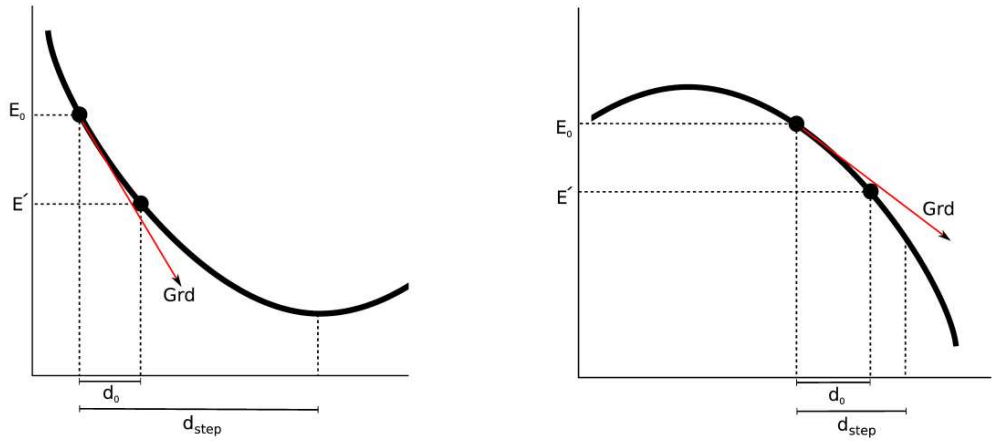

Figure 4.5: The function minimizer in NENA assumes a parabolic function profile in the direction of the gradient. The minimizer evaluates the energy and the gradient at the current spot. It takes a test step in the direction of the downhill gradient of magnitude d_0. It evaluates the energy at this new spot. Based on these two energies, and the initial magnitude of the gradient, the minimizer calculates where the minimum (or maximum) of the parabola is. If the parabola is concave up, a step of size d_{step} is taken in the downhill gradient direction. If the parabola is concave down, a step of size $1.5\ d_0$ is taken in the downhill gradient direction. The size of the taken step is then used as the test step size d_0 for the next minimizer iteration.

Appendix A

The two protein constraints

A.1 Geometry constraint

The geometry constraint ensures all the bond lengths and angles are correct (data from Engh 1991[16]), all the peptide bonds are in the trans orientation, and the Ramachandran angles are not in the sterically forbidden region of the Ramachandran plot. An atomic configuration satisfying these conditions is a valid protein conformation (rotamer). We use the following penalty function to implement the geometry constraint:

$$0 = \sum_{i \in \text{bonds}} B_i + \sum_{i \in \text{angles}} A_i + \sum_{i \in \text{peptide bond}} \Omega_i + \sum_{i \in \text{dets}} D_i + \sum_{i \in \text{ramas}} R_i \qquad \text{(A.1)}$$

For the first term of equation (A.1), the index i runs over all the bonds in the protein. For each bond, $B_i = p_i \left(\vec{\mathbf{v}}_i \cdot \vec{\mathbf{v}}_i - b_i^2\right)^2$. Each bond has a target length, b_i, a penalty weight p_i, and $\vec{\mathbf{v}}_i$ is the vector connecting the two atoms participating in the bond. Essentially, each B_i is a measure of how correct the i^{th} bond is, and p_i is the relative cost of the i^{th} bond deviating from correct. For all backbone bonds (such as C_α-C, C-N, or N-C_α) p_i is 4, for the bonds coming directly off the backbone (such as C-O, N-H, or C_α-C_β) p_i is 2, and the bonds within a sidegroup are given a penalty weight of 1.

In the second term of equation (A.1), the index i runs over all the angles in the protein. The i^{th} angle is defined by two vectors, $\vec{\mathbf{v}}_{i,1}$ and $\vec{\mathbf{v}}_{i,2}$. For each angle, $A_i = p_i \left(\vec{\mathbf{v}}_{i,1} \cdot \vec{\mathbf{v}}_{i,2} - a_i\right)^2$. Every angle has a target dot product for the two vectors, a_i, and a penalty weight p_i. Note that the magnitudes of $\vec{\mathbf{v}}_{i,1}$ and $\vec{\mathbf{v}}_{i,2}$ are each controlled by the bond constraint above. Like B_i above, A_i is a measure of how correct the i^{th} angle is, and p_i is the relative cost of the i^{th} angle deviating from correct. For all backbone angles (such as C_α-C-N, C-N-C_α, N-C_α-C) p_i is 2, for angles involving bonds directly off the

background (such as O-C-C_α, O-C-N, H-N-C_α, H-N-C, C_β-C_α-C, and C_β-C_α-N) p_i is 1, and for all angles within a sidegroup p_i is 0.5.

For the third term of equation (A.1), the index i runs over backbone peptide bonds. These Ω_i terms ensure that backbone hydrogens and oxygens are in the trans configuration, and that the atoms H, N, C, and O all lie in a plane. Figure A.1 shows the correct trans orientation of the atoms participating in a peptide bond, and will aid in visualizing the vectors described below. To ensure the trans configuration, terms $\Omega_i = p_i \left(\vec{\mathbf{v}}_{i,1} \cdot \vec{\mathbf{v}}_{i,3} - d_0\right)^2$ are included. Here, $\vec{\mathbf{v}}_{i,1}$ is the O-C vector for the i^{th} peptide bond, $\vec{\mathbf{v}}_{i,3}$ is the N-H vector for the i^{th} peptide bond, and d_0 is the correct dot product of these two vectors. The penalty weight p_i is 1.5. To ensure the atoms H, N, C, and O all lie in a plane, terms $\Omega_i = p_i \left(\vec{\mathbf{v}}_{i,1} \cdot (\vec{\mathbf{v}}_{i,2} \times \vec{\mathbf{v}}_{i,3})\right)^2$ are included. Here, $\vec{\mathbf{v}}_{i,1}$ and $\vec{\mathbf{v}}_{i,3}$ have the same meaning as before, $\vec{\mathbf{v}}_{i,2}$ is the C-N peptide bond vector, and the penalty weight p_i is 0.2 . To ensure the trans configuration of consecutive C_α's, and to ensure that the four atoms C_α, C, N, and C_α all lie in a plane, there are identical constraints involving these four atoms.

The fourth term of equation (A.1) is for four-atom configurations where left-handed versus right-handed orientations are relevant. For every amino acid (except glycine) the four atoms C_β, C_α, C, and N define a parallelepiped, whose volume is constrained by $D_i = p_i \left(\vec{\mathbf{v}}_{i,1} \cdot (\vec{\mathbf{v}}_{i,2} \times \vec{\mathbf{v}}_{i,3}) - V_0\right)^2$, where $\vec{\mathbf{v}}_{i,1}$ is the C_β-C_α vector, $\vec{\mathbf{v}}_{i,2}$ is the C_α-C vector , $\vec{\mathbf{v}}_{i,3}$ is the C_α-N vector, V_0 is the target parallelepiped volume (note the sign of V_0 dictates left handed versus right handed orientations), and the penalty weight p_i is 1. Also, for sidegroups that have a left handed versus right handed preference, such as the side group of isoleucine, there is an identical constraint relating the orientation of the relevant atoms.

The last term of equation (A.1) controls the range of the Ramachandran angles ϕ and

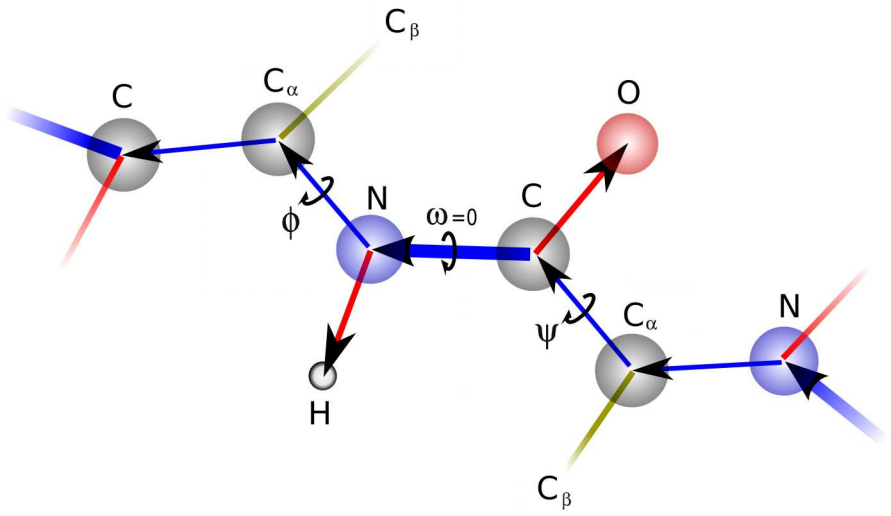

Figure A.1: This figure is the same as figure 1.5. It is reproduced here to aid in visualizing the vectors used in equation (A.1). The atoms H, N, C, and O all lie in a plane. The H and O are in the trans configuration.

ψ. Due to steric repulsion, there is a significant region of the Ramachandran plot that is inaccessible to proteins, shown in figure A.2. Rather than calculating ϕ, it is easier to calculate the sine and the cosine of ϕ by:

$$\sin\phi = \vec{\mathbf{v}}_2 \cdot (\vec{\mathbf{v}}_3 \times \vec{\mathbf{v}}_1) \, \frac{|\vec{\mathbf{v}}_2|}{|\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2| \, |\vec{\mathbf{v}}_2 \times \vec{\mathbf{v}}_3|}$$

$$\cos\phi = [(\vec{\mathbf{v}}_1 \cdot \vec{\mathbf{v}}_2) \, (\vec{\mathbf{v}}_2 \cdot \vec{\mathbf{v}}_3) - (\vec{\mathbf{v}}_1 \cdot \vec{\mathbf{v}}_3) \, (\vec{\mathbf{v}}_2 \cdot \vec{\mathbf{v}}_2)] \, \frac{1}{|\vec{\mathbf{v}}_1 \times \vec{\mathbf{v}}_2| \, |\vec{\mathbf{v}}_2 \times \vec{\mathbf{v}}_3|}$$

where $\vec{\mathbf{v}}_1$, $\vec{\mathbf{v}}_2$, and $\vec{\mathbf{v}}_3$ are the three vectors defining the torsion angle ϕ. Specifically, $\vec{\mathbf{v}}_1$ is the N-C vector, $\vec{\mathbf{v}}_2$ is the C_α-N vector, and $\vec{\mathbf{v}}_3$ is the C-C_α vector. Referring to figure A.1 will help visualizing the vector definitions. Similarly, the sine and the cosine of ψ are calculated in the same way, except $\vec{\mathbf{v}}_1$, $\vec{\mathbf{v}}_2$, and $\vec{\mathbf{v}}_3$ are the three vectors defining the torsion angle ψ, or $\vec{\mathbf{v}}_1$ is the C_α-N vector, $\vec{\mathbf{v}}_2$ is the C-C_α vector, and $\vec{\mathbf{v}}_3$ is the N-C vector.

In terms of ϕ and ψ, the R_i in equation (A.1) is the sum of a function controlling the ϕ distribution and the ψ distribution. Specifically,

$$
\begin{aligned}
R_i &= \begin{cases} 0 & \text{if} \quad f(\phi) < -0.26 \\ 0.3\,(f(\phi)+0.26)^2 & \text{if} \quad f(\phi) > -0.26 \end{cases} \\
&+ \begin{cases} 0 & \text{if} \quad g(\psi) > -0.50 \\ 6.0\,(g(\psi)+0.50)^2 & \text{if} \quad g(\psi) < -0.50 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\text{where} \quad f(\phi) &= 0.83\ \sin\phi + 0.17\ \cos\phi \\
\text{and} \quad g(\psi) &= 0.56\ \sin\psi + 0.44\ \cos\psi\ .
\end{aligned}
$$

R_i is plotted in the black shading in figure A.2. This function crudely, though sufficiently, approximates the region of the Ramachandran plot inaccessible due to steric repulsion, also shown in figure A.2.

A member of the geometry constraint has each of the penalty functions above equal to zero. For the projection to this constraint space, all five terms of equation (A.1) are minimized by an adaptive step-size steepest descent algorithm, and are considered sufficiently close to zero when the total penalty is less than 0.001 per amino acid. The various energy weights p_i used above were chosen such that this minimization is computed efficiently, and never frustrated. After the minimization, all of the bonds have the proper length, all of the angles are correct, all of the peptide bonds are planar and in the trans configuration, and all of the Ramachandran angles are in the allowed region of figure A.2. However, an atomic configuration satisfying the geometry constraint may have non-bonded atoms overlapping. It is this fact that atoms are allowed to pass through each other that makes the minimization of the penalty function always successful, and never frustrated.

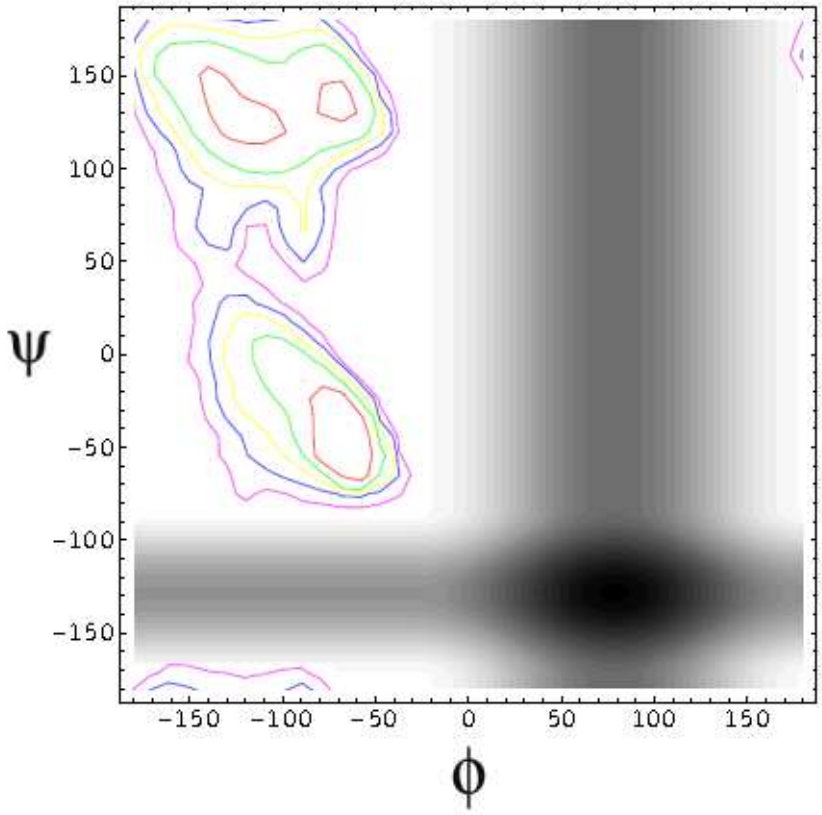

Figure A.2: Also shown is a contour plot of the Ramachandran energy used in equation (A.1). The Ramachandran distribution data is taken from Kleywegt 1996[28]. 98% of all non-glycine Ramachandran angles lie in the purple contour, 95% (blue), 90% (yellow), 80% (green) and 50% (red). The black region is the value of R_i in equation (A.1).

A.2 Energy constraint

The energy constraint is defined as the set of all atomic configurations with a non-bonded energy below a given target, E_0. The constraint space is thus dependent on the energy target E_0. An atomic configuration satisfying this condition does not necessarily have a valid peptide geometry, indeed bonded atoms may be quite separated, and the atomic configuration may not even resemble a polypeptide. In detail, this constraint is deter-

mined as follows:

$$E_0 > E_{NB} = \sum_{i,j \in \text{atoms}} \text{VE}_{ij} + \sum_{i,j \in \text{atoms}} \text{HP}_{ij} + \sum_{i \in H,\, j \in O} \text{HB}_{ij} \qquad \text{(A.2)}$$

The first term of equation (A.2) prevents atoms from overlapping. Specifically,

$$\text{VE}_{ij} = \begin{cases} 0 & \text{if} \quad r_{ij} > r_0 \\ \left(1 - \left(\frac{r_{ij}}{r_0}\right)^2\right)^2 & \text{if} \quad r_{ij} < r_0 \end{cases}$$

where r_{ij} is the distance between the i^{th} and j^{th} atoms, and r_0 is the distance at which the atoms start overlapping. The r_0 used is a sum of the atomic van der Waals radii of the the i^{th} and j^{th} atoms. The following radii are used: 1.57Å for aliphatic sidegroup carbon, 1.41Å for aromatic sidegroup carbon, 1.44Å for backbone carbon, 1.34Å for nitrogens, 1.20Å for oxygens, 0.65Å for hydrogens, and 1.57Å for sulfur. These radii are based on data from a previous study[32], and have been adapted to our VE_{ij} functional form. Note VE_{ij} goes to 1 at an atomic separation of $r_{ij} = 0$, and smoothly goes to zero at $r_{ij} = r_0$. Atoms that are bonded together must be treated specially, since they must be allowed to come closer together than non-bonded atoms. For these pairs, r_0 is 40% of the sum of constituent atomic van der Waals radii.

The second term of equation (A.2) simulates the entropic interaction of water with various sidegroup atoms. Two atoms only interact via this hydrophobic energy if they are both sidegroup atoms (C_β, C_γ, etc.), and they belong to different amino acids. The functional form of HP_{ij} is,

$$\text{HP}_{ij} = \begin{cases} E_{ij}\left(2\left(\frac{r_0}{r_{ij}}\right)^2 - \left(\frac{r_0}{r_{ij}}\right)^4\right) & \text{if} \quad r_{ij} > r_0 \\ E_{ij} & \text{if} \quad r_{ij} < r_0 \end{cases}$$

where r_0 is calculated the same as above, and E_{ij} is the interaction energy depending on the participating atom types. The energies used are shown in table A.1. Since hydrophobic atoms, in this model, attract each other, they tend to form a well defined oily

Table A.1: A negative number means the two atom types attract each other, positive numbers indicate repulsion. These parameters are based on those found in a previous study[32], and have been adapted to our functional form of HP_{ij}.

	Aliphatic Carbon	Aromatic Carbon	Polar	Sulfur
Aliphatic Carbon	-0.108	-0.075	0.072	-0.063
Aromatic Carbon	-0.075	-0.081	0.093	-0.048
Polar	0.072	0.093	0.126	0.084
Sulfur	-0.063	-0.048	0.084	-0.126

core, while polar atoms repel every atom type, and thereby tend to inhabit the surface of the protein.

The final energy of equation (A.2) represents hydrogen bonding. The indices i and j run over all the backbone hydrogen atoms and all the backbone oxygen atoms respectively. The functional form of HB_{ij} is the product of a distance dependent function $f(r)$ and a function of the two angles $g(\theta_a\ ,\theta_b)$ formed by the C-O-H angle θ_a, and the O-H-N angle θ_b. These definitions for θ_a and θ_b are shown in figure A.3. In terms of the distance function $f(r)$ and the angular function $g(\theta_a\ ,\theta_b)$, HB_{ij} is,

$$\text{HB}_{ij} = f(r_{ij})\ g(\theta_a\ ,\theta_b)$$

$$f(r_{ij}) = \begin{cases} -1.5\left(2\left(\frac{r_0}{r_{ij}}\right)^2 - \left(\frac{r_0}{r_{ij}}\right)^4\right) & \text{if} \quad r_{ij} > r_0 \\ -1.5 & \text{if} \quad r_{ij} < r_0 \end{cases}$$

$$g(\theta_a\ ,\theta_b) = \begin{cases} \cos^2\theta_a \cos^2\theta_b & \text{if} \quad \cos\theta_a > 0 \text{ and } \cos\theta_b > 0 \\ 0 & \text{if} \quad \cos\theta_a < 0 \text{ or } \cos\theta_b < 0 \end{cases}$$

The target hydrogen bond distance, r_0, is 1.9Å, and the distance between the i^{th} hydrogen and the j^{th} oxygen is r_{ij}. Also, two backbone atoms can form hydrogen bonds only

if they are separated by at least two other amino acids.

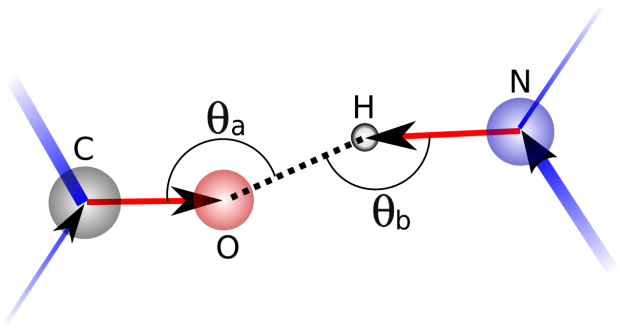

Figure A.3: θ_a and θ_b are both used in the calculation of the hydrogen bond energy. The dotted line is the hydrogen bond. The hydrogen bond energy is a product of a distance dependent function and an angular function.

The three energies in equation (A.2), in total, constitute the non-bonded energy of an atomic configuration. The energy constraint space is defined as the set of atomic configurations whose non-bonded energy is less than a predefined target energy, E_0. The projection to this constraint space is done by minimizing the total energy with an adaptive step-size steepest descent algorithm until the total energy is less than or equal to E_0.

Note finally that an atomic configuration whose non-bonded energy is less than E_0 (and therefore a member of this constraint space) does not necessarily have a valid peptide geometry. For example, it is possible to have two bonded atoms separated by large distances, and the atomic configuration can still be a member of this constraint space. The energy functions above treat the atomic configuration as a collection of independent atoms, rather than a linked chain.

Appendix B

DM explorer program

Recently a convenient program was created to demonstrate the search dynamics of the difference map (DM) in 2D.[1] With this program, the two constraint spaces used by the DM are represented by a blue and red contour, drawn in the plane. Every pixel in the field of view is iterated, and the number of iterations, for every pixel, is recorded. The field of view is then rendered in an RGB scale to indicate which pixels converged fast (dark blue colors) and which pixels search for a long time before finding a fixed point (dark red colors). There is a large variety of constraint geometries that can be explored, and various DM β's can be tested.[2]

The main computation of the DM iteration is done in C, and a convenient user interface was created in Python. The Python graphical user interface (GUI) calls precompiled C routines via an interpreting language, Pyrex.

Understanding the interaction of these three languages is fundamental for understanding the protein folding software package, NENA (the subject of chapter 4). The main structure of the DM explorer program (DMX) is the same as that of NENA, though the details are much simpler. Therefore it is recommended that before attempting to modify NENA, first the reader should understand thoroughly how the DMX program works.

When creating the DMX program, there was a dearth of available examples for using C in Python. For this reason, the code will be reproduced and explained here in detail. In the chapter devoted to NENA (see chapter 4), the NENA code is not reproduced. The DMX code below is meant to aid in understanding the details of the NENA code. The

[1] For an explanation of the difference map, see chapter 2.

[2] See chapter 2for the definition of β.

full NENA code, with additional comments, is included in its entirety in Appendix C.

In the description below, variables and classes will be referred to with **bold** font, while lines from the code will be referred to with italics.

B.1 Functionality

A picture of the various text boxes and the main display area is shown in figure B.1.

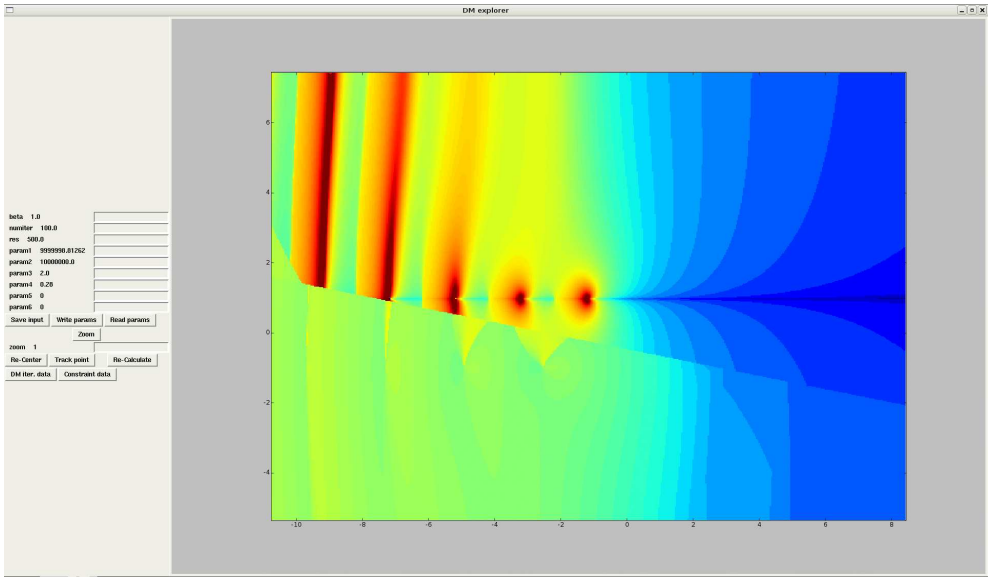

Figure B.1: On the left are the various DM parameters and text boxes. On the right is the plotting area.

The text boxes on the left allow the user to input DM parameters. The **beta** text box is for the β in the DM.[3] The **numiter** text box is to specify how many iterations (maximum) a pixel is to be iterated before the DM gives up and begins calculating the next pixel. If this is small (about 100) then the total calculation of the entire field of view is fast. However, if around 1000 iterations are allowed, some particularly difficult

[3]See chapter 2 for details on the meaning of β.

pixels will converge.[4] If this parameter is too low, the calculated picture will have large regions of uniform color, where each pixel failed to converge. The **res** text box specifies the picture resolution. Resolutions around 100 will yield a quickly calculated picture, but course features. Resolutions around 600 yield very fine detail, but take a long time to calculate.

The various **param** text boxes are for inputing constraint space parameters. Currently, param5 and param6 do nothing; they are only there for possible future uses. Constraint 1 is an adjustable circle, with the x coordinate of its center specified by **param1**, and its radius specified by **param2**. The text boxes **param3** and **param4** affect the second constraint space geometry. **param3** specifies how many circles constitute the second constraint, and **param4** controls a global rotation of constraint 2. **param4** is interpreted as a fraction of 2π, so .25 represents a rotation of $\pi/2$.

After parameters are input into the text boxes, hitting the **Save** button will save these parameters, and prepare the DM for calculation. Alternately, hitting the **Re-Calculate** will recalculate the field of view, taking into account any newly input parameters.

The **write params** button will take the current set of parameters and write the parameters to a file called "params.db" in the current directory.[5] The **read params** button does the inverse, reading the "params.db" file and replacing the current parameters with those saved in the file.[6] If a particular nice set of parameters are discovered by the user, this feature allows the parameters to be preserved.

To navigate the 2D landscape, the **zoom** button and the **re-center** button are used. The **zoom** button rescales the picture (in proportion to the input of the text box next to

[4] 1000 iterations seems to be sufficient for most pixels with typical constraints, to converge.

[5] **write params** will not record parameters input into the text boxes that have not been "saved" yet.

[6] Note this does not recalculate the picture.

it), and automatically recalculates the field of view. For example, if 0.1 is in the text box next to this button, pressing **zoom** will zoom in by a factor of 10 upon the picture's center. Likewise, a zoom factor of 10 will zoom out, keeping the pictures center the same. To re-center the plot region, use the **re-center** button. Left clicking on the picture will produce a red dot where clicked. After this dot is plotted, pressing **re-center** will re-calculate the picture with the indicated spot as the new center, with the same scale.

The **track point** button will draw the DM evolution of a selected point. Left clicking on the picture will produce a red dot, and the **track point** button will show how this spot is evolved by the DM. As the selected point is iterated, the plotted iterates change color. The first 200 iterations are blue, the second 200 are cyan, the next 200 are green, the next 200 are yellow, and iterations 800 to 1000 are red. Also, the line connecting iterates becomes thinner as the iterations increase. And example DM trajectory displayed by the **track point** button is shown in figure B.2.

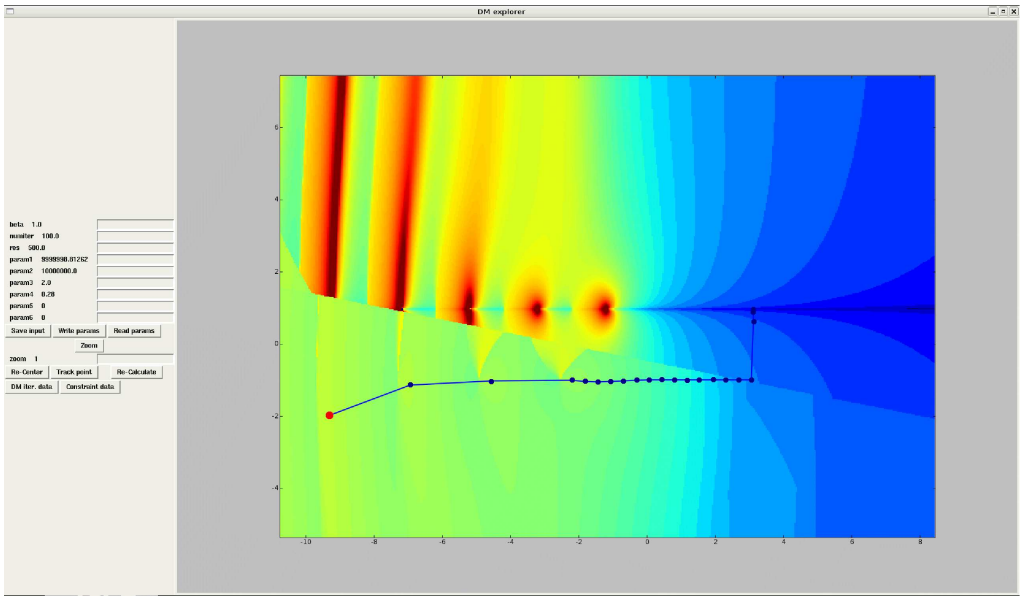

Figure B.2: An example trajectory for the DM. The initial point is shown as a red dot. Subsequent iterations are shown as blue dots connected with a blue line.

To view the constraint geometry, press the **constraint data** button. In this mode,[7] inputing new parameters in the **param** text boxes, and pressing **re-calculate** will update the constraint geometry. Re-centering and zooming are done in the same way as indicated above. Furthermore, the **track point** feature works the same way. When satisfied with the constraint geometry, hitting the "DM data" button will return to the original DM iteration mode with the updated geometry.

B.2 Python GUI code

The GUI code for DMX is entirely contained in the file "DMx2.py", which is a Python script file. Most of the file will be reproduced here, and explained. The file begins with included packages:

```
import matplotlib
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import Tkinter as Tk
import pylab
import numpy
import pickle
from pyconverter import *
```

Notice the imported package, **pyconverter**. The result of the Pyrex code (see Pyrex code section below) is a precompiled Python library that has in it Python usable C functions. This precompiled library is called "pyconverter.so", and its creation is described in the compilation section below.

Continuing with the "DMx2.py" code, we have an important class:

```
class dataholder(object):
    def __init__(self):
    self.param={
        "beta":1.,
        "xmin":-4*1.5,
        "xmax":4*1.5,
        "ymin":-4,
        "ymax":4,
        "res":200,
        "numiter":100,
        "param1":-.3,
```

[7] Adjusting the constraint geometry should be done in this mode only.

```
        "param2":2.225042147666,
        "param3":3,
        "param4":.13,
        "param5":0,
        "param6":0,
        "zoom":1,
        "trkpt":[0,0]}
    self.newrng=[[-2*1.5,2*1.5],[-2,2]]
    self.dat=[0,0]
    self.dataset="DMiter"
```

This **dataholder** class has a library member called **param**, which holds all of the parameters used to calculate the picture. When new parameters are input in the text boxes, they are subsequently stored in this library. The initial values displayed above are the parameter values used when starting the DMX program. The **trkpt** key in the library stores where the user clicks in the picture. The class member **self.dataset** indicates which data set to display; either the DM iteration data or the constraints picture. The member **self.dat** stores all the data that is displayed (both the DM iteration data, and the constraint picture data).

Next in the “DMx2.py” code, we have a function that creates a text box:

```
def mktxbx(name, frame):
    tmpf=Tk.Frame(frame)
    tmpf.pack(side="top", fill="x")
    txt = Tk.StringVar()
    tmpl = Tk.Label(tmpf, textvariable=txt)
    txt.set("  "+name+"    "+str(ad.param[name])+"   ")
    tmpl.pack(side="left")
    tmpv = Tk.Entry(tmpf)
    tmpv.pack(side="right")
    return {"frm":tmpf, "lab":txt, "val":tmpv}
```

This function (mktxbx is short for “make text box”) is passed both a name (string) for the text box, and a frame label to indicate where to put the text box. The function returns a library. The **lab** key in the library is a pointer to the text displayed next to the text box. By modifying the entry for this key, it changes the text displayed next to the box. This is how the numeric values of the parameters are displayed next to text boxes, and how they are modified.

That is all the definitions, now we begin the main operation of the program.

```
ad=dataholder()
cdat=c_picholder()

root = Tk.Tk()
root.title("DM explorer")
pylab.ion()
```

Hence, **ad** [8] is an instance of the **dataholder** class defined above. The **c_picholder** class is defined in the imported library "pyconverter.so", which will be described in the Pyrex section below. The **c_picholder** class contains all of the C functions as member functions. Hence, the *cdat=c_picholder()* line means the fast C functions can be called as **cdat.[c function]**. The **root** variable is the top level Tkinter widget that will contain all of the GUI. **Root** is broken up into smaller frames next.

```
optfram = Tk.Frame(root)
optfram.pack(side="left")
paramframe=Tk.Frame(optfram)
paramframe.pack(side="top", fill="x")
butframe=Tk.Frame(optfram)
butframe.pack(side="top", fill="x")
datframe=Tk.Frame(optfram)
datframe.pack(side="bottom", fill="x")
```

Here, the main **root** window is broken up into smaller frames. The **optfram** frame [9] contains the frames **paramframe** for text boxes, **butframe** for buttons, and the **datframe** for the data set frame (for the **DM data** and **constraint data** buttons). Once the frames for text boxes and buttons are made, next the picture is placed in the **root** window.

```
f = matplotlib.figure.Figure()
ax = f.add_subplot(111)

canvas = FigureCanvasTkAgg(f, master=root)
canvas.get_tk_widget().pack(fill="both", expand=1)
```

The variable **canvas** here, embedded in the root window, is where the main picture is drawn. Specifically, the DM iteration data is plotted in the **ax** subplot of the figure **f**, shown in the picture **canvas**, embedded in the **root** window.

[8] **ad** is short for "all data"

[9] "optfram" is short for "options frame".

At this point all the frames and pictures are defined, next we define the functions called by the various buttons. After the functions are defined, the buttons and text boxes will be placed in the frames.

The first function is the **save** function, which is run when the **save** button is pressed.

```
def Save():
     for i in range(len(prmtxbxs)):
          contents = prmtxbxs[i]["val"].get()
          name = (prmtxbxs[i]["lab"].get()).split()[0]
          if not (contents==''):
               try:
                    float(contents)
               except:
                    print "Not a number!!!!"
                    break
               ad.param[name]=float(contents)
          prmtxbxs[i]["lab"].set(" "+name+" "+str(ad.param[name])+" ")
          prmtxbxs[i]["val"].delete(0,"end")
```

Here **prmtxbxs**[10] is a list of text boxes. The loop considers each text box one at a time, and if the text box is not empty, the corresponding parameter in the **param** library of **ad** is updated. The label to the non-empty text is updated to show the new parameter value, and the text box is cleared. Thus, the **save** button records any new input information into the parameter library, but doesn't redraw the picture.

Next we have the function called when the **write params** and **read params** buttons are pressed:

```
def Write():
    f = file("params.db", "wb")
    notes = ad.param
    pickle.dump(notes, f)
    f.close()

def Read():
    f = file("params.db", "rb")
    notes = pickle.load(f)
    ad.param=notes
    f.close()
    Save()
```

Their operation if fairly straightforward. The **pickle**[11] command saves the parameter library to a file very conveniently. Notice that at the end of the **Read** function, the **save**

[10] "prmtxbxs" is short for "parameter text boxes"

[11] The creators of the Python language had a fine sense of humor.

function is called, updating the displayed parameter values.

The next two functions, **zoom** and **center**, are activated by pressing their corresponding buttons.

```
def Zoom():
    Save()
    zvo2=ad.param["zoom"]/2.
    xcent=(ad.param["xmax"]+ad.param["xmin"])/2.
    ycent=(ad.param["ymax"]+ad.param["ymin"])/2.
    ad.newrng[0][0]=xcent-(ad.param["xmax"]-ad.param["xmin"])*zvo2
    ad.newrng[0][1]=xcent+(ad.param["xmax"]-ad.param["xmin"])*zvo2
    ad.newrng[1][0]=ycent-(ad.param["ymax"]-ad.param["ymin"])*zvo2
    ad.newrng[1][1]=ycent+(ad.param["ymax"]-ad.param["ymin"])*zvo2
    ad.param["xmin"]=ad.newrng[0][0]
    ad.param["xmax"]=ad.newrng[0][1]
    ad.param["ymin"]=ad.newrng[1][0]
    ad.param["ymax"]=ad.newrng[1][1]
    Calc()
    ad.param["zoom"]=1
    Save()

def Center():
    ad.newrng[0][0]=ad.param["trkpt"][0]-(ad.param["xmax"]-ad.param["xmin"])/2.
    ad.newrng[0][1]=ad.param["trkpt"][0]+(ad.param["xmax"]-ad.param["xmin"])/2.
    ad.newrng[1][0]=ad.param["trkpt"][1]-(ad.param["ymax"]-ad.param["ymin"])/2.
    ad.newrng[1][1]=ad.param["trkpt"][1]+(ad.param["ymax"]-ad.param["ymin"])/2.
    ad.param["xmin"]=ad.newrng[0][0]
    ad.param["xmax"]=ad.newrng[0][1]
    ad.param["ymin"]=ad.newrng[1][0]
    ad.param["ymax"]=ad.newrng[1][1]
    Calc()
```

Both the **zoom** and **center** functions work the same way. First, the new field of view is calculated, the new field of view is saved in the parameter library of **ad** and the new picture is calculated (via the **Calc()** command). Notice the **zoom** function first saves any new input parameters (such as a zoom value), and uses the new zoom value to calculate the field of view. Afterwards it sets the zoom value in the **param** library to 1, and displays this updated parameter in the window with the **save** function.

Finally, we have the main calculating function, **Calc”**. This function calculates the displayed pictures.

```
def Calc():
    Save()
    cdat.mk_c_params(ad.param)
    ad.dat=cdat.c_get_pic()
    ax.clear()
    if ad.dataset=="DMiter":
        ax.imshow(ad.dat[0])
    if ad.dataset=="constraints":
```

```
        ax.imshow(ad.dat[1])
    canvas.show()
```

Upon pressing the **re-calculate** button, first any new input parameters are recorded via the **save** function. Next these parameters are passed to a member function of **cdat**. **cdat** is an instance of a class defined in the Pyrex code (see the Pyrex Code section below). The line *cdat.mk_c_params(ad.param)* essentially creates a C structure, also a member of the class **cdat**. Next this C structure is sent to a C function (another member function of **cdat**) that iterates every pixel in the field of view and records how long it takes to converge. This function (explained below) returns a list of two data sets, the first is the DM iteration data, the second is the constraint sets data. These data sets are stored in the **dat** member list of **ad**. Finally, the picture is cleared, re-drawn (depending on which mode **ad.dataset** is currently), and shown. The execution of this function is the most time consuming aspect of the program, and for this reason implemented primarily in C.

The next function in the Python GUI is activated by clicking on the **track point** button.

```
def Trackpt():
    tmp=ad.param["numiter"]
    ad.param["numiter"]=1000
    cdat.mk_c_params(ad.param)
    path=cdat.c_trackpt()
    ax.clear()
    ax.plot([ad.param["trkpt"][0]],[ad.param["trkpt"][1]],'ro')
    for i in range(int(ad.param["numiter"])-1):
        p1=[path[i][0],path[i+1][0]]
        p2=[path[i][0],path[i+1][0]]
        ax.plot(p1, p2, 'bo-', ms=2.-2.*i/1000., linewidth=2.-1.9*i/1000.)
    if ad.dataset=="DMiter":
        ax.imshow(ad.dat[0])
    if ad.dataset=="constraints":
        ax.imshow(ad.dat[1])
    ad.param["numiter"]=tmp
    canvas.show()
```

Notice that the maximum number of DM iterations is set temporarily to 1000, and reset back to its initial value at the end of the function. Like in the **calc** function,

the line *cdat.mk_c_params(ad.param)* creates a C structure, which is a member of the **c_picholder** class defined in the Pyrex code below. The C function **c_trackpt** is executed and returns a list of 1000 points the selected point[12] is iterated through. The picture is cleared, these 1000 points drawn in the figure, the relevant data set redrawn, and finally shown.

The **onpress** function is executed when the user clicks on the picture.

```
def onpress(event):
    if ax==event.inaxes:
        ad.param["trkpt"]=[event.xdata, event.ydata]
        ax.clear()
        ax.plot([event.xdata], [event.ydata], 'ro')
        if ad.dataset=="DMiter":
            ax.imshow(ad.dat[0])
        if ad.dataset=="constraints":
            ax.imshow(ad.dat[1])
        canvas.show()
```

If the click occurs outside the picture, this function does nothing. Otherwise, the coordinates of the click are recorded in the "trkpt" key of the "params" library. The picture is cleared, the red dot drawn, followed by redrawing the relevant data set. Finally the picture is shown on the canvas.

The next two functions are activated by clicking on the **DM iter. data** and the **Constraint data** buttons respectively.

```
def DMdata():
    ad.param["res"]=250
    ad.param["numiter"]=50
    ad.dataset="DMiter"
    Calc()

def consdata():
    ad.param["res"]=200
    ad.param["numiter"]=2
    ad.dataset="constraints"
    Calc()
```

These two functions change the value of the **dataset** member of the **ad** variable.[13] They also change the picture resolution and the maximum number of DM iterations in

[12]The coordinates of the initial point are stored in the **trkpt** key of the **params** library.

[13]**ad** is an instance of the **dataholder** class defined above.

order to make constraint picture redraw quickly when constraint parameters are changed. Notice when these functions are executed, there is a time consuming re-calculation of both data sets.

Now all the functions tied to the various buttons have been defined. Next we add the text boxes to the relevant frames.

```
prmtxbxs=[]
prmtxbxs.append(mktxbx("beta", paramframe))
prmtxbxs.append(mktxbx("numiter", paramframe))
prmtxbxs.append(mktxbx("res", paramframe))
prmtxbxs.append(mktxbx("param1", paramframe))
prmtxbxs.append(mktxbx("param2", paramframe))
prmtxbxs.append(mktxbx("param3", paramframe))
prmtxbxs.append(mktxbx("param4", paramframe))
prmtxbxs.append(mktxbx("param5", paramframe))
prmtxbxs.append(mktxbx("param6", paramframe))
```

The list **prmtxbxs** is defined here, and subsequently filled with the various text boxes. All of these text boxes go in the **paramframe** frame defined above. The function **mktxbx** is also defined above, it literally makes the text box, and returns a library with pointer information regarding access to the created text box.

After the text boxes are created, the buttons are created next,

```
save_button = Tk.Button(paramframe, text="Save input", command = Save)
save_button.pack(side="left")
write_button = Tk.Button(paramframe, text="Write params", command = Write)
write_button.pack(side="left")
read_button = Tk.Button(paramframe, text="Read params", command = Read)
read_button.pack(side="left")
zoom_button = Tk.Button(butframe, text="Zoom", command = Zoom)
zoom_button.pack(side="top")
prmtxbxs.append(mktxbx("zoom", butframe))
center_button = Tk.Button(butframe, text="Re-Center", command = Center)
center_button.pack(side="left")
tp_button = Tk.Button(butframe, text="Track point", command = Trackpt)
tp_button.pack(side="left")
calc_button = Tk.Button(butframe, text="Re-Calculate", command = Calc)
calc_button.pack(side="bottom")
DMd_button = Tk.Button(datframe, text="DM iter. data", command = DMdata)
DMd_button.pack(side="left")
consd_button = Tk.Button(datframe, text="Constraint data", command = consdata)
consd_button.pack(side="left")
```

Notice the buttons are created, and tied to the various functions defined above. Notice also that there is a text box created in the **butframe** frame, and added to the **prmtxbxs** list of text boxes.

Finally, the GUI ends with,

```
paramframe.bind("<Return>", Save)
ax.figure.canvas.mpl_connect('button_press_event',onpress)

Calc()
root.mainloop()
```

This makes the "return" key execute the **save** function above, and clicking on the picture executes the **onpress** function defined above. Ultimately, the picture is calculated with the **Calc** function defined above, and the GUI is begun with the *root.mainloop()* command.

B.3 Pyrex code

Pyrex is a computer language that allows C functions to be made into a Python library. The code below is meant to accept Python data types, turn them into C structures, execute C code on these C structures, and return Python data types. The Pyrex file, along with the C code, needs to be compiled into a shared library accessible by the main Python code. The compilation process is explained below.

The syntax in Pyrex is similar to that of Python, but there are many C commands that are usable. For example, the Pyrex code used by the DMX is contained in the file "pyconverter.pyx". The file begins with,

```
cdef extern from "stdlib.h":
    void *malloc(int)
    void free(void *)
```

Notice the syntax is similar to that of Python. This small bit of code finds the **malloc** and **free** C commands in "stdlib.h" (a C header file) and makes them usable in the current Pyrex file.

Similarly, the C header file "c_code.h" is turned into something Pyrex can interpret by the following bit of Pyrex code,

```
cdef extern from "c_code.h":
    ctypedef struct c_params "params":
        double beta
        double xmin
        double xmax
        double ymin
        double ymax
        int res
        int numiter
        double param1
        double param2
        double param3
        double param4
        double param5
        double param6
        double ipt[2]

    cdef extern void gtpltdata "gtpltdata" (double **, double **, c_params)
    cdef extern void gtpath "gtpath" (double **, c_params)
```

In the header file "c_code.h",[14] there is a C structure defined called **params**. This bit of Pyrex code creates a local data type called "c_params" which, like a C structure or a Python class, has members whose data types are here defined. Also included in "c_code.h", the two C functions **gtpltdata** and **gtpath** are prototyped. Here in the Pyrex code, they are prototyped as well so they can be used by the Pyrex code below. They are given the same name as the C version, though they don't necessarily have to be.

The next part of the Pyrex code is,

```
cdef class c_picholder:
    cdef c_params par
```

This is the essence of Pyrex: a strange mixture of Python syntax and C syntax. Essentially this is the beginning of a Python class [15], with more member functions below. The line *cdef c_params par* is essentially a C declaration. Here, the variable **par** is defined to be of the type **c_params**. This data type was defined above. The **c_params** data type has members like a C structure.

The next function,[16]

[14]The file is displayed below, in the C Code section.

[15]The Python class is named **c_picholder**. In the Python GUI above, the variable **cdat** is an instance of this class.

[16]Note all of the following functions are members of the **c_picholder** class.

```
def mk_c_params(self, pythpar):
    self.par.beta=pythpar["beta"]
    self.par.xmin=pythpar["xmin"]
    self.par.xmax=pythpar["xmax"]
    self.par.ymin=pythpar["ymin"]
    self.par.ymax=pythpar["ymax"]
    self.par.res=pythpar["res"]
    self.par.numiter=pythpar["numiter"]
    self.par.param1=pythpar["param1"]
    self.par.param2=pythpar["param2"]
    self.par.param3=pythpar["param3"]
    self.par.param4=pythpar["param4"]
    self.par.param5=pythpar["param5"]
    self.par.param6=pythpar["param6"]
    self.par.ipt[0]=pythpar["trkpt"][0]
    self.par.ipt[1]=pythpar["trkpt"][1]
```

takes in a Python libary **pythpar**, and copies its contents into the **c_params** variable **par**. This **mk_c_params** function is used in the Python GUI in the line *cdat.mk_c_params(ad.param)*.

The next function (also a member of the **c_picholder** class) is the main computation function. It accesses the parameters stored in the variable **par**. **par** was created and filled in with a passed Python library with the function **mk_c_params**.

```
def c_get_pic(self):
    cdef double **pic
    cdef double **cons
    n=self.par.res

    pic = <double **> malloc(n * sizeof(double *))
    for i in range(n):
        pic[i]=<double *> malloc(n * sizeof(double))
    cons = <double **> malloc(n * sizeof(double *))
    for i in range(n):
        cons[i]=<double *> malloc(n * sizeof(double))

    gtpltdata(pic, cons, self.par)

    picout=[]
    consout=[]
    for i in range(n):
        picout.append([])
        consout.append([])
        for j in range(n):
            picout[i].append(pic[i][j])
            consout[i].append(cons[i][j])
    for i in range(n):
        free(pic[i])
    free(pic)
    for i in range(n):
        free(cons[i])
    free(cons)
    return [picout, consout]
```

Notice the strange mixture of Python and C here. *cdef double **pic* is C syntax that declares the variable **pic** to be an array of pointers. The lines:

```
    pic = <double **> malloc(n * sizeof(double *))
    for i in range(n):
        pic[i]=<double *> malloc(n * sizeof(double))
```

make the variable **pic** into an n by n array of double precision numbers.

The function **gtpltdata** is a C function (described in the C Code section below), that was prototyped in this Pyrex file above. It takes as arguments pointers to two n by n double arrays, and a C structure containing the DM parameters. Inside the **gtpltdata** function, the two arrays are filled in with data. Next the Python command *picout=[]* declares **picout** as an empty Python list. **picout** is subsequently filled in with data from the 2D **pic** array. Next the **pic** array is freed from memory, and finally two Python lists are returned. This function is called in the Python GUI by the line *ad.dat=cdat.c_get_pic()*.

The last function in ”pyconverter.pyx” is the track point function.

```
def c_trackpt(self):
    cdef double **path
    n=self.par.numiter

    path = <double **> malloc(n * sizeof(double *))
    for i in range(n):
        path[i]=<double *> malloc(2 * sizeof(double))

    gtpath(path, self.par)

    pathout=[]
    for i in range(n):
        pathout.append([path[i][0],path[i][1]])

    for i in range(n):
        free(path[i])
    free(path)

    return pathout
```

Structurally it is very similar to the previous, **c_get_pic** function. First there is the allocation of memory to a 2D C array of double precision numbers. Next this array is passed to a C function, **gtpath**. This function is described below. The function fills in the array with the iteration trajectory of the initial point specified in the **self.par** structure. Finally the trajectory is copied to a Python list, the allocated memory is freed, and the trajectory is returned to the Python GUI as a list.

B.4 C code

The C code used to do the main computation in the DMX program is much less esoteric that the Python GUI, or the Pyrex interpreting code, so it will be reproduced here only with minimal comments.

The header file "c_code.h" defines the **params** C structure. This C structure is passed to all the C functions and contains information about the range of pixels to iterate, the DM β, and parameters for the constraint geometries.

The header file has,

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

typedef struct {
    double beta;
    double xmin;
    double xmax;
    double ymin;
    double ymax;
    int res;
    int numiter;
    double param1;
    double param2;
    double param3;
    double param4;
    double param5;
    double param6;
    double ipt[2];
} params;
```

Note this structure is type defined again in the Pyrex code above. The two definitions must be identical, or there are compilation errors.

The header file ends with prototypes for all the functions in "c_code.c",

```
void proj1(double *, double *, params par);
void proj2(double *, double *, params par);
double DM(double *, double *, params par);
int iterate(double *, params par);
void gtpltdata(double **, double **, params);
void gtpath(double **, params);
```

In "c_code.c", the first function is passed an input array (**x**), an output array (**y**), and a **params** structure containing all the parameters.

```
void proj1(double *x, double *y, params par){
    double mag, fc[2];

    fc[0]=x[0]-par.param1;
    fc[1]=x[1]-0;

    mag=sqrt(fc[0]*fc[0]+fc[1]*fc[1]);

    if (mag>0) {
        y[0]=par.param1+fc[0]*par.param2/mag;
        y[1]=0+fc[1]*par.param2/mag; }
    if (mag==0)  { y[0]=x[0];  y[1]=x[1]; }
}
```

This function performs a projection, and uses the input coordinates in the **x** array and uses them to fill the **y** array. The constraint space associated with this projection is a circle of radius **par.param2**, and its center is located at **par.param1** on the x axis.

The next function of "c_code.c" is the second projection,

```
void proj2(double *x, double *y, params par){
    int numcirc= (int) par.param3, i, cc=0, scc=0;
    double mag, **cntrs, dots[numcirc], mx=-10000, smx=-10000, rmin, dist[2];

    cntrs = (double **) malloc(numcirc * sizeof(double *));
    for (i=0; i<numcirc; i++){
            cntrs[i]=(double *) malloc(2 * sizeof(double));
    }
    for (i=0; i<numcirc; i++){
        cntrs[i][0]=cos(3.14159265358979*2.*(1.*i/numcirc+par.param4));
        cntrs[i][1]=sin(3.14159265358979*2.*(1.*i/numcirc+par.param4));
    }
    for (i=0; i<numcirc; i++){
        dots[i]=x[0]*cntrs[i][0]+x[1]*cntrs[i][1];
        if (dots[i]>mx) { smx=mx; mx=dots[i];  scc=cc; cc=i; }
        else if (dots[i]>smx) { smx=dots[i];  scc=i; }
    }

    if (numcirc==1){
        cc=0;
    }
    dist[0]=(x[0]-cntrs[cc][0]);
    dist[1]=(x[1]-cntrs[cc][1]);
    mag=sqrt(dist[0]*dist[0]+dist[1]*dist[1]);

    if (mag>0) {
        y[0]=cntrs[cc][0]+(x[0]-cntrs[cc][0])/mag;
        y[1]=cntrs[cc][1]+(x[1]-cntrs[cc][1])/mag;
    }
    else  { y[0]=cntrs[cc][0];  y[1]=cntrs[cc][1]; }

    rmin=sqrt(2-2*cos(3.14159265358979-3.14159265358979*2./numcirc));
    if (numcirc!=1){
        if (sqrt(y[0]*y[0]+y[1]*y[1])<rmin  ){
            y[0]=(cntrs[cc][0]+cntrs[scc][0]);
            y[1]=(cntrs[cc][1]+cntrs[scc][1]);
        }
    }
    for (i=0; i<numcirc; i++){
            free(cntrs[i]);
```

```
    }
        free(cntrs);
}
```

An example of this constraint space is shown in figure B.3. We refer to this geometry as the "exterior union" of n circles (in this case, 5 circles). Each circle has unit radius and their centers are evenly spread around the unit circle.

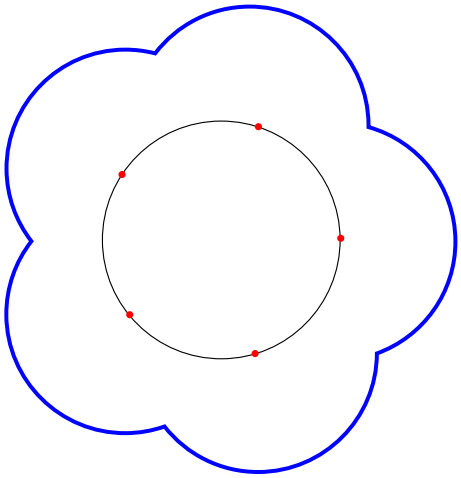

Figure B.3: Here, the exterior union of five circles is shown as the blue contour. The circles' centers are shown by red dots, evenly spread out around the unit circle.

Just like the function **proj1** above, this function takes in two arrays and a **params** structure. The **x** array is the input to the projection and the **y** array is filled with the output of the projection.

Next is the DM function.

```
double DM(double *x, double *y, params par){
    double p1[2], p2[2], f1[2], f2[2], d[2];

    proj1(x, p1, par);
      proj2(x, p2, par);

    f1[0]=p1[0]+(p1[0]-x[0])/par.beta;
    f1[1]=p1[1]+(p1[1]-x[1])/par.beta;
    f2[0]=p2[0]-(p2[0]-x[0])/par.beta;
    f2[1]=p2[1]-(p2[1]-x[1])/par.beta;

    proj1(f2, p1, par);
    proj2(f1, p2, par);
```

```
    d[0]=p2[0]-p1[0];
    d[1]=p2[1]-p1[1];

    y[0]=x[0]+par.beta*d[0];
    y[1]=x[1]+par.beta*d[1];
    return sqrt(d[0]*d[0]+d[1]*d[1]);
}
```

Like the two previous functions, this function takes in two arrays, **x** is the input and **y** is filled with the output. This function evolves the point **x** by one DM iteration.[17] The function returns how far the input point was moved.

Next is the function that iterates a pixel via the DM until a fixed point is found.

```
int iterate(double *x, params par){
    int count, i;
    double er, xnew[2];

    count=0;
    for (i=0; i<par.numiter; i++){
        er=DM(x, xnew, par);
        count=count+1;
        x[0]=xnew[0];
        x[1]=xnew[1];
        if (er<.00001)  break;
    }
    return count;
}
```

It takes in a 2D point, and iterates it via the DM function above until a fixed point is found.[18] Upon either finding a fixed point or the iterations run out, the number of iterations conducted is returned.

The last two functions are the main functions of this file. The first function, **gtpltdata**, generates the pictures to be displayed. The second function, **gtpath**, calculates the iterate trajectory for a given initial point.

```
void gtpltdata(double **pic, double **cons, params par){
    double xmin, xmax, ymin, ymax, px, py, x[2], p1[2], p2[2], d;
    int i, j, it, d1[2], d2[2];

    xmin=par.xmin;
    xmax=par.xmax;
    ymin=par.ymin;
    ymax=par.ymax;
```

[17]For an explanation on the DM, see chapter 2.

[18]The **iterate** function quits if either a fixed point is found, or if **par.numiter** iterations happen– whichever happens first.

```
    for (j=0; j<par.res; j++) for (i=0; i<par.res; i++)  cons[i][j]=0;

    for (j=0; j<par.res; j++){
        for (i=0; i<par.res; i++){
            px=xmin+(xmax-xmin)*(1.*j)/par.res;
            py=ymax-(ymax-ymin)*(1.*i)/par.res;
            x[0]=px;
            x[1]=py;
            it=iterate(x, par);
            pic[i][j]=log(it);
            px=xmin+(xmax-xmin)*(1.*j)/par.res;
            py=ymax-(ymax-ymin)*(1.*i)/par.res;
            x[0]=px;
            x[1]=py;
            proj1(x, p1, par);
            d1[0]=(int)((p1[0]-xmin)*par.res/(xmax-xmin));
            d1[1]=(int)(-1.*(p1[1]-ymax)*par.res/(ymax-ymin));
            if (d1[0]>=0 && d1[0]<par.res && d1[1]>=0 && d1[1]<par.res) {
                cons[ d1[1] ][ d1[0] ]=.8;
                d=sqrt((x[0]-p1[0])*(x[0]-p1[0])+(x[1]-p1[1])*(x[1]-p1[1]));
                if (d<1.*(xmax-xmin)/par.res ) cons[ i ][ j ]=1.;
            }
            proj2(x, p2, par);
            d2[0]=(int)((p2[0]-xmin)*par.res/(xmax-xmin));
            d2[1]=(int)(-1.*(p2[1]-ymax)*par.res/(ymax-ymin));
            if (d2[0]>=0 && d2[0]<par.res && d2[1]>=0 && d2[1]<par.res) {
                cons[ d2[1] ][ d2[0] ]=-.8;
                d=sqrt((x[0]-p2[0])*(x[0]-p2[0])+(x[1]-p2[1])*(x[1]-p2[1]));
                if (d<1.*(xmax-xmin)/par.res ) cons[ i ][ j ]=-1.;
            }
        }
    }
}
```

This function fills in both the **pic** array (the DM iteration data) but also the **cons** array (the picture of the the constraint spaces). These two arrays are created in the Pyrex code above, and this function is called in the Pyrex line, *gtpltdata(pic, cons, self.par)*, inside the Pyrex function **c_get_pic**. Note also that numbers put into the **pic** array are the logarithm of the number of iterations.

The final function in “c_code.c” is,

```
void gtpath(double **path, params par){
    int i;
    double er, xnew[2], x[2];

    x[0]=par.ipt[0];
    x[1]=par.ipt[1];
    for (i=0; i<par.numiter; i++){
        path[i][0]=x[0];
        path[i][1]=x[1];
        er=DM(x, xnew, par);
        x[0]=xnew[0];
        x[1]=xnew[1];
    }
}
```

This function is passed an array of 1000 2D points. The initial point to begin the DM iteration from is specified in the **params** structure. This point is iterated 1000 times, and its trajectory is stored in the **path** array. This function is called in the line *gtpath(path, self.par)* inside the **c_trackpt** function in the Pyrex code.

B.5 Compilation

The C code is compiled by the "setup.py" file, which is in turn called by the Makefile. The Makefile is very standard and straightforward,

```
all:    clean
    python Setup.py build_ext --inplace

clean:
    @rm -f pyconverter.c *.o *.so *~ core
    @rm -rf build
```

First, it removes old versions of the compiled files, then it calls the "setup.py" file,

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

setup(
  ext_modules=[
    Extension("pyconverter", ["pyconverter.pyx", "c_code.c"]),
    ],
  cmdclass = {'build_ext': build_ext}
)
```

This setup file compiles the C code and Pyrex code into a shared library for Python, called "pyconverter.so". From Python, the command *from pyconverter import* * imports the class **c_picholder** defined in "pyconverter.pyx". Once this is done, the C functions are able to be accessed and run from Python.

In summary, typing "make" from the directory containing the C and the Pyrex code, the makefile, and the Setup file will compile the external modules so the Python script can access them. Once completed successfully, the main script "DMX.py" can be run from python.

Appendix C

NENA code

Here the source code for the protein structure prediction program NENA is entirely reproduced. The code itself is sufficiently commented, and before each reproduced file there is a short explanation of the contained functions' uses and operation.

C.1 NENA source code

The main program comes in three distinct parts; the Python GUI, the C code (where most of the heavy computation is done), and the Pyrex converter code that enables Python to access the C routines.

A schematic of the various file dependencies can be seen in figure C.1.

The program's structure is very similar to that of the difference map explorer program (DMX), which is thoroughly explained in appendix B. As an aid to understand the NENA code, the reader is strongly encouraged to read and completely understand appendix B.

C.1.1 Python GUI

The Python aspect of the program consists of three files; "nena.py", "pyfold.py", and "pyinit.py". The GUI is entirely contained in "nena.py", which mainly calls functions defined in "pyfold.py". The file "pyinit.py" only contains the initialization function. The initializing function takes the atom list from the relevant PDB file, and creates the atom lists, bond lists, determinant lists, and the Ramachandran lists used by NENA.[1]

[1] See chapter 4 for how these lists are used.

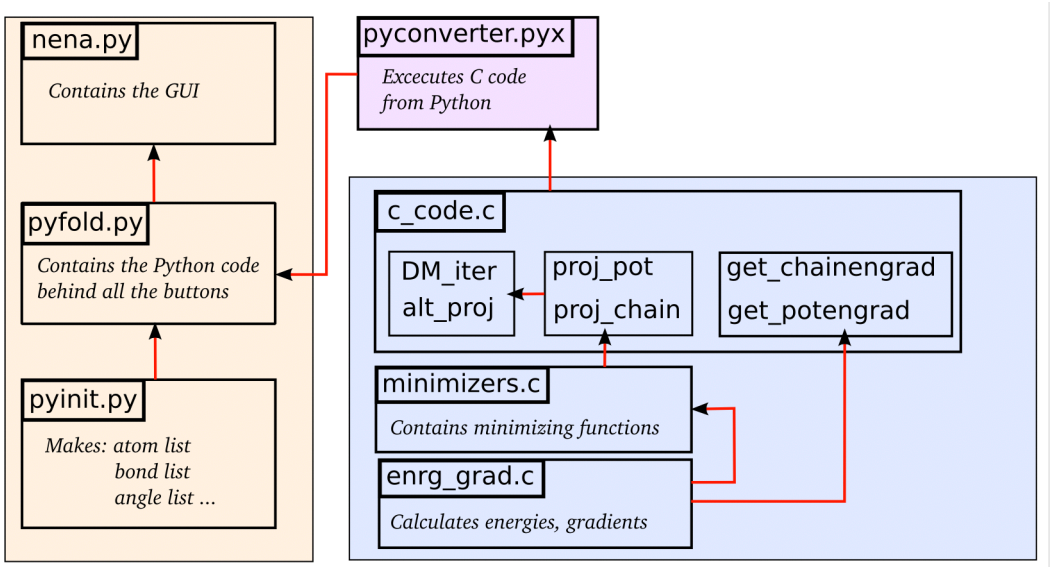

Figure C.1: Here the basic code structure for NENA is shown. The Python files are in the orange box, the C files are in the blue box, and the Pyrex converter file is in the purple box. The file structure is very similar to that in the difference map explorer program (DMX). The DMX program is thoroughly explained in appendix B.

nena.py

In this file the program's GUI is written. The GUI is very similar to the DMX GUI explained in appendix B. This file calls mainly functions found in "pyfold.py".

pyfold.py

This file defines most of the functions activated by the buttons in "nena.py". The functions contained here mostly call C code functions through the Pyrex file, "pyconverter.pyx". For a clear explanation of the role of the Pyrex file, see appendix B.

pyinit.py

This file contains one function, “init”. This function is passed an atom list from a PDB file, and creates a list of atoms, bonds, angles, determinants, and Ramachandran constraints. There is an important section near the end where such parameters as atomic radii, constraint energy weights, atomic charge, and hydrophobicity, are set.

C.1.2 Pyrex converter code

Pyrex is a computer language that allows C functions to be compiled into a Python library. The code below is meant to accept Python data types, turn them into C structures, execute C code on these C structures, and return Python data types. The Pyrex file “pyconverter.pyx”, along with the C code, needs to be compiled into a shared library accessible by the main Python code. This compilation is done with the Makefile, and “setup.py”.

pyconverter.pyx

This file prototypes all the C structures and C functions that are accessed from Python. It is compiled by “setup.py”. Note that the prototyping is also done in “c_code.h”, and the duplicate prototyping must be identical.

setup.py

This file compiles “pyconverter.pyx” into “pyconverter.so”, a shared library accessible by Python. This library is then imported, and the C functions below can be accessed by Python. The file “setup.py” is run by executing the Makefile.

C.1.3 C code

Most of the intensive computation is performed in the following C files. These functions are prototyped both in their corresponding header files, and also in “pyconverter.pyx”. By including them in “pyconverter.pyx”, they can then be accessed by Python.

c_code.h c_code.c

In this file are the main C functions. Contained here are the DM iteration function, and the alternating projection function. These functions call the subsequent functions in the files “minimizers.c” and “enrg_grad.c”.

minimizers.h minimizers.c

The energy minimizer routines are written in this file. These two functions essentially perform the projections used by the difference map.[2] The algorithm to minimize the various energies is described in chapter 4.4.1.

enrg_grad.h enrg_grad.c

The file “enrg_grad.c” contains functions that calculate the energy and gradient of an atomic configuration. Also in “enrg_grad.c” is the function that calculates the neighbor list of all the atoms.[3]

vector.h vector.c

These files contain some simple vector definitions and functions.

[2] See chapter 2 for more on the difference map algorithm.
[3] The use of the neighbor list is described in chapter 4.3.2.

structures.h

In this file the main C structures used by the C code are defined. Their use is fully explained in chapter 4.

Makefile

From the command line, typing “make” executes this file. It compiles the C code, and creates a Python shared library from “pyconverter.pyx”. The library can then be imported by Python.

```
import matplotlib
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure
import Tkinter as Tk
import sys
import pylab
import pickle
import subprocess

from pyfold import *   #"pyfold.py" has all the important protein operation functions

#foldmain() is in "pyfold.py"
pro=foldmain()

#This function makes a text box
def mktxbx(name, frame):
    tmpf=Tk.Frame(frame)
    tmpf.pack(side="top", fill="x")
    txt = Tk.StringVar()
    tmpl = Tk.Label(tmpf, textvariable=txt)
    txt.set(" "+name+"  "+str(pro.proteininfo[name])+" ")
    tmpl.pack(side="left")
    tmpv = Tk.Entry(tmpf)
    tmpv.pack(side="right")
    return {"frm":tmpf, "lab":txt, "val":tmpv}

root = Tk.Tk()
root.title("Protein Folder")
pylab.ion()
matplotlib.use('TkAgg')

#make frames
optfram = Tk.Frame(root);
optfram.pack(side="left");
paramframe=Tk.Frame(optfram);
paramframe.pack(side="top", fill="x");
butframe=Tk.Frame(optfram);
butframe.pack(side="top", fill="x");
datframe=Tk.Frame(optfram);
datframe.pack(side="bottom", fill="x");

#"ax" is a plot region in figure "f", on "canvas"
f = Figure(figsize=(3,2), dpi=200)
ax = f.add_subplot(111)
canvas = FigureCanvasTkAgg(f, master=root)
canvas.get_tk_widget().pack(fill="both", expand=1)
#now plotting is done by:
#ax.clear()
#ax.imshow(2D DATA SET, interpolation="nearest")
#canvas.show()

#this function is run during the operation of many buttons
#it records any new input parameters in "pro.proteininfo[PARAMETER]"
def Save():
    for i in range(len(prmtxbxs)):
        contents = prmtxbxs[i]["val"].get()
        name = (prmtxbxs[i]["lab"].get()).split()[0]
        if not (contents==''):
            print name
            if name=="name":
                pro.proteininfo[name]=contents
                pro.prefold()
```

```
            else:
                try:
                    float(contents)
                except:
                    print "Not a number!!!!"
                    break
                pro.proteininfo[name]=float(contents)
        prmtxbxs[i]["lab"].set(" "+name+"  "+str(pro.proteininfo[name])+" ")
        prmtxbxs[i]["val"].delete(0,"end")

#when the "Write new params" button is pressed, this function is run
#it checks for new input parameters, and writes params to file "params_tmp.db"
#it writes atoms list to file "atoms_tmp.db"
def Write():
    Save()
    pro.get_c_version()
    f = file("params_tmp.db", "wb")
    pickle.dump(pro.proteininfo, f)
    f.close()
    f = file("atoms_tmp.db", "wb")
    pickle.dump(pro.atoms, f)
    f.close()
    pro.make_c_version()

#when the "Read saved params" button is pressed, this function is run
#it reads file "params_tmp.db", displays new parameters in file.  Also, it reads "atoms_tmp.db", puts atom list in "pro.atoms"
def Read():
    f = file("params_tmp.db", "rb")
    pro.proteininfo=pickle.load(f)
    f.close()
    Save()
    pro.prefold()
    f = file("atoms_tmp.db", "rb")
    pro.atoms = pickle.load(f)
    f.close()
    pro.make_c_version()

#this function runs when "View iterate" button is pressed
#opens rasmol protein viewer in new window
def ViewIter():
    pro.get_c_version()
    outputPDBfile(pro.pdb_file, pro.atoms, "outputs/iter"+str(len(pro.progdat))+".pdb")
    subprocess.Popen("rasmol "+"outputs/iter"+str(len(pro.progdat))+".pdb", shell=True)
    pro.make_c_version()

#this function runs when "Randomize atoms" button is pressed
#add random vectors of magnitude "pro.proteininfo["rand_mag"]" to atom coordinates.
def Rdmz():
    pro.get_c_version()
    randomize(pro.atoms, pro.proteininfo["rand_mag"])
    pro.make_c_version()

#this function runs when "Refine atoms" button is pressed
# conducts alternating projections to find nearby local energy minimum
def Refine():
    pro.c_pro.c_altproj()
    pro.get_c_version()
    pro.make_c_version()
    Save()

#runs when "Chain Project" button is pressed
#projects the atomic iterate to the chain constraint
```

```
def ChainP():
    pro.c_pro.c_proj_chain()
    pro.get_c_version()
    pro.make_c_version()

#runs when "Potential Project" button is pressed
#projects the atomic iterate to the potential constraint
def PotP():
    pro.c_pro.c_proj_pot()
    pro.get_c_version()
    pro.make_c_version()

#runs when "DM iter" button is pressed
#does "pro.proteininfo["dm_iters"]" DM iterations.  Also it updates picture every 10 iterations
def DMi():
    for i in range(int(pro.proteininfo["dm_iters"]/10.)):
        pro.startfold(10)
        pro.get_c_version()
        pro.make_c_version()
        pro.progdat.reverse()
        ax.clear()
        ax.imshow(pro.progdat, interpolation="nearest")
        pro.progdat.reverse()
        Save()
        canvas.show()
    pro.startfold(pro.proteininfo["dm_iters"]-(int(pro.proteininfo["dm_iters"]/10.)*10))
    pro.get_c_version()
    pro.make_c_version()
    pro.progdat.reverse()
    ax.clear()
    ax.imshow(pro.progdat, interpolation="nearest")
    pro.progdat.reverse()
    Save()

#Make the text boxes
prmtxbxs=[]
prmtxbxs.append(mktxbx("name", paramframe))
prmtxbxs.append(mktxbx("beta", paramframe))
prmtxbxs.append(mktxbx("min_dm_err", paramframe))
prmtxbxs.append(mktxbx("dm_iters", paramframe))
prmtxbxs.append(mktxbx("refine_iters", paramframe))
prmtxbxs.append(mktxbx("restartevery", paramframe))
prmtxbxs.append(mktxbx("outputprogevery", paramframe))
prmtxbxs.append(mktxbx("refinenotfor", paramframe))
prmtxbxs.append(mktxbx("prg", paramframe))
prmtxbxs.append(mktxbx("rand_mag", paramframe))
prmtxbxs.append(mktxbx("chain_proj_iter_max", paramframe))
prmtxbxs.append(mktxbx("chain_max_step_size", paramframe))
prmtxbxs.append(mktxbx("chain_min_step_size", paramframe))
prmtxbxs.append(mktxbx("chain_min_nrg", paramframe))
prmtxbxs.append(mktxbx("pot_proj_iter_max", paramframe))
prmtxbxs.append(mktxbx("pot_max_step_size", paramframe))
prmtxbxs.append(mktxbx("pot_min_step_size", paramframe))
prmtxbxs.append(mktxbx("pot_min_nrg", paramframe))

#make buttons
write_button = Tk.Button(paramframe, text="Write new params", command = Write)
write_button.pack(side="left")
read_button = Tk.Button(paramframe, text="Read saved params", command = Read)
read_button.pack(side="left")
vi_button = Tk.Button(butframe, text="View iterate", command = ViewIter)
vi_button.pack(side="top")
```

```
rdmz_button = Tk.Button(butframe, text="Randomize atoms", command = Rdmz)
rdmz_button.pack(side="top")
refine_button = Tk.Button(butframe, text="Refine atoms", command = Refine)
refine_button.pack(side="top")
chainp_button = Tk.Button(butframe, text="Chain Project", command = ChainP)
chainp_button.pack(side="bottom")
potp_button = Tk.Button(butframe, text="Potential Project", command = PotP)
potp_button.pack(side="bottom")
DM_button = Tk.Button(butframe, text="DM iter", command = DMi)
DM_button.pack(side="top")

#BEGIN!!!
root.mainloop()
```

```
import numpy
import scipy
import Scientific.IO.PDB as sci
import pylab
from pyconverter import *
from pyinit import *

#adds a random vector of magnitude "mag" to every atom coordinate
def randomize(atoms, mag):
    for i in range(len(atoms)):
        nt=100
        while nt>1.:
            vt=sci.Vector(scipy.rand()*2.-1., scipy.rand()*2.-1., scipy.rand()*2.-1.)
            nt=vt.length()
        atoms[i]["pos"]=atoms[i]["pos"]+vt*mag/nt

#outputs the atom coordinates in PDB form
def outputPDBfile(x,atoms,name):
    for i in range(len(atoms)):
        aminoacidnum=atoms[i]["aanum"]
        atomname=atoms[i]["pdbc"]
        atomposition=atoms[i]["pos"]
        rescale=atoms[i]["rscl"]
        x[aminoacidnum].atoms[atomname].position=atomposition
        x[aminoacidnum].atoms[atomname].properties["temperature_factor"]=rescale
    x.writeToFile(name)

#this class holds all the parameters, and important functions
class foldmain(object):
    def __init__(self):
        # Read the PDB file
        self.proteininfo={
                "name":"",
                "beta":1,
                "min_dm_err":.01,
                "dm_iters":5,
                "refine_iters":200,
                "restartevery":5000,
                "outputprogevery":99999999,
                "ermin":100,
                "refinelast":0,
                "refinenotfor":300,
                "chain_proj_iter_max":100000,
                "chain_max_step_size":1,
                "chain_min_step_size":.000001,
                "chain_min_nrg":.001,
                "pot_proj_iter_max":100000,
                "pot_max_step_size":1,
                "pot_min_step_size":.000001,
                "pot_min_nrg":-40,
                "rand_mag":1.5,
                "prg":2}

    #this function creates (and replaces) a C version of all the python lists
    def make_c_version(self):
        self.c_pro.mk_c_info(self.proteininfo)
        self.c_pro.mk_c_atmlst(self.atoms)
        self.c_pro.mk_c_bndlst(self.bonds)
        self.c_pro.mk_c_anglst(self.angles)
        self.c_pro.mk_c_detlst(self.dets)
        self.c_pro.mk_c_ramalst(self.ramas)
```

```
    #this function GETS the C version, and replaces the local python lists
    def get_c_version(self):
        self.atoms=self.c_pro.gt_c_atmlst()
        self.bonds=self.c_pro.gt_c_bndlst()
        self.angles=self.c_pro.gt_c_anglst()
        self.dets=self.c_pro.gt_c_detlst()
        self.ramas=self.c_pro.gt_c_ramalst()

    #Once a name is given for the protein (in self.proteininfo["name"])
    #This function creates all the local python lists
    def prefold(self):
        self.atoms=[]
        self.PDBtoatoms={}
        self.bonds=[]
        self.angles=[]
        self.dets=[]
        self.ramas=[]
        self.progdat=[]
        self.pdb_file=sci.Structure(self.proteininfo["name"])
        self.pdb_file.deleteHydrogens()
        #HERE IS THE initialization function
        init(self.pdb_file, self.atoms, self.PDBtoatoms, self.bonds, self.angles, self.dets, self.ramas)
        self.c_pro=c_protein()
        self.proteininfo["numaminos"]=len(self.pdb_file)
        self.proteininfo["numatoms"]=len(self.atoms)
        self.proteininfo["numbonds"]=len(self.bonds)
        self.proteininfo["numangles"]=len(self.angles)
        self.proteininfo["numdets"]=len(self.dets)
        self.proteininfo["numramas"]=len(self.ramas)
        self.make_c_version()


    #this function does "numiter" DM iterations
    def startfold(self, numiter):
        for j in range(int(numiter)):
            #do 1 DM iter, get result
            er=self.c_pro.c_DM_iter(1)
            self.get_c_version()
            self.make_c_version()
            #output progress to screen
            print "iterate ",len(self.progdat)," er=",er , "  last refined ", self.proteininfo["refinelast"]," next refine> ",self.proteininfo["refinelast"]+self.proteininfo["refinenotfor"],
" minerror=", self.proteininfo["ermin"]
            # next two for loops make new line to be plotted in progress graph
            dattmp=[]
            # first the rama angles.  the -90 and +170 are just to make the colors preferable
            for k in range(self.proteininfo["numramas"]):
                dattmp.append( ((self.ramas[k]["phi"]-90)%360-180)/180.)
                dattmp.append( ((self.ramas[k]["psi"]+170)%360-180)/180.)
                dattmp.append(-1)
                dattmp.append(-1)
            # this makes the "DM error" plot on the right
            for k in range(10):
                if er>(k+1)*3./10: dattmp.append(1)
                elif (k+1)*3./10>er>k*3./10: dattmp.append(er-k*3./10)
                elif k*3./10>er: dattmp.append(0)
            self.progdat.append(dattmp)
            # if it's time for a periodic refinement
            if er< self.proteininfo["ermin"] and len(self.progdat)>self.proteininfo["refinelast"]+self.proteininfo["refinenotfor"]:
                self.proteininfo["ermin"]=er
                self.proteininfo["refinelast"]=len(self.progdat)
                self.c_pro.c_altproj()
```

```
                self.get_c_version()
                self.make_c_version()
                poten=self.c_pro.c_get_potengrad()["en"]
                outputPDBfile(self.pdb_file, self.atoms, "outputs/foldout"+str(poten)+".pdb")
            self.proteininfo["ermin"]+=.00005
            # if DM succeeded in finding new low E state
            if er<self.proteininfo["min_dm_err"] :
                self.c_pro.c_altproj()
                poten=self.c_pro.c_get_potengrad()["en"]
                self.get_c_version()
                self.make_c_version()
                self.proteininfo["pot_min_nrg"]=poten
                outputPDBfile(self.pdb_file, self.atoms, "outputs/foldsubout"+str(poten)+".pdb")
                randomize(self.atoms, 200.)
                self.make_c_version()
                self.c_pro.c_altproj()
                self.proteininfo["ermin"]=100
                self.proteininfo["refinelast"]=len(self.progdat)
            # output the iterate every so often...
            if len(self.progdat)%self.proteininfo["outputprogevery"]==0 :
                self.get_c_version()
                self.make_c_version()
                outputPDBfile(self.pdb_file, self.atoms, "outputs/foldprog"+str(len(self.progdat))+".pdb")
            # time for a restart?
            if len(self.progdat)%self.proteininfo["restartevery"]==0 :
                self.get_c_version()
                randomize(self.atoms, 200.)
                self.make_c_version()
                self.c_pro.c_altproj()
                self.proteininfo["ermin"]=100
                self.proteininfo["refinelast"]=len(self.progdat)


#Someday this will be the beginning of a no-GUI command line version
if __name__=="__main__":
    import sys
    file_name = sys.argv[1:]
    if len(file_name)==0:
         print "Usage:"
         print "  %s [files]" % sys.argv[0]
         sys.exit(0)

    print "filename= ", sys.argv
    tst=foldmain(file_name[0])
    tst.prefold()
    tst.startfold()
```

```
import scipy
import Scientific.IO.PDB as sci
import pylab

#these small functions turn given numbers into libraries.
#The atom, bond, etc... lists are lists of libraries
def mkatom(aanum,aatp,pos,pdbcode,hydro,attp,gnptcd):
        return {"aanum":aanum,"aatp":aatp,"pos":pos,"pdbc":pdbcode,"hdro":hydro,"atp":attp,"gnptcd":gnptcd}
def mkbond(atom1,atom2,tar,er,tp,ewt):
        return {"atom1":atom1, "atom2":atom2, "tar":tar*tar, "er":er, "type":tp, "ewt":ewt}
def mkangl(atom1, atom2, atom3, tar, er, tp, ewt):
        return {"atom1":atom1, "atom2":atom2, "atom3":atom3, "tar":tar, "er":er, "type":tp, "ewt":ewt}
def mkdet(atom1, atom2, atom3, atom4, tar, er, tp, ewt):
        return {"atom1":atom1, "atom2":atom2, "atom3":atom3, "atom4":atom4, "tar":tar, "er":er, "type":tp, "ewt":ewt}
def mkrama(atom1, atom2, atom3, atom4, atom5, aanum, er, tp, score, phi, psi):
        return {"atom1":atom1, "atom2":atom2, "atom3":atom3, "atom4":atom4, "atom5":atom5, "aanum":aanum, "er":er, "type":tp, "score":score, "phi":phi, "psi":psi}

#This function is given empty lists, and the PDB atom coordinates
#From this, it fills in the atoms list, and the constraint lists
def init(x, atoms, PDBtoatoms, bonds, angles, dets, ramas):
    #n is the number of amino acids
    n=len(x)
    for i in range(n):
        PDBtoatoms[i]={}
        aminoacid=x[i].name

        #backbone atoms...
        for s in [["N",0,"N","NH"], ["CA",0,"C","CAH"], ["C",0,"C","CO"], ["O",0,"O","OC"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        #handle proline backbone separate from other amino acids
        if aminoacid=="PRO":
            #backbone bonds...
            bonds.append(mkbond(PDBtoatoms[i]["N"], PDBtoatoms[i]["CA"], 1.466, -1, "bkbn", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CA"], PDBtoatoms[i]["C"], 1.525, -1, "bkbn", -1))
            bonds.append(mkbond(PDBtoatoms[i]["C"], PDBtoatoms[i]["O"], 1.231, -1, "bkbnatt", -1))
            if not(i==0):
                bonds.append(mkbond(PDBtoatoms[i-1]["C"], PDBtoatoms[i]["N"], 1.341, -1, "bkbn", -1))
            #backbone angles...
            angles.append(mkangl(PDBtoatoms[i]["N"], PDBtoatoms[i]["CA"], PDBtoatoms[i]["C"], 1.466*1.525*scipy.cos(111.8*3.14159/180), -1, "bkbn", -1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],PDBtoatoms[i]["O"],1.525*1.231*scipy.cos(120.8*3.14159/180),-1,"bkbnatt", -1))
            if not(i==0):
                angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],1.466*1.341*scipy.cos(123.0*3.14159/180),-1,"bkbn", -1))
                angles.append(mkangl(PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],PDBtoatoms[i-1]["O"],1.341*1.231*scipy.cos(122.0*3.14159/180),-1,"bkbnatt", -1))
                angles.append(mkangl(PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],PDBtoatoms[i-1]["CA"],1.341*1.525*scipy.cos(116.9*3.14159/180),-1,"bkbn", -1))
            #backbone dets (like omega)...
            if not(i==0):
                dets.append(mkdet(PDBtoatoms[i-1]["CA"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],0,-1,"cbkbn",-1))
                dets.append(mkdet(PDBtoatoms[i-1]["CA"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],1.525*1.466*scipy.cos(6.1*3.14159/180),-1,"dbkb
n",-1))
        else:  #thie else implies amino acid is NOT proline
            #add hydrogen atom
            if i==0:
                hloc=x[i].atoms["N"].position+(.5*x[i].atoms["N"].position+.5*x[i].atoms["C"].position)-x[i].atoms["CA"].position
            else :
                hloc=x[i].atoms["N"].position+.8*(x[i-1].atoms["C"].position-x[i-1].atoms["O"].position)
            x[i].addAtom(sci.Atom("H ",hloc))
            x[i].atoms["H"].properties["element"]="H"
            x[i].atoms["H"].properties["temperature_factor"]=99
            s=["H",0,"H","H"]
```

```
            atoms.append(mkatom(i,aminoacid,hloc,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i]["H"]=len(atoms)-1
            #backbone bonds...
            bonds.append(mkbond(PDBtoatoms[i]["N"], PDBtoatoms[i]["CA"], 1.458, -1, "bkbn", -1))
            bonds.append(mkbond(PDBtoatoms[i]["N"], PDBtoatoms[i]["H"], 1.000, -1, "bkbnatt", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CA"], PDBtoatoms[i]["C"], 1.525, -1, "bkbn", -1))
            bonds.append(mkbond(PDBtoatoms[i]["C"], PDBtoatoms[i]["O"], 1.231, -1, "bkbnatt", -1))
            if not(i==0):
                bonds.append(mkbond(PDBtoatoms[i-1]["C"], PDBtoatoms[i]["N"], 1.329, -1, "bkbn", -1))
            #backbone angles...
            angles.append(mkangl(PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.458*1.525*scipy.cos(111.2*3.14159/180),-1,"bkbn",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],PDBtoatoms[i]["O"],1.525*1.231*scipy.cos(120.8*3.14159/180),-1,"bkbnatt",-1))
            angles.append(mkangl(PDBtoatoms[i]["H"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],1.000*1.458*scipy.cos(119.2*3.14159/180),-1,"bkbnatt",-1))
            if not(i==0):
                angles.append(mkangl(PDBtoatoms[i]["H"],PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],1.000*1.329*scipy.cos(119.1*3.14159/180),-1,"bkbnatt",-1))
                angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],1.458*1.329*scipy.cos(121.7*3.14159/180),-1,"bkbn",-1))
                angles.append(mkangl(PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],PDBtoatoms[i-1]["O"],1.329*1.231*scipy.cos(123.0*3.14159/180),-1,"bkbnatt",-1))
                angles.append(mkangl(PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],PDBtoatoms[i-1]["CA"],1.329*1.525*scipy.cos(116.2*3.14159/180),-1,"bkbn",-1))
            #backbone dets (like omega)...
            if not(i==0):
                dets.append(mkdet(PDBtoatoms[i-1]["CA"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],0,-1,"cbkbn",-1))
                dets.append(mkdet(PDBtoatoms[i-1]["CA"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],1.525*1.458*scipy.cos(5.5*3.14159/180),-1,"dbkb
n",-1))
                dets.append(mkdet(PDBtoatoms[i-1]["O"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["H"],0,-1,"catt",-1))
                dets.append(mkdet(PDBtoatoms[i-1]["O"],PDBtoatoms[i-1]["C"],PDBtoatoms[i]["N"],PDBtoatoms[i]["H"],1.231*1.000*scipy.cos(3.9*3.14159/180),-1,"datt",-
1))
        #add ramachandrin constraint.
        ramas.append(mkrama(-1,PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],-1,i,0,"stndrd",0,"phi","psi"))
        if not(i==0):
            ramas[len(ramas)-1]["atom1"]=PDBtoatoms[i-1]["C"]
            ramas[len(ramas)-2]["atom5"]=PDBtoatoms[i]["N"]
        if i==0:
            ramas[len(ramas)-1]["type"]="frst"
        if i==n-1:
            ramas[len(ramas)-1]["type"]="lst"

        #sidegroup atoms, bonds...
        if aminoacid=="ALA":
            for s in [
                        ["CB",1,"C","CH3"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.521, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.521*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.521*scipy.cos(110.4*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.477772,-1,"cbdet",-1))
        elif aminoacid=="ARG":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",1,"C","CH2"],
                        ["CD",1,"C","CR2"],
                        ["NE",3,"N","NR1"],
                        ["CZ",1,"C","CR3"],
                        ["NH1",3,"N","NR2"],
                        ["NH2",3,"N","NR2"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.520, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD"], PDBtoatoms[i]["CG"], 1.520, -1, "rgrp", -1))
```

```
            bonds.append(mkbond(PDBtoatoms[i]["NE"], PDBtoatoms[i]["CD"], 1.460, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["NE"], 1.329, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["NH1"], 1.326, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["NH2"], 1.326, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.520*scipy.cos(114.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],1.520*1.520*scipy.cos(111.3*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["NE"],1.520*1.460*scipy.cos(112.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["NE"],PDBtoatoms[i]["CZ"],1.460*1.329*scipy.cos(124.2*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["NE"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["NH1"],1.329*1.326*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["NE"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["NH2"],1.329*1.326*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["NH1"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["NH2"],1.326*1.326*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="ASN":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",1,"C","CO"],
                        ["OD1",3,"O","OC"],
                        ["ND2",3,"N","NAS"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.516, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OD1"], PDBtoatoms[i]["CG"], 1.231, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["ND2"], PDBtoatoms[i]["CG"], 1.328, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.516*scipy.cos(112.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["OD1"],1.231*1.516*scipy.cos(120.8*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["ND2"],1.328*1.516*scipy.cos(116.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["OD1"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["ND2"],1.328*1.231*scipy.cos(122.6*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="ASP":
            for s in [
                        ["CB",1,"C","CH2B"],
                        ["CG",1,"C","CX1"],
                        ["OD1",3,"O","OX1"],
                        ["OD2",3,"O","OX1"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.516, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OD1"], PDBtoatoms[i]["CG"], 1.249, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OD2"], PDBtoatoms[i]["CG"], 1.249, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.516*scipy.cos(112.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["OD1"],1.249*1.516*scipy.cos(118.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["OD2"],1.249*1.516*scipy.cos(118.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["OD1"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["OD2"],1.249*1.249*scipy.cos(122.9*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="CYS":
            for s in [
                        ["CB",1,"C","CH2C"],
                        ["SG",4,"S","SH"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CA"], PDBtoatoms[i]["CB"], 1.530, -1, "rgrp", -1))
```

```
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["SG"], 1.808, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["SG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.808*scipy.cos(114.4*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="GLN":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",1,"C","CH2"],
                        ["CD",1,"O","CO"],
                        ["OE1",3,"O","OC"],
                        ["NE2",3,"N","NAS"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.520, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD"], 1.516, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OE1"], PDBtoatoms[i]["CD"], 1.231, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["NE2"], PDBtoatoms[i]["CD"], 1.328, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.520*scipy.cos(114.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],1.516*1.520*scipy.cos(112.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["OE1"],1.231*1.516*scipy.cos(120.8*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["NE2"],1.328*1.516*scipy.cos(116.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["OE1"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["NE2"],1.328*1.231*scipy.cos(122.6*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="GLU":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",1,"C","CH2B"],
                        ["CD",1,"O","CX1"],
                        ["OE1",3,"O","OX1"],
                        ["OE2",3,"O","OX1"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.520, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD"], 1.516, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OE1"], PDBtoatoms[i]["CD"], 1.249, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["OE2"], PDBtoatoms[i]["CD"], 1.249, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.520*scipy.cos(114.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],1.516*1.520*scipy.cos(112.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["OE1"],1.249*1.516*scipy.cos(118.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["OE2"],1.249*1.516*scipy.cos(118.4*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["OE1"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["OE2"],1.249*1.249*scipy.cos(122.9*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="GLY":
            ramas[len(ramas)-1]["type"]="gly"
        elif aminoacid=="HIS":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",2,"C","CGHP"],
                        ["CD2",2,"C","CGHP"],
                        ["ND1",3,"N","NDHP"],
                        ["NE2",3,"N","NDHP"],
                        ["CE1",2,"C","CHEP"]]:
                atompos=x[i].atoms[s[0]].position
```

```
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.497, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD2"], 1.353, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CD2"], PDBtoatoms[i]["NE2"], 1.353, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["NE2"], PDBtoatoms[i]["CE1"], 1.353, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CE1"], PDBtoatoms[i]["ND1"], 1.353, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["ND1"], PDBtoatoms[i]["CG"], 1.353, -1, "rgrp", -1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.497*scipy.cos(113.8*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["ND1"],1.497*1.353*scipy.cos(126.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],1.497*1.353*scipy.cos(126.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["NE2"],1.353*1.353*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["NE2"],PDBtoatoms[i]["CE1"],1.353*1.353*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["NE2"],PDBtoatoms[i]["CE1"],PDBtoatoms[i]["ND1"],1.353*1.353*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CE1"],PDBtoatoms[i]["ND1"],PDBtoatoms[i]["CG"],1.353*1.353*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["ND1"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],1.353*1.353*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["NE2"],0,-1,"crgrp",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["ND1"],PDBtoatoms[i]["CE1"],0,-1,"crgrp",-1))
    elif aminoacid=="ILE":
        for s in [
                    ["CB",1,"C","CH2"],
                    ["CG1",1,"C","CH2"],
                    ["CG2",1,"C","CH3"],
                    ["CD1",1,"C","CH3"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.540, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG1"], PDBtoatoms[i]["CB"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG2"], PDBtoatoms[i]["CB"], 1.521, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG1"], PDBtoatoms[i]["CD1"], 1.513, -1, "rgrp", -1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.540*scipy.cos(109.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.540*scipy.cos(111.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG2"],1.540*1.521*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG1"],1.540*1.530*scipy.cos(110.4*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CG2"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG1"],1.521*1.530*scipy.cos(110.7*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG1"],PDBtoatoms[i]["CD1"],1.513*1.530*scipy.cos(113.8*3.14159/180),-1,"rgrp",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.51876,-1,"cbdet",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["CG2"],PDBtoatoms[i]["CG1"],2.64475,-1,"cbdet",-1))
    elif aminoacid=="LEU":
        for s in [
                    ["CB",1,"C","CH2"],
                    ["CG",1,"C","CH2"],
                    ["CD2",1,"C","CH3"],
                    ["CD1",1,"C","CH3"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD2"], 1.521, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD1"], 1.521, -1, "rgrp", -1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.530*scipy.cos(116.3*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.530*1.521*scipy.cos(110.7*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],1.530*1.521*scipy.cos(110.7*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],1.521*1.521*scipy.cos(110.8*3.14159/180),-1,"rgrp",-1))
```

```
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
    elif aminoacid=="LYS":
        for s in [
                    ["CB",1,"C","CH2"],
                    ["CG",1,"C","CH2"],
                    ["CD",1,"C","CH2"],
                    ["CE",1,"C","CH2K"],
                    ["NZ",3,"N","NX"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.520, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CD"], PDBtoatoms[i]["CG"], 1.520, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CE"], PDBtoatoms[i]["CD"], 1.520, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["NZ"], PDBtoatoms[i]["CE"], 1.489, -1, "rgrp", -1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.520*scipy.cos(114.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],1.520*1.520*scipy.cos(111.3*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["CE"],1.520*1.520*scipy.cos(111.3*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["CE"],PDBtoatoms[i]["NZ"],1.520*1.489*scipy.cos(111.9*3.14159/180),-1,"rgrp",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
    elif aminoacid=="MET":
        for s in [
                    ["CB",1,"C","CH2"],
                    ["CG",1,"C","CH2M"],
                    ["SD",4,"S","SM"],
                    ["CE",1,"C","CH3M"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.520, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["SD"], PDBtoatoms[i]["CG"], 1.803, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CE"], PDBtoatoms[i]["SD"], 1.791, -1, "rgrp", -1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.520*scipy.cos(114.1*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["SD"],1.520*1.803*scipy.cos(112.7*3.14159/180),-1,"rgrp",-1))
        angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["SD"],PDBtoatoms[i]["CE"],1.803*1.791*scipy.cos(100.9*3.14159/180),-1,"rgrp",-1))
        dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
    elif aminoacid=="PHE":
        for s in [
                    ["CB",1,"C","CH2"],
                    ["CG",2,"C","CFH"],
                    ["CD1",2,"C","CFH"],
                    ["CD2",2,"C","CFH"],
                    ["CE1",2,"C","CFH"],
                    ["CE2",2,"C","CFH"],
                    ["CZ",2,"C","CFH"]]:
            atompos=x[i].atoms[s[0]].position
            atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
            PDBtoatoms[i][s[0]]=len(atoms)-1
        bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.502, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD1"], 1.383, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CD1"], PDBtoatoms[i]["CE1"], 1.383, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CE1"], PDBtoatoms[i]["CZ"], 1.383, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["CE2"], 1.383, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CE2"], PDBtoatoms[i]["CD2"], 1.383, -1, "rgrp", -1))
        bonds.append(mkbond(PDBtoatoms[i]["CD2"], PDBtoatoms[i]["CG"], 1.383, -1, "rgrp", -1))
```

```
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.502*scipy.cos(113.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.502*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CE1"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CZ"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CE2"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],1.383*1.502*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],0,-1,"crgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CG"],0,-1,"crgrp",-1))
        elif aminoacid=="PRO":
            for s in [
                        ["CB",1,"C","CH2E"],
                        ["CG",1,"C","CH2P"],
                        ["CD",1,"C","CH2P"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.492, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD"], 1.503, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD"], PDBtoatoms[i]["N"], 1.473, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(103.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.492*1.530*scipy.cos(106.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],1.492*1.503*scipy.cos(107.7*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD"],PDBtoatoms[i]["N"],1.473*1.503*scipy.cos(104.8*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],1.473*1.466*scipy.cos(112.0*3.14159/180),-1,"rgrp",-1))
            if not(i==0):
                    angles.append(mkangl(PDBtoatoms[i]["CD"],PDBtoatoms[i]["N"],PDBtoatoms[i-1]["C"],1.473*1.341*scipy.cos(125.0*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.729635,-1,"cbdet",-1))
            if not(i==0):
                        dets.append(mkdet(PDBtoatoms[i]["CD"],PDBtoatoms[i]["N"],PDBtoatoms[i]["CA"],PDBtoatoms[i-1]["C"],0,-1,"catt",-1))
            ramas[len(ramas)-1]["type"]="pro"
        elif aminoacid=="SER":
            for s in [
                        ["CB",1,"C","CH2S"],
                        ["OG",3,"O","OH"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CA"], PDBtoatoms[i]["CB"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["OG"], 1.417, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["OG"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],1.530*1.417*scipy.cos(111.1*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
        elif aminoacid=="THR":
            for s in [
                        ["CB",1,"C","CH2S"],
                        ["OG1",3,"O","OH"],
                        ["CG2",1,"C","CH3"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.540, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["OG1"], 1.433, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CG2"], 1.521, -1, "rgrp", -1))
```

```
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.540*scipy.cos(109.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.540*scipy.cos(111.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG2"],1.540*1.521*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["OG1"],1.540*1.433*scipy.cos(109.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG2"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["OG1"],1.521*1.433*scipy.cos(109.3*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.51876,-1,"cbdet",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["CG2"],PDBtoatoms[i]["OG1"],2.551736,-1,"cbdet",-1))
        elif aminoacid=="TRP":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",1,"C","CGTR"],
                        ["CD1",1,"C","CHTR"],
                        ["CD2",2,"C","CGTR"],
                        ["CE2",2,"C","CHTR"],
                        ["NE1",3,"N","NDHS"],
                        ["CZ2",2,"C","CFH"],
                        ["CE3",2,"C","CFH"],
                        ["CH2",2,"C","CFH"],
                        ["CZ3",2,"C","CFH"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CG"], 1.498, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD1"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD1"], PDBtoatoms[i]["NE1"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["NE1"], PDBtoatoms[i]["CE2"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CE2"], PDBtoatoms[i]["CD2"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD2"], PDBtoatoms[i]["CG"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD2"], PDBtoatoms[i]["CE3"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CE3"], PDBtoatoms[i]["CZ3"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ3"], PDBtoatoms[i]["CH2"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CH2"], PDBtoatoms[i]["CZ2"], 1.389, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ2"], PDBtoatoms[i]["CE2"], 1.389, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.498*scipy.cos(113.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],1.389*1.498*scipy.cos(126.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.389*1.498*scipy.cos(126.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],PDBtoatoms[i]["NE1"],1.389*1.389*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD1"],PDBtoatoms[i]["NE1"],PDBtoatoms[i]["CE2"],1.389*1.389*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["NE1"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],1.389*1.389*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],1.389*1.389*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.389*1.389*scipy.cos(108.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CE3"],PDBtoatoms[i]["CZ3"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE3"],PDBtoatoms[i]["CZ3"],PDBtoatoms[i]["CH2"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CZ3"],PDBtoatoms[i]["CH2"],PDBtoatoms[i]["CZ2"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CH2"],PDBtoatoms[i]["CZ2"],PDBtoatoms[i]["CE2"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CZ2"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CE3"],1.389*1.389*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CE3"],1.389*1.389*scipy.cos(132.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["NE1"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CZ2"],1.389*1.389*scipy.cos(132.0*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CD2"],0,-1,"crgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CE3"],0,-1,"crgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["NE1"],PDBtoatoms[i]["CZ2"],0,-1,"crgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CZ2"],PDBtoatoms[i]["CH2"],PDBtoatoms[i]["CZ3"],PDBtoatoms[i]["CE3"],0,-1,"crgrp",-1))
        elif aminoacid=="TYR":
            for s in [
                        ["CB",1,"C","CH2"],
                        ["CG",2,"C","CFH"],
                        ["CD1",2,"C","CFH"],
```

```
                    ["CD2",2,"C","CFH"],
                    ["CE1",2,"C","CFH"],
                    ["CE2",2,"C","CFH"],
                    ["CZ",2,"C","CZ"],
                    ["OH",3,"O","OH"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.530, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CB"], 1.502, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CG"], PDBtoatoms[i]["CD1"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD1"], PDBtoatoms[i]["CE1"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CE1"], PDBtoatoms[i]["CZ"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["CE2"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CE2"], PDBtoatoms[i]["CD2"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CD2"], PDBtoatoms[i]["CG"], 1.383, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CZ"], PDBtoatoms[i]["OH"], 1.376, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.530*scipy.cos(110.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.530*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],1.530*1.502*scipy.cos(113.6*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.502*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CE1"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CZ"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CE2"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],1.383*1.383*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CD2"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CB"],1.383*1.502*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE2"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["OH"],1.383*1.376*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["OH"],1.383*1.376*scipy.cos(120.0*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.503196,-1,"cbdet",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG"],PDBtoatoms[i]["CD1"],PDBtoatoms[i]["CD2"],0,-1,"crgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["OH"],PDBtoatoms[i]["CZ"],PDBtoatoms[i]["CE1"],PDBtoatoms[i]["CE2"],0,-1,"crgrp",-1))
        elif aminoacid=="VAL":
            for s in [
                    ["CB",1,"C","CH2"],
                    ["CG1",1,"C","CH3"],
                    ["CG2",1,"C","CH3"]]:
                atompos=x[i].atoms[s[0]].position
                atoms.append(mkatom(i,aminoacid,atompos,s[0],s[1],s[2],s[3]))
                PDBtoatoms[i][s[0]]=len(atoms)-1
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CA"], 1.540, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CG1"], 1.521, -1, "rgrp", -1))
            bonds.append(mkbond(PDBtoatoms[i]["CB"], PDBtoatoms[i]["CG2"], 1.521, -1, "rgrp", -1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["C"],1.525*1.540*scipy.cos(109.1*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],1.458*1.540*scipy.cos(111.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG2"],1.540*1.521*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CA"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG1"],1.540*1.521*scipy.cos(110.5*3.14159/180),-1,"rgrp",-1))
            angles.append(mkangl(PDBtoatoms[i]["CG2"],PDBtoatoms[i]["CB"],PDBtoatoms[i]["CG1"],1.521*1.521*scipy.cos(110.8*3.14159/180),-1,"rgrp",-1))
            dets.append(mkdet(PDBtoatoms[i]["CB"],PDBtoatoms[i]["CA"],PDBtoatoms[i]["N"],PDBtoatoms[i]["C"],2.51876,-1,"cbdet",-1))
        else :
            print "amino acid type unknown!!! %s" % aminoacid
################################################
    #radii, charge, and re-scale initial values:
    for i in range(len(atoms)):
        if atoms[i]["atp"]=="C":
            if atoms[i]["hdro"]==1: atoms[i]["rad"]=2.06
            elif atoms[i]["hdro"]==2: atoms[i]["rad"]=1.86
            elif atoms[i]["hdro"]==0: atoms[i]["rad"]=1.9
            else : print "hydro type unknown in radii init!!! %d" % atoms[i]["hdro"]
        elif atoms[i]["atp"]=="N": atoms[i]["rad"]=1.76
        elif atoms[i]["atp"]=="O": atoms[i]["rad"]=1.58
```

```
        elif atoms[i]["atp"]=="H": atoms[i]["rad"]=.85
        elif atoms[i]["atp"]=="S": atoms[i]["rad"]=2.07
        else : print "ATOM type unknown in radii init!!! %d" % atoms[i]["atp"]

        if atoms[i]["pdbc"]=="O" : atoms[i]["chrg"]=-.6
        elif atoms[i]["pdbc"]=="C" : atoms[i]["chrg"]=.6
        elif atoms[i]["pdbc"]=="N" : atoms[i]["chrg"]=-.4
        elif atoms[i]["pdbc"]=="H" : atoms[i]["chrg"]=.4
        else :  atoms[i]["chrg"]=0

        atoms[i]["rscl"]=1
##############################################
    #make bonded list; who is bonded to atom i?
    #not just who is bond to atom i, who should atom i ignore in the potential function...
    for i in range(len(atoms)):
        atoms[i]["bonded"]=[]
        atoms[i]["numbonded"]=0
    for i in range(len(bonds)):
        a1=bonds[i]["atom1"]
        a2=bonds[i]["atom2"]
        if not(a2 in atoms[a1]["bonded"]):
            atoms[a1]["bonded"].append(a2)
            atoms[a1]["numbonded"]+=1
        if not(a1 in atoms[a2]["bonded"]):
            atoms[a2]["bonded"].append(a1)
            atoms[a2]["numbonded"]+=1
    for i in range(len(angles)):
        a1=angles[i]["atom1"]
        a2=angles[i]["atom2"]
        a3=angles[i]["atom3"]
        if not(a2 in atoms[a1]["bonded"]):
            atoms[a1]["bonded"].append(a2)
            atoms[a1]["numbonded"]+=1
        if not(a3 in atoms[a1]["bonded"]):
            atoms[a1]["bonded"].append(a3)
            atoms[a1]["numbonded"]+=1
        if not(a1 in atoms[a2]["bonded"]):
            atoms[a2]["bonded"].append(a1)
            atoms[a2]["numbonded"]+=1
        if not(a3 in atoms[a2]["bonded"]):
            atoms[a2]["bonded"].append(a3)
            atoms[a2]["numbonded"]+=1
        if not(a1 in atoms[a3]["bonded"]):
            atoms[a3]["bonded"].append(a1)
            atoms[a3]["numbonded"]+=1
        if not(a2 in atoms[a3]["bonded"]):
            atoms[a3]["bonded"].append(a2)
            atoms[a3]["numbonded"]+=1
##############################################
    #determine initial values
    for i in range(len(atoms)):
        atoms[i]["rscl"]=1
    for i in range(len(bonds)):
        a1=atoms[bonds[i]["atom1"]]["pos"]
        a2=atoms[bonds[i]["atom2"]]["pos"]
        v1=a1-a2
        bonds[i]["act"]=v1*v1
        bonds[i]["er"]=v1*v1-bonds[i]["tar"]
    for i in range(len(angles)):
        a1=atoms[angles[i]["atom1"]]["pos"]
        a2=atoms[angles[i]["atom2"]]["pos"]
        a3=atoms[angles[i]["atom3"]]["pos"]
```

```
        v1=a2-a1
        v2=a2-a3
        angles[i]["act"]=v1*v2
        angles[i]["er"]=v1*v2-angles[i]["tar"]
    for i in range(len(dets)):
        a1=atoms[dets[i]["atom1"]]["pos"]
        a2=atoms[dets[i]["atom2"]]["pos"]
        a3=atoms[dets[i]["atom3"]]["pos"]
        a4=atoms[dets[i]["atom4"]]["pos"]
        v1=a1-a2
        v2=a2-a3
        v3=a3-a4
        if dets[i]["type"] in ["catt", "cbkbn", "crgrp"]:
            dets[i]["act"]=v1*sci.Vector.cross(v2,v3)
            dets[i]["er"]=v1*sci.Vector.cross(v2,v3)-dets[i]["tar"]
        if dets[i]["type"] in ["datt", "dbkbn"]:
            dets[i]["act"]=v1*v3
            dets[i]["er"]=v1*v3-dets[i]["tar"]
        if dets[i]["type"] in ["cbdet"]:
            dets[i]["act"]=v1*sci.Vector.cross(v2,v3)
            dets[i]["er"]=v1*sci.Vector.cross(v2,v3)-dets[i]["tar"]
    for i in range(len(ramas)):
        a2=atoms[ramas[i]["atom2"]]["pos"]
        a3=atoms[ramas[i]["atom3"]]["pos"]
        a4=atoms[ramas[i]["atom4"]]["pos"]
        if not(ramas[i]["type"]=="frst"):
            #phi calculation
            a1=atoms[ramas[i]["atom1"]]["pos"]
            v1=a2-a1
            v2=a3-a2
            v3=a4-a3
            A=scipy.sqrt((sci.Vector.cross(v1,v2)*sci.Vector.cross(v1,v2))*(sci.Vector.cross(v3,v2)*sci.Vector.cross(v3,v2)))
            sinphi=v2*sci.Vector.cross(v3,v1)*scipy.sqrt(v2*v2)/A
            cosphi=((v1*v2)*(v2*v3)-(v1*v3)*(v2*v2))*1./A
            ramas[i]["phi"]=scipy.arctan2(sinphi, cosphi)*180/3.14159265
            ramas[i]["sinphi"]=sinphi
            ramas[i]["cosphi"]=cosphi
            ramas[i]["Aphi"]=A
        else :
            ramas[i]["phi"]=0
            ramas[i]["sinphi"]=0
            ramas[i]["cosphi"]=0
            ramas[i]["Aphi"]=1
        if not(ramas[i]["type"]=="lst"):
            #psi calculation
            a5=atoms[ramas[i]["atom5"]]["pos"]
            v1=a3-a2
            v2=a4-a3
            v3=a5-a4
            A=scipy.sqrt((sci.Vector.cross(v1,v2)*sci.Vector.cross(v1,v2))*(sci.Vector.cross(v3,v2)*sci.Vector.cross(v3,v2)))
            sinpsi=v2*sci.Vector.cross(v3,v1)*scipy.sqrt(v2*v2)/A
            cospsi=((v1*v2)*(v2*v3)-(v1*v3)*(v2*v2))*1./A
            ramas[i]["psi"]=scipy.arctan2(sinpsi, cospsi)*180/3.14159265
            ramas[i]["sinpsi"]=sinpsi
            ramas[i]["cospsi"]=cospsi
            ramas[i]["Apsi"]=A
        else :
            ramas[i]["psi"]=0
            ramas[i]["sinpsi"]=0
            ramas[i]["cospsi"]=0
            ramas[i]["Apsi"]=1
################################################
```

```
    #determine energy weights...
    bond_ewt_dct={"bkbn":4, "bkbnatt":2, "rgrp":1}
    angl_ewt_dct={"bkbn":2, "bkbnatt":1, "rgrp":.5}
    det_ewt_dct={"datt":1.5, "catt":.2, "dbkbn":1.5, "cbkbn":.2, "cbdet":1., "crgrp":.1}
    for i in range(len(bonds)):
        bonds[i]["ewt"]=bond_ewt_dct[bonds[i]["type"]]
    for i in range(len(angles)):
        angles[i]["ewt"]=angl_ewt_dct[angles[i]["type"]]
    for i in range(len(dets)):
        dets[i]["ewt"]=det_ewt_dct[dets[i]["type"]]
```

```
import Scientific.IO.PDB
from Scientific.IO.PDB import Vector

cdef extern from "stdlib.h":
    void *malloc(int)
    void free(void *)

#this structure and function, used in this file below, are found in "vector.h"
cdef extern from "vector.h":
    ctypedef struct c_vector "fvector":
        double x
        double y
        double z
    cdef extern c_vector c_Vectormkr "Vectormkr" (double xc, double yc, double zc)

#these structures are actually in "structures.h", but included in "c_code.h"
cdef extern from "c_code.h":
    ctypedef struct c_proteininfo "proteininfo":
        double beta
        double min_dm_err
        int dm_iters
        int refine_iters
        int restartevery
        int outputprogevery
        int numaminos
        int numatoms
        int numbonds
        double bond_er
        int numangles
        double angle_er
        int numdets
        double det_er
        int numramas
        double rama_er
        int chain_proj_iter_max
        double chain_max_step_size
        double chain_min_step_size
        double chain_min_nrg
        int pot_proj_iter_max
        double pot_max_step_size
        double pot_min_step_size
        double pot_min_nrg
        int prg

    ctypedef struct c_atom "atom":
        int aminoacidnum
        c_vector r
        char *atomtype
        char *genpotcode
        char *pdbcode
        char *aminoacidtype
        int hydro
        double rscl
        double radius
        double charge
        int num_nghbrs
        int *nghbrs
        int num_bonded
        int *bonded
    ctypedef struct c_bond "bond":
        int atom1
```

```
        int atom2
        double tar
        double act
        char *type
        double er
        double ewt
    ctypedef struct c_angl "angl":
        int atom1
        int atom2
        int atom3
        double tar
        double act
        char *type
        double er
        double ewt
    ctypedef struct c_det "det":
        int atom1
        int atom2
        int atom3
        int atom4
        double tar
        double act
        char *type
        double er
        double ewt
    ctypedef struct c_rama "rama":
        int atom1
        int atom2
        int atom3
        int atom4
        int atom5
        int aminoacidnum
        char *type
        double phi
        double psi
        double sinphi
        double cosphi
        double Aphi
        double sinpsi
        double cospsi
        double Apsi
        double score
        double er

    #These functions are prototyped in "c_code.h", and again here so Python can use them
    cdef extern double c_get_chainengrad "get_chainengrad" (c_atom *, c_bond *, c_angl *, c_det *, c_rama *, c_proteininfo, c_vector *)
    cdef extern int c_proj_chain "proj_chain" (c_atom *, c_bond *, c_angl *, c_det *, c_rama *, c_proteininfo)
    cdef extern double c_get_potengrad "get_potengrad" (c_atom *, c_proteininfo, c_vector *)
    cdef extern int c_proj_pot "proj_pot" (c_atom *, c_proteininfo)

    cdef extern void alt_proj "alt_proj" (c_atom *, c_bond *, c_angl *, c_det *, c_rama *, c_proteininfo)
    cdef extern double DM_iter "DM_iter" (c_atom *, c_bond *, c_angl *, c_det *, c_rama *, c_proteininfo)


#This structure contains function callable by python,
#while these functions call pre-compiled C functions
cdef class c_protein:
    cdef c_proteininfo info
    cdef c_bond *c_bonds
    cdef c_atom *c_atoms
    cdef c_angl *c_angls
    cdef c_det *c_dets
```

```
    cdef c_rama *c_ramas

    #make a c version of the python library, foldmain.proteininfo
    def mk_c_info(self, infolist):
        self.info.beta=infolist["beta"]
        self.info.min_dm_err=infolist["min_dm_err"]
        self.info.dm_iters=infolist["dm_iters"]
        self.info.refine_iters=infolist["refine_iters"]
        self.info.restartevery=infolist["restartevery"]
        self.info.outputprogevery=infolist["outputprogevery"]
        self.info.numaminos=infolist["numaminos"]
        self.info.numatoms=infolist["numatoms"]
        self.info.numbonds=infolist["numbonds"]
        self.info.bond_er=-1
        self.info.numangles=infolist["numangles"]
        self.info.angle_er=-1
        self.info.numdets=infolist["numdets"]
        self.info.det_er=-1
        self.info.numramas=infolist["numramas"]
        self.info.rama_er=-1
        self.info.chain_proj_iter_max=infolist["chain_proj_iter_max"]
        self.info.chain_max_step_size=infolist["chain_max_step_size"]
        self.info.chain_min_step_size=infolist["chain_min_step_size"]
        self.info.chain_min_nrg=infolist["chain_min_nrg"]
        self.info.pot_proj_iter_max=infolist["pot_proj_iter_max"]
        self.info.pot_max_step_size=infolist["pot_max_step_size"]
        self.info.pot_min_step_size=infolist["pot_min_step_size"]
        self.info.pot_min_nrg=infolist["pot_min_nrg"]
        self.info.prg=infolist["prg"]

    #make a C version of the python list foldmain.atoms
    def mk_c_atmlst(self, pyatomlist):
        n=len(pyatomlist)
        self.info.numatoms=n
        self.c_atoms = <c_atom *> malloc(n * sizeof(c_atom))
        for i in range(n):
            self.c_atoms[i].aminoacidnum=pyatomlist[i]["aanum"]
            self.c_atoms[i].aminoacidtype=pyatomlist[i]["aatp"]
            self.c_atoms[i].r=c_Vectormkr(pyatomlist[i]["pos"][0],pyatomlist[i]["pos"][1],pyatomlist[i]["pos"][2])
            self.c_atoms[i].pdbcode=pyatomlist[i]["pdbc"]
            self.c_atoms[i].hydro=pyatomlist[i]["hdro"]
            self.c_atoms[i].charge=pyatomlist[i]["chrg"]
            self.c_atoms[i].rscl=pyatomlist[i]["rscl"]
            self.c_atoms[i].radius=pyatomlist[i]["rad"]
            self.c_atoms[i].atomtype=pyatomlist[i]["atp"]
            self.c_atoms[i].genpotcode=pyatomlist[i]["gnptcd"]
            self.c_atoms[i].num_nghbrs=0
            self.c_atoms[i].nghbrs= <int *> malloc(n * sizeof(int))
            self.c_atoms[i].num_bonded=pyatomlist[i]["numbonded"]
            self.c_atoms[i].bonded= <int *> malloc(pyatomlist[i]["numbonded"] * sizeof(int))
            for j in range(pyatomlist[i]["numbonded"]):
                self.c_atoms[i].bonded[j]=pyatomlist[i]["bonded"][j]

    #Get the C version atom list, return it
    def gt_c_atmlst(self):
        atmout=[]
        for i in range(self.info.numatoms):
            atmtmp={}
            atmtmp["aanum"]=self.c_atoms[i].aminoacidnum
            atmtmp["aatp"]=self.c_atoms[i].aminoacidtype
            atmtmp["pos"]=Vector(self.c_atoms[i].r.x, self.c_atoms[i].r.y, self.c_atoms[i].r.z)
            atmtmp["pdbc"]=self.c_atoms[i].pdbcode
```

```
            atmtmp["hdro"]=self.c_atoms[i].hydro
            atmtmp["chrg"]=self.c_atoms[i].charge
            atmtmp["rscl"]=self.c_atoms[i].rscl
            atmtmp["rad"]=self.c_atoms[i].radius
            atmtmp["atp"]=self.c_atoms[i].atomtype
            atmtmp["gnptcd"]=self.c_atoms[i].genpotcode
            atmtmp["num_nghbrs"]=self.c_atoms[i].num_nghbrs
            free(self.c_atoms[i].nghbrs)
            atmtmp["numbonded"]=self.c_atoms[i].num_bonded
            bondedtmp=[]
            for j in range( self.c_atoms[i].num_bonded ):
                bondedtmp.append(self.c_atoms[i].bonded[j])
            free(self.c_atoms[i].bonded)
            atmtmp["bonded"]=bondedtmp
            atmout.append(atmtmp)
        free(self.c_atoms)
        return atmout

    #make a C version of the python list foldmain.bonds
    def mk_c_bndlst(self, pybondlist):
        n=len(pybondlist)
        self.info.numbonds=n
        self.c_bonds = <c_bond *> malloc(n * sizeof(c_bond))
        for i in range(n):
            self.c_bonds[i].atom1=pybondlist[i]["atom1"]
            self.c_bonds[i].atom2=pybondlist[i]["atom2"]
            self.c_bonds[i].tar=pybondlist[i]["tar"]
            self.c_bonds[i].act=pybondlist[i]["act"]
            self.c_bonds[i].er=pybondlist[i]["er"]
            self.c_bonds[i].type=pybondlist[i]["type"]
            self.c_bonds[i].ewt=pybondlist[i]["ewt"]

    #GET the C version of the bond list, return it
    def gt_c_bndlst(self):
        bndout=[]
        for i in range(self.info.numbonds):
            bndtmp={}
            bndtmp["atom1"]=self.c_bonds[i].atom1
            bndtmp["atom2"]=self.c_bonds[i].atom2
            bndtmp["tar"]=self.c_bonds[i].tar
            bndtmp["act"]=self.c_bonds[i].act
            bndtmp["er"]=self.c_bonds[i].er
            bndtmp["type"]=self.c_bonds[i].type
            bndtmp["ewt"]=self.c_bonds[i].ewt
            bndout.append(bndtmp)
        free(self.c_bonds)
        return bndout

    #make a C version of the python list foldmain.angles
    def mk_c_anglst(self, pyangllist):
        n=len(pyangllist)
        self.info.numangles=n
        self.c_angls = <c_angl *> malloc(n * sizeof(c_angl))
        for i in range(n):
            self.c_angls[i].atom1=pyangllist[i]["atom1"]
            self.c_angls[i].atom2=pyangllist[i]["atom2"]
            self.c_angls[i].atom3=pyangllist[i]["atom3"]
            self.c_angls[i].tar=pyangllist[i]["tar"]
            self.c_angls[i].act=pyangllist[i]["act"]
            self.c_angls[i].er=pyangllist[i]["er"]
            self.c_angls[i].type=pyangllist[i]["type"]
            self.c_angls[i].ewt=pyangllist[i]["ewt"]
```

```
    #GET the C version of the angle list, return it
    def gt_c_anglst(self):
        angout=[]
        for i in range(self.info.numangles):
            angtmp={}
            angtmp["atom1"]=self.c_angls[i].atom1
            angtmp["atom2"]=self.c_angls[i].atom2
            angtmp["atom3"]=self.c_angls[i].atom3
            angtmp["tar"]=self.c_angls[i].tar
            angtmp["act"]=self.c_angls[i].act
            angtmp["er"]=self.c_angls[i].er
            angtmp["type"]=self.c_angls[i].type
            angtmp["ewt"]=self.c_angls[i].ewt
            angout.append(angtmp)
        free(self.c_angls)
        return angout

    #make a C version of the python list foldmain.dets
    def mk_c_detlst(self, pydetlist):
        n=len(pydetlist)
        self.info.numdets=n
        self.c_dets = <c_det *> malloc(n * sizeof(c_det))
        for i in range(n):
            self.c_dets[i].atom1=pydetlist[i]["atom1"]
            self.c_dets[i].atom2=pydetlist[i]["atom2"]
            self.c_dets[i].atom3=pydetlist[i]["atom3"]
            self.c_dets[i].atom4=pydetlist[i]["atom4"]
            self.c_dets[i].tar=pydetlist[i]["tar"]
            self.c_dets[i].act=pydetlist[i]["act"]
            self.c_dets[i].er=pydetlist[i]["er"]
            self.c_dets[i].type=pydetlist[i]["type"]
            self.c_dets[i].ewt=pydetlist[i]["ewt"]

    #Get the C version of the determinate list, return it
    def gt_c_detlst(self):
        detout=[]
        for i in range(self.info.numdets):
            dettmp={}
            dettmp["atom1"]=self.c_dets[i].atom1
            dettmp["atom2"]=self.c_dets[i].atom2
            dettmp["atom3"]=self.c_dets[i].atom3
            dettmp["atom4"]=self.c_dets[i].atom4
            dettmp["tar"]=self.c_dets[i].tar
            dettmp["act"]=self.c_dets[i].act
            dettmp["er"]=self.c_dets[i].er
            dettmp["type"]=self.c_dets[i].type
            dettmp["ewt"]=self.c_dets[i].ewt
            detout.append(dettmp)
        free(self.c_dets)
        return detout

    #make a C version of the python list foldmain.ramas
    def mk_c_ramalst(self, pyramalist):
        n=len(pyramalist)
        self.info.numramas=n
        self.c_ramas = <c_rama *> malloc(n * sizeof(c_rama))
        for i in range(n):
            self.c_ramas[i].atom1=pyramalist[i]["atom1"]
            self.c_ramas[i].atom2=pyramalist[i]["atom2"]
            self.c_ramas[i].atom3=pyramalist[i]["atom3"]
            self.c_ramas[i].atom4=pyramalist[i]["atom4"]
```

```
            self.c_ramas[i].atom5=pyramalist[i]["atom5"]
            self.c_ramas[i].aminoacidnum=pyramalist[i]["aanum"]
            self.c_ramas[i].er=pyramalist[i]["er"]
            self.c_ramas[i].type=pyramalist[i]["type"]
            self.c_ramas[i].score=pyramalist[i]["score"]
            self.c_ramas[i].phi=pyramalist[i]["phi"]
            self.c_ramas[i].psi=pyramalist[i]["psi"]
            self.c_ramas[i].sinphi=pyramalist[i]["sinphi"]
            self.c_ramas[i].cosphi=pyramalist[i]["cosphi"]
            self.c_ramas[i].Aphi=pyramalist[i]["Aphi"]
            self.c_ramas[i].sinpsi=pyramalist[i]["sinpsi"]
            self.c_ramas[i].cospsi=pyramalist[i]["cospsi"]
            self.c_ramas[i].Apsi=pyramalist[i]["Apsi"]

    #Get the C version of the rama list, return it
    def gt_c_ramalst(self):
        ramaout=[]
        for i in range(self.info.numramas):
            ramatmp={}
            ramatmp["atom1"]=self.c_ramas[i].atom1
            ramatmp["atom2"]=self.c_ramas[i].atom2
            ramatmp["atom3"]=self.c_ramas[i].atom3
            ramatmp["atom4"]=self.c_ramas[i].atom4
            ramatmp["atom5"]=self.c_ramas[i].atom5
            ramatmp["aanum"]=self.c_ramas[i].aminoacidnum
            ramatmp["er"]=self.c_ramas[i].er
            ramatmp["type"]=self.c_ramas[i].type
            ramatmp["score"]=self.c_ramas[i].score
            ramatmp["phi"]=self.c_ramas[i].phi
            ramatmp["psi"]=self.c_ramas[i].psi
            ramatmp["sinphi"]=self.c_ramas[i].sinphi
            ramatmp["cosphi"]=self.c_ramas[i].cosphi
            ramatmp["Aphi"]=self.c_ramas[i].Aphi
            ramatmp["sinpsi"]=self.c_ramas[i].sinpsi
            ramatmp["cospsi"]=self.c_ramas[i].cospsi
            ramatmp["Apsi"]=self.c_ramas[i].Apsi
            ramaout.append(ramatmp)
        free(self.c_ramas)
        return ramaout

    #Call the C funciton c_get_chainengrad, calculate chain energy, gradient, return both
    #c_get_chainengrad is defined in this file near top, and again in "c_code.c" as "get_chainengrad"
    def c_get_chainengrad(self):
        cdef c_vector *grd
        n=self.info.numatoms
        grd = <c_vector *> malloc(n * sizeof(c_vector))
        en=c_get_chainengrad(self.c_atoms, self.c_bonds, self.c_angls, self.c_dets, self.c_ramas, self.info, grd)
        chgrdout=[]
        for i in range(n):
            grdtmp=Vector(grd[i].x, grd[i].y, grd[i].z)
            chgrdout.append(grdtmp)
        free(grd)
        return {"en":en,"grd":chgrdout}
    #Do a chain projection
    #c_proj_chain is defined in this file near top, and again in "c_code.c" as "proj_chain"
    def c_proj_chain(self):
        took=c_proj_chain(self.c_atoms, self.c_bonds, self.c_angls, self.c_dets, self.c_ramas, self.info)
        return took

    #Call the C funciton c_get_potengrad, calculate potential energy, gradient, return both
    #c_get_potengrad is defined in this file near top, and again in "c_code.c" as "get_potengrad"
    def c_get_potengrad(self):
```

```
        cdef c_vector *grd
        n=self.info.numatoms
        grd = <c_vector *> malloc(n * sizeof(c_vector))
        en=c_get_potengrad(self.c_atoms, self.info, grd)
        ptgrdout=[]
        for i in range(n):
            grdtmp=Vector(grd[i].x, grd[i].y, grd[i].z)
            ptgrdout.append(grdtmp)
        free(grd)
        return {"en":en,"grd":ptgrdout}
    #Do a potential projection
    #c_proj_pot is defined in this file near top, and again in "c_code.c" as "proj_pot"
    def c_proj_pot(self):
        took=c_proj_pot(self.c_atoms, self.info)
        return took

    #Do self.info.refine_iters number of alternating projections
    #alt_proj is defined near to of this file, and again in "c_code.c"
    def c_altproj(self):
        self.info.chain_proj_iter_max=1000
        self.info.chain_max_step_size=.1
        self.info.chain_min_step_size=.00001
        self.info.chain_min_nrg=.001
        self.info.pot_proj_iter_max=100
        self.info.pot_max_step_size=.1
        self.info.pot_min_step_size=.0001
        self.info.prg=0
        alt_proj(self.c_atoms, self.c_bonds, self.c_angls, self.c_dets, self.c_ramas, self.info)

    #Do "num" DM iterations, or until DM er is less than self.info.min_dm_err
    #DM_iter is defined near to of this file, and again in "c_code.c"
    def c_DM_iter(self,num):
        for i in range(num):
            er=DM_iter(self.c_atoms, self.c_bonds, self.c_angls, self.c_dets, self.c_ramas, self.info)
            if er<self.info.min_dm_err : break
        return er
```

```
from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

setup(
  ext_modules=[
    Extension("pyconverter", ["pyconverter.pyx", "c_code.c", "vector.c", "enrg_grad.c", "minimizers.c"]),
    ],
  cmdclass = {'build_ext': build_ext}
)
```

```
#include <stdio.h>
#include <stdlib.h>
#include "./minimizers.h"


double get_chainengrad(atom *, bond *, angl *, det *, rama *, proteininfo, fvector *);
int proj_chain(atom *, bond *, angl *, det *, rama *, proteininfo);

double get_potengrad(atom *, proteininfo, fvector *);
int proj_pot(atom *, proteininfo);

void alt_proj(atom *, bond *, angl *, det *, rama *, proteininfo);
double DM_iter(atom *, bond *, angl *, det *, rama *, proteininfo);
```

```
#include "c_code.h"

//This function is passed all the constraint lists, atom coordinates
//and returns the energy while filling in the "grd" array with the gradient
double get_chainengrad(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo info, fvector *grd)
{
    double en=chainengrad(atmlst, bndlst, anglst, detlst, rmalst, &info, grd);
    return en;
}

//This function does the chain projection, returns how many minimization steps it took
int proj_chain(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo info)
{
    int took=Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
    return took;
}

//This function first calculates the neighbor list,
//then uses the neighbor and atom list to calculate the potential energy.
//returns how long it took, and fills in the gradient array
double get_potengrad(atom *atmlst, proteininfo info, fvector *grd)
{
    double en;
    nghbrmaker(atmlst, &info);
    en=potengrad(atmlst, &info, grd);
    return en;
}

//This function does the potential projection, returns how many minimization steps it took
int proj_pot(atom *atmlst, proteininfo info)
{
    int took=Projpot(atmlst, &info);
    return took;
}

//this function does info.refine_iters alternating projections
void alt_proj(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo info)
{
    int i;
    double en;
    fvector grd[info.numatoms];//this grd array is dummy, it's never used

    nghbrmaker(atmlst, &info);//create the neighbor list

    for (i=0; i<info.refine_iters; i++)
    {
        en=potengrad(atmlst, &info, grd);
        info.pot_min_nrg=en-1;//potential project just a little down hill, 1 unit
        Projpot(atmlst, &info);
        Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
        if (i%10==0) {//every 10 iterations, output AP progress, recalculate neighbor list
            printf("pot en=%f  chain en=%f  iter=%d / %d \n",en,chainengrad(atmlst, bndlst, anglst, detlst, rmalst, &info, grd), i, info.refine_iters );
            nghbrmaker(atmlst, &info);
        }
    }
printf("refine done:  pot en=%f  chain en=%f  iter=%d / %d \n",en,chainengrad(atmlst, bndlst, anglst, detlst, rmalst, &info, grd), i, info.refine_iters );
}

//This function does 1 DM iteration
double DM_iter(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo info)
{
```

```
    int na=info.numatoms, i;
    double er=0, erav;
    fvector p1[na], p2[na], p3[na], p4[na], d[na], orig[na];

    //save the initial atom configuration
    for(i=0; i<na; i++) {    orig[i].x=atmlst[i].r.x;     orig[i].y=atmlst[i].r.y;     orig[i].z=atmlst[i].r.z; }

    if(info.beta==1)//if beta=1, DM takes special quicker form
    {
    Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
        for(i=0; i<na; i++) {        p2[i].x=atmlst[i].r.x;        p2[i].y=atmlst[i].r.y;        p2[i].z=atmlst[i].r.z;}
        for(i=0; i<na; i++) atmlst[i].r=Vadd(p2[i], Vsub(p2[i],orig[i]));
    Projpot(atmlst, &info);
        for(i=0; i<na; i++) {        p3[i].x=atmlst[i].r.x;        p3[i].y=atmlst[i].r.y;        p3[i].z=atmlst[i].r.z;}
        for(i=0; i<na; i++) {        p4[i].x=p2[i].x;        p4[i].y=p2[i].y;        p4[i].z=p2[i].z;}
    }
    else if(info.beta==-1)//if beta=1, DM takes special quicker form
    {
    Projpot(atmlst, &info);
        for(i=0; i<na; i++) {        p1[i].x=atmlst[i].r.x;        p1[i].y=atmlst[i].r.y;        p1[i].z=atmlst[i].r.z;}
        for(i=0; i<na; i++) {        p3[i].x=p1[i].x;        p3[i].y=p1[i].y;        p3[i].z=p1[i].z;}
    for(i=0; i<na; i++) atmlst[i].r=Vadd(p1[i], Vsub(p1[i],orig[i]));
    Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
        for(i=0; i<na; i++) {        p4[i].x=atmlst[i].r.x;        p4[i].y=atmlst[i].r.y;        p4[i].z=atmlst[i].r.z;}
    }
    else
    {
    //this moves the configuration, "atmlst" to where it potential projects
    Projpot(atmlst, &info);
        for(i=0; i<na; i++) {
    //saves projected point into array "p1", makes "atmlst" like it started
        p1[i].x=atmlst[i].r.x; atmlst[i].r.x=orig[i].x;
        p1[i].y=atmlst[i].r.y; atmlst[i].r.y=orig[i].y;
        p1[i].z=atmlst[i].r.z; atmlst[i].r.z=orig[i].z;}
    //this moves the configuration "atmlst" to where it chain projects
    Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
        for(i=0; i<na; i++) {
    //saves projected point into array "p2", "atmlst" is left as configuration "p2"
        p2[i].x=atmlst[i].r.x;
        p2[i].y=atmlst[i].r.y;
        p2[i].z=atmlst[i].r.z;}
    //This next line makes "atmlst" into one of the f's
        for(i=0; i<na; i++) atmlst[i].r=Vadd(p2[i], Vscalmul(Vsub(p2[i],orig[i]), 1./info.beta));
    //project the f
    Projpot(atmlst, &info);
        for(i=0; i<na; i++) {
    //saves result of projection into "p3"
        p3[i].x=atmlst[i].r.x;
        p3[i].y=atmlst[i].r.y;
        p3[i].z=atmlst[i].r.z;}
    //This next line makes "atmlst" into the other f
    for(i=0; i<na; i++) atmlst[i].r=Vadd(p1[i], Vscalmul(Vsub(p1[i],orig[i]), -1./info.beta));
    //projects this second f
    Projchain(atmlst, bndlst, anglst, detlst, rmalst, &info);
        for(i=0; i<na; i++) {
    //puts result of projection into "p4"
        p4[i].x=atmlst[i].r.x;
        p4[i].y=atmlst[i].r.y;
        p4[i].z=atmlst[i].r.z;}
    }

    for(i=0; i<na; i++)  d[i]=Vsub(p3[i],p4[i]);//calculate d
```

```
    for(i=0; i<na; i++)  atmlst[i].r=Vadd(orig[i],Vscalmul(d[i],info.beta));//make move

    er=0;
    for(i=0; i<na; i++)  er+=Vnorm(d[i]);
    erav=er/na;//calculate DM error.  This is returned by the funtcion

    //update rescale parameters.  Atoms that move a lot get heavier
    if(erav>info.min_dm_err)  for(i=0; i<na; i++) atmlst[i].rscl=atmlst[i].rscl*.9+.1*Vnorm(d[i])/erav;
    return(erav);
}
```

```
#include "./enrg_grad.h"

int Projchain(atom *, bond *, angl *, det *, rama *, proteininfo *);
int Projpot(atom *, proteininfo *);
```

```
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "./minimizers.h"

//This function takes the atom coordinates in "atmlst"
//and REPLACES them with the chain projection
int Projchain(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo *info)
{
    int i, j, k=0, na=info->numatoms;
    atom xn[na];
    fvector grd[na], grdtmp[na], vtmp;
    double d, dold, en, entmp, maggrd;

    dold=.1;//Initial step size
    for(k=1; k<info->chain_proj_iter_max+1; k++)
    {
        for(i=0; i<na; i++) {
//"atmlst" coordinates are stored in "xn".  xn is saved until last step
        xn[i].r.x=atmlst[i].r.x;
        xn[i].r.y=atmlst[i].r.y;
        xn[i].r.z=atmlst[i].r.z; }

 //compute energy (en), and gradient (grd)
        en=chainengrad(atmlst, bndlst, anglst, detlst, rmalst, info, grd);
        maggrd=0.;
        for(i=0; i<na; i++) maggrd+=Vnormsquare(grd[i]);
        maggrd=sqrt(maggrd);    //MAGNITUDE OF THE GRADIENT
        for(i=0; i<na; i++) grd[i]=Vscalmul(grd[i], 1./maggrd);//normalize gradient
        if (maggrd<0.00001) {printf(" chain maggrd too small maggrd=%f ********** \n", maggrd); break;}
//compute test step length, so energy decreases
//If gradient is calculated right, there HAS to be a step short enough so energy decreases
        for (j=1; j<20; j++){
            for(i=0; i<na; i++) {
//move atom i by d*grd[i]/inertia[i]
                vtmp=Vsub(xn[i].r,Vscalmul(grd[i],dold/atmlst[i].rscl ));
                atmlst[i].r.x=vtmp.x;
                atmlst[i].r.y=vtmp.y;
                atmlst[i].r.z=vtmp.z; }
//what is new energy after move?
            entmp=chainengrad(atmlst, bndlst, anglst, detlst, rmalst, info, grdtmp);
            if (entmp<en) break; else dold=dold*.5;  //if energy decreased, move on, otherwise, shorter step
        }

//compute step length (d).  Assumes parabolic energy, along grd line
        if(en-entmp<=maggrd*dold) d=(dold/3.)/(1.-((en-entmp)/(maggrd*dold))); else d=dold*1.5;
        if(d>info->chain_max_step_size) d=info->chain_max_step_size;


//checks and output
        if( (info->bond_er+info->angle_er+info->det_er+info->rama_er)/info->numaminos < info->chain_min_nrg) {
            if(info->prg >=2) printf("   chainproj  made it (e).     totaler =%f  iter=%d \n",info->bond_er+info->angle_er+info->det_er+info->rama_er, k);   break;    }
        if(d < info->chain_min_step_size) {
            if(info->prg >=1) printf("     chainproj quit due to step size.  en=%f  iter=%d  d*1000000=%f  ****************************************\n", en, k, d*1000000);    break;    }
        if(k==info->chain_proj_iter_max){
            if(info->prg >=1) printf("     chainproj ran out of iterations.  en=%f  iter=%d  maggrd=%f  **********************\n",en, k, maggrd);    break;}
        if(info->prg >= 3 && k%2000==0)    printf(" det er %f  angle er %f  bond er %f  rama er %f  totaler =%f  iter=%d  d=%f \n",info->det_er, info->angle_er, info->bond_er, info->rama_er,
 info->bond_er+info->angle_er+info->det_er+info->rama_er, k, d);
        if(info->prg >= 5)  printf(" det er %f  angle er %f  bond er %f  rama er %f  totaler =%f  iter=%d \n",info->det_er, info->angle_er, info->bond_er, info->rama_er, info->bond_er+inf
o->angle_er+info->det_er+info->rama_er, k);
```

```
//take the step
        for(i=0; i<na; i++) {
            vtmp=Vsub(xn[i].r,Vscalmul(grd[i],d/atmlst[i].rscl ));
            atmlst[i].r.x=vtmp.x;
            atmlst[i].r.y=vtmp.y;
            atmlst[i].r.z=vtmp.z; }
        dold=d;    //next initial test step is the old step length
    }

//fill the constraint lists with updated energy values
    chainengrad(atmlst, bndlst, anglst, detlst, rmalst, info, grd);

    return(k);//number of minimization iterations returned
}



//This function takes the atom coordinates in "atmlst"
//and REPLACES them with the potential projection
int Projpot(atom *atmlst, proteininfo *info)
{
    int i, j, k=0, na=info->numatoms;
    atom xn[na];
    fvector grd[na], grdtmp[na], vtmp;
    double d, dold, en, entmp, maggrd;

    nghbrmaker(atmlst, info);

    dold=.1;//Initial step size
    for(k=1; k<info->pot_proj_iter_max+1; k++) {
        if (k%10==1) nghbrmaker(atmlst, info); //every 10 iteration, recompute neighbors
        for(i=0; i<na; i++) {
//"atmlst" coordinates are stored in "xn".  xn is saved until last step
            xn[i].r.x=atmlst[i].r.x;
            xn[i].r.y=atmlst[i].r.y;
            xn[i].r.z=atmlst[i].r.z; }

 //compute energy (en), and gradient (grd)
        en=potengrad(atmlst, info, grd);
        maggrd=0.;
        for(i=0; i<na; i++) maggrd+=Vnormsquare(grd[i]);
        maggrd=sqrt(maggrd);     //MAGNITUDE OF THE GRADIENT
        for(i=0; i<na; i++) grd[i]=Vscalmul(grd[i], 1./maggrd);//normalize gradient
        if (maggrd<0.00001) {printf(" potential maggrd too small maggrd=%f ********** \n", maggrd); break;}
//compute test step length, so energy decreases
//If gradient is calculated right, there HAS to be a step short enough so energy decreases
        for (j=1; j<20; j++){
            for(i=0; i<na; i++) {
//move atom i by d*grd[i]/inertia[i]
                vtmp=Vsub(xn[i].r,Vscalmul(grd[i],dold/atmlst[i].rscl ));
                atmlst[i].r.x=vtmp.x;
                atmlst[i].r.y=vtmp.y;
                atmlst[i].r.z=vtmp.z; }
//what is new energy after move?
        entmp=potengrad(atmlst, info, grdtmp);
        if (entmp<en) break; else dold=dold*.5; //if energy decreased, move on, otherwise, shorter step
        }

//compute step length (d).  Assumes parabolic energy, along grd line
        if(en-entmp<=maggrd*dold) d=(dold/3.)/(1.-((en-entmp)/(maggrd*dold))); else d=dold*1.5;
```

```
        if(d>info->pot_max_step_size) d=info->pot_max_step_size;


//checks and outputs
        if(en < info->pot_min_nrg) {if(info->prg>=2) printf("    potential proj made it (e). totalen =%f  iter=%d \n",en, k);        break;}
        if(d < info->pot_min_step_size) {
            if(info->prg >=1) printf("     genpotproj quit due to step size.  en=%f  iter=%d  d*1000000=%f  *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*\n", en, k, d*1000000);   break;     }
        if(k==info->pot_proj_iter_max){
            if(info->prg >=1) printf("     genpotproj ran out of iterations.  en=%f  iter=%d  maggrd=%f  *-*-*-*-*-*-*-*-*-*-*-*\n",en, k, maggrd);  break;}
        if(info->prg >= 3) if(k%2000==0)    printf(" genpotproj  en=%f  iter=%d  maggrd=%f  d=%f  d/maggrd=%f  j=%d \n",en, k, maggrd, d, d/maggrd, j);
        if(info->prg >= 5)  printf(" genpotproj  en=%f  iter=%d  maggrd=%f  d=%f  d/maggrd=%f  j=%d \n",en, k, maggrd, d, d/maggrd, j);



//take the step
        for(i=0; i<na; i++) {
            vtmp=Vsub(xn[i].r,Vscalmul(grd[i],d/atmlst[i].rscl ));
            atmlst[i].r.x=vtmp.x;
            atmlst[i].r.y=vtmp.y;
            atmlst[i].r.z=vtmp.z; }
    dold=d;    //next initial test step is the old step length

    }

    nghbrmaker(atmlst, info);
//fill the info structure with updated energy values
    potengrad(atmlst, info, grd);

    return(k);//number of minimization iterations returned
}
```

```
#include "./structures.h"

double chainengrad(atom *, bond *, angl *, det *, rama *, proteininfo *, fvector *);
double ramaengrad(atom *, rama *, proteininfo *, fvector *);

void nghbrmaker(atom *, proteininfo *);
double potengrad(atom *, proteininfo *, fvector *);
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "./enrg_grad.h"

//calculates the energy of an atomic configuration, and fills in the gradient array
double chainengrad(atom *atmlst, bond *bndlst, angl *anglst, det *detlst, rama *rmalst, proteininfo *info, fvector *grd)
{
    fvector v1, v2, v3, v4;
    int i, a1, a2, a3, a4;
    double sum=0, ewt, st, tar, act;
    char *type;

    for(i=0; i<info->numatoms; i++) grd[i]=Vectormkr(0,0,0); //clear the gradient array

    info->bond_er=0;//clear the total bond error
    for(i=0; i<info->numbonds; i++) //go through the bond list
    {
        a1=bndlst[i].atom1;
        a2=bndlst[i].atom2;
        tar=bndlst[i].tar;//target value
        ewt=bndlst[i].ewt;//energy weight

        v1=Vsub(atmlst[a1].r, atmlst[a2].r);
        act=Vdot(v1,v1);
        st=(act-tar);
        sum=sum+ewt*st*st;
        info->bond_er+=ewt*st*st;//add the ith bond error to the total bond error
        bndlst[i].act=act;//actual bond length
        bndlst[i].er=st*st;//energy weighted unsatisfiedness
        grd[a1]=Vadd(grd[a1],Vscalmul(v1,2.*ewt*st*2.));
        grd[a2]=Vadd(grd[a2],Vscalmul(v1,-2.*ewt*st*2.));
    }

    info->angle_er=0;//clear the total angle error
    for(i=0; i<info->numangles; i++)//go through the angle list
    {
        a1=anglst[i].atom1;
        a2=anglst[i].atom2;
        a3=anglst[i].atom3;
        tar=anglst[i].tar;//target value
        ewt=anglst[i].ewt;//energy weight

        v1=Vsub(atmlst[a1].r, atmlst[a2].r);
        v2=Vsub(atmlst[a3].r, atmlst[a2].r);
        act=Vdot(v1, v2);
        st=(act-tar);
        sum=sum+ewt*st*st;
        info->angle_er+=ewt*st*st;//add the ith angle error to the total angle error
        anglst[i].act=act;//actual angle value
        anglst[i].er=st*st;//energy weighted unsatisfiedness
        grd[a1]=Vadd(grd[a1],Vscalmul(v2,ewt*st*2.));
        grd[a2]=Vadd(grd[a2],Vscalmul(Vadd(v1,v2),-1.*ewt*st*2.));
        grd[a3]=Vadd(grd[a3],Vscalmul(v1,ewt*st*2.));
    }

    info->det_er=0;//clear the total determinant error
    for(i=0; i<info->numdets; i++)//go through the determinant list
    {
        a1=detlst[i].atom1;
        a2=detlst[i].atom2;
```

```
        a3=detlst[i].atom3;
        a4=detlst[i].atom4;
        v1=Vsub(atmlst[a1].r, atmlst[a2].r);
        v2=Vsub(atmlst[a2].r, atmlst[a3].r);
        v3=Vsub(atmlst[a3].r, atmlst[a4].r);
        v4=Vsub(atmlst[a1].r, atmlst[a4].r);
        ewt=detlst[i].ewt;//energy weight
        tar=detlst[i].tar;//target value
        type=detlst[i].type;

        if (strcmp(type,"datt")==0 || strcmp(type,"dbkbn")==0) {
            act=Vdot(v1, v3);
            st=(act-tar);
            sum=sum+ewt*st*st;
            info->det_er+=ewt*st*st;//add the ith det error to the total det error
            detlst[i].act=act;//actual determinant value
            detlst[i].er=st*st;//energy weighted unsatisfiedness
            grd[a1]=Vadd(grd[a1],Vscalmul(v3,ewt*st*2.));
            grd[a2]=Vadd(grd[a2],Vscalmul(v3,-1.*ewt*st*2.));
            grd[a3]=Vadd(grd[a3],Vscalmul(v1,ewt*st*2.));
            grd[a4]=Vadd(grd[a4],Vscalmul(v1,-1.*ewt*st*2.));
        }
        else if (strcmp(type,"catt")==0 || strcmp(type,"cbkbn")==0 || strcmp(type,"crgrp")==0) {
            act=Vdot(v1,Vcross(v2,v3));
            st=(act-tar);
            sum=sum+ewt*st*st;
            info->det_er+=ewt*st*st;//add the ith det error to the total det error
            detlst[i].act=act;//actual determinant value
            detlst[i].er=st*st;//energy weighted unsatisfiedness
            grd[a1]=Vadd(grd[a1],Vscalmul(Vcross(v2,v3),2.*ewt*st));
            grd[a2]=Vadd(grd[a2],Vscalmul(Vcross(v3,v4),2.*ewt*st));
            grd[a3]=Vadd(grd[a3],Vscalmul(Vcross(v1,v4),2.*ewt*st));
            grd[a4]=Vadd(grd[a4],Vscalmul(Vcross(v2,v1),2.*ewt*st));
        }
        else if (strcmp(type,"cbdet")==0) {
            act=Vdot(v1,Vcross(v2,v3));
            st=(act-tar);
            sum=sum+ewt*st*st;
            info->det_er+=ewt*st*st;//add the ith det error to the total det error
            detlst[i].act=act;//actual determinant value
            detlst[i].er=st*st;//energy weighted unsatisfiedness
            grd[a1]=Vadd(grd[a1],Vscalmul(Vcross(v2,v3),2.*ewt*st));
            grd[a2]=Vadd(grd[a2],Vscalmul(Vcross(v3,v4),2.*ewt*st));
            grd[a3]=Vadd(grd[a3],Vscalmul(Vcross(v1,v4),2.*ewt*st));
            grd[a4]=Vadd(grd[a4],Vscalmul(Vcross(v2,v1),2.*ewt*st));
        }
        else {//Should never get here
            printf(" UNKNOWN det type in enrg_grad.c    type=%s    ", type);
            printf(" at1=%d  at2=%d  at3=%d  at4=%d  ewt=%f  tar=%f  \n", a1, a2, a3, a4, ewt, tar);
        }
    }
    //the ramachandrin energy is a big function, it gets its own function below
    sum=sum+ramaengrad(atmlst, rmalst, info, grd);
    //rescale gradient vectors according to inertia
    for(i=0; i<info->numatoms; i++) grd[i]=Vscalmul(grd[i],1./atmlst[i].rscl);
    //gradient array is now filled, return energy INCLUDING guiding function in ramaengrad
    return(sum);
}

//calculate the ramachandrin energy, adds ramachandrin gradient to total gradient
double ramaengrad(atom *atmlst, rama *rmalst, proteininfo *info, fvector *grd){
    fvector v1, v2, v3, v4, c12, c23, c34, c31, c42;
```

```
    fvector dsphidv0, dsphidv1, dsphidv2, dsphidv3, dsphidv4, dcphidv0, dcphidv1, dcphidv2, dcphidv3, dcphidv4;
    fvector dAphidv1,dAphidv2,dAphidv3,dApsidv2,dApsidv3,dApsidv4;
    fvector dspsidv1, dspsidv2, dspsidv3, dspsidv4, dspsidv5, dcpsidv1, dcpsidv2, dcpsidv3, dcpsidv4, dcpsidv5, vtemp1, vtemp2, vtemp3;
    int i, a1, a2, a3, a4, a5;
    double ewt, ler, ler1, ler2, sinphi, cosphi, sinpsi, cospsi, ang, Aphi, Apsi, tar, sum=0;
    double  sphio,spsio,cphio,cpsio, d11, d22, srtd22, d33, srtd33, d44, d12, d23, d34, d13, d24, stphi, stpsi, cphi, cpsi;

    //calculate ramachandran angles
    for(i=0; i<info->numramas; i++)
    {
        a2=rmalst[i].atom2;
        a3=rmalst[i].atom3;
        a4=rmalst[i].atom4;

        if (strcmp(rmalst[i].type,"frst")!=0){//first amino acid has no phi
            //phi calculation
            a1=rmalst[i].atom1;
            v1=Vsub(atmlst[a2].r, atmlst[a1].r);
            v2=Vsub(atmlst[a3].r, atmlst[a2].r);
            v3=Vsub(atmlst[a4].r, atmlst[a3].r);
            c12=Vcross(v1,v2);
            c23=Vcross(v2,v3);
            c31=Vcross(v3,v1);
            Aphi=sqrt(Vdot(c12,c12)*Vdot(c23,c23));
            sinphi=Vdot(v2,c31)*sqrt(Vdot(v2,v2))/Aphi;
            cosphi=(Vdot(v1,v2)*Vdot(v2,v3)-Vdot(v1,v3)*Vdot(v2,v2))/Aphi;
            ang=atan2(sinphi,cosphi)*180./3.14159265;
            rmalst[i].phi=ang;//actual phi value
            rmalst[i].sinphi=sinphi;
            rmalst[i].cosphi=cosphi;
            rmalst[i].Aphi=Aphi;}//store Aphi for later, rather than recalculate
        else {rmalst[i].phi=0.;  rmalst[i].sinphi=0.; rmalst[i].cosphi=0.; rmalst[i].Aphi=1.;}

        if (strcmp(rmalst[i].type,"lst")!=0){//last amino acid has no psi
            //psi calculation
            a5=rmalst[i].atom5;
            v2=Vsub(atmlst[a3].r, atmlst[a2].r);
            v3=Vsub(atmlst[a4].r, atmlst[a3].r);
            v4=Vsub(atmlst[a5].r, atmlst[a4].r);
            c23=Vcross(v2,v3);
            c34=Vcross(v3,v4);
            c42=Vcross(v4,v2);
            Apsi=sqrt(Vdot(c23,c23)*Vdot(c34,c34));
            sinpsi=Vdot(v3,c42)*sqrt(Vdot(v3,v3))/Apsi;
            cospsi=(Vdot(v2,v3)*Vdot(v3,v4)-Vdot(v2,v4)*Vdot(v3,v3))/Apsi;
            ang=atan2(sinpsi,cospsi)*180./3.14159265;
            rmalst[i].psi=ang;//actual psi value
            rmalst[i].sinpsi=sinpsi;
            rmalst[i].cospsi=cospsi;
            rmalst[i].Apsi=Apsi;}//store Apsi for later, rather than recalculate
        else {rmalst[i].psi=0.;  rmalst[i].sinpsi=0.; rmalst[i].cospsi=0.; rmalst[i].Apsi=1.;}

    // calculate score for each phi-psi pair
    // Is the ith amino acid alpha helix like, or beta sheet like?
        rmalst[i].score=0.;

        if(strcmp(rmalst[i].type,"lst")==0 || strcmp(rmalst[i].type,"frst")==0 ) rmalst[i].score=0.;
        else {
            sphio=-.8387;    spsio=-.7314;    cphio=.5446;     cpsio=.6820;          //a helix
            ler1= ((sinphi-sphio)*(sinphi-sphio)+(cosphi-cphio)*(cosphi-cphio)+(sinpsi-spsio)*(sinpsi-spsio)+(cospsi-cpsio)*(cospsi-cpsio))/1.; //<--- the over 1 he
re makes alpha helix basin smaller than beta sheet
            sphio=-.7071;    spsio=.7071;     cphio=-.7071;    cpsio=-.7071;         //b sheet
```

```
            ler2= ((sinphi-sphio)*(sinphi-sphio)+(cosphi-cphio)*(cosphi-cphio)+(sinpsi-spsio)*(sinpsi-spsio)+(cospsi-cpsio)*(cospsi-cpsio))/2.; //<--- the over 2 he
re makes beta sheet basin bigger then alpha helix
            if (ler1<1.) rmalst[i].score+=-1.*(1.-ler1)*(1.-ler1);                  //a helix
            if (ler2<1.) rmalst[i].score+=(1.-ler2)*(1.-ler2);                      //b sheet
        }
        rmalst[i].er=0.;//clear the ith ramachandrin error.  Will be filled in next...
    }

    //Do the rama energies
    info->rama_er=0.;//clear the total rama error
    for(i=0; i<info->numramas; i++)
    {
        a1=rmalst[i].atom1;
        a2=rmalst[i].atom2;
        a3=rmalst[i].atom3;
        a4=rmalst[i].atom4;
        a5=rmalst[i].atom5;

        if(strcmp(rmalst[i].type,"frst")!=0) v1=Vsub(atmlst[a2].r, atmlst[a1].r); else v1=Vectormkr(0,0,0);
        v2=Vsub(atmlst[a3].r, atmlst[a2].r);
        v3=Vsub(atmlst[a4].r, atmlst[a3].r);
        if(strcmp(rmalst[i].type,"lst")!=0) v4=Vsub(atmlst[a5].r, atmlst[a4].r);  else v4=Vectormkr(0,0,0);
    //calculate these all once here, rather than many times later
        c12=Vcross(v1,v2);
        c23=Vcross(v2,v3);
        c34=Vcross(v3,v4);
        c31=Vcross(v3,v1);
        c42=Vcross(v4,v2);
        d11=Vdot(v1,v1);
        d22=Vdot(v2,v2);  srtd22=sqrt(d22);
        d33=Vdot(v3,v3);  srtd33=sqrt(d33);
        d44=Vdot(v4,v4);
        d12=Vdot(v1,v2);
        d23=Vdot(v2,v3);
        d34=Vdot(v3,v4);
        d13=Vdot(v1,v3);
        d24=Vdot(v2,v4);

    // lookup phi and psi for ith amino acid, calculated above, near beginning of this function
        Aphi=rmalst[i].Aphi;
        sinphi=rmalst[i].sinphi;
        cosphi=rmalst[i].cosphi;
        Apsi=rmalst[i].Apsi;
        sinpsi=rmalst[i].sinpsi;
        cospsi=rmalst[i].cospsi;

    // Calculate dsinpsidv and dcospsidv and dsinphidv and dcosphidv
    // Independent of where the angle is being pushed to, these only matter where it IS
        if(strcmp(rmalst[i].type,"frst")!=0){//This calculates the phi gradient coefficients
                vtemp1=Vadd(Vscalmul(v1,2.*d22),Vscalmul(v2,-2.*d12));
            dAphidv1=Vscalmul(vtemp1,Vdot(c23,c23)/(2.*Aphi));
                vtemp1=Vadd(Vscalmul(v3,2.*d22),Vscalmul(v2,-2.*d23));
            dAphidv3=Vscalmul(vtemp1,Vdot(c12,c12)/(2.*Aphi));
                vtemp1=Vadd(Vscalmul(v2,2.*d11),Vscalmul(v1,-2.*d12));
                vtemp2=Vadd(Vscalmul(v2,2.*d33),Vscalmul(v3,-2.*d23));
            dAphidv2=Vadd(Vscalmul(vtemp1,Vdot(c23,c23)/(2.*Aphi)),Vscalmul(vtemp2,Vdot(c12,c12)/(2.*Aphi)));

            dsphidv0=Vectormkr(0,0,0);
                vtemp1=Vscalmul(c23,srtd22/Aphi);
                vtemp2=Vscalmul(dAphidv1,-1.*Vdot(v2,c31)*srtd22/(Aphi*Aphi));
            dsphidv1=Vadd(vtemp1, vtemp2);
                vtemp1=Vscalmul(c12,srtd22/Aphi);
```

```
            vtemp2=Vscalmul(dAphidv3,-1.*Vdot(v2,c31)*srtd22/(Aphi*Aphi));
        dsphidv3=Vadd(vtemp1, vtemp2);
            vtemp1=Vscalmul(c31,srtd22/Aphi);
            vtemp2=Vscalmul(v2,Vdot(v2,c31)/(srtd22*Aphi));
            vtemp3=Vscalmul(dAphidv2,-1.*Vdot(v2,c31)*srtd22/(Aphi*Aphi));
        dsphidv2=Vadd(Vadd(vtemp1, vtemp2),vtemp3);
        dsphidv4=Vectormkr(0,0,0);

        dcphidv0=Vectormkr(0,0,0);
            vtemp1=Vadd(Vscalmul(v2,d23/Aphi),Vscalmul(v3,-1.*d22/Aphi));
            vtemp2=Vscalmul(dAphidv1,-1.*(d12*d23-d13*d22)/(Aphi*Aphi));
        dcphidv1=Vadd(vtemp1, vtemp2);
            vtemp1=Vadd(Vscalmul(v2,d12/Aphi),Vscalmul(v1,-1.*d22/Aphi));
            vtemp2=Vscalmul(dAphidv3,-1.*(d12*d23-d13*d22)/(Aphi*Aphi));
        dcphidv3=Vadd(vtemp1, vtemp2);
            vtemp1=Vadd(Vadd(Vscalmul(v1,d23/Aphi),Vscalmul(v3,d12/Aphi)),Vscalmul(v2,-2.*d13/Aphi));
            vtemp2=Vscalmul(dAphidv2,-1.*(d12*d23-d13*d22)/(Aphi*Aphi));
        dcphidv2=Vadd(vtemp1, vtemp2);
        dcphidv4=Vectormkr(0,0,0);
    }
    else         {//first amino acid has no phi
        dsphidv0=Vectormkr(0,0,0);        dsphidv1=Vectormkr(0,0,0);        dsphidv2=Vectormkr(0,0,0);
        dsphidv3=Vectormkr(0,0,0);        dsphidv4=Vectormkr(0,0,0);
        dcphidv0=Vectormkr(0,0,0);        dcphidv1=Vectormkr(0,0,0);        dcphidv2=Vectormkr(0,0,0);
        dcphidv3=Vectormkr(0,0,0);        dcphidv4=Vectormkr(0,0,0);
    }

    if(strcmp(rmalst[i].type,"lst")!=0){//This calculates the psi gradient coefficients
            vtemp1=Vadd(Vscalmul(v2,2.*d33),Vscalmul(v3,-2.*d23));
        dApsidv2=Vscalmul(vtemp1,Vdot(c34,c34)/(2.*Apsi));
            vtemp1=Vadd(Vscalmul(v4,2.*d33),Vscalmul(v3,-2.*d34));
        dApsidv4=Vscalmul(vtemp1,Vdot(c23,c23)/(2.*Apsi));
            vtemp1=Vadd(Vscalmul(v3,2.*d22),Vscalmul(v2,-2.*d23));
            vtemp2=Vadd(Vscalmul(v3,2.*d44),Vscalmul(v4,-2.*d34));
        dApsidv3=Vadd(Vscalmul(vtemp1,Vdot(c34,c34)/(2.*Apsi)),Vscalmul(vtemp2,Vdot(c23,c23)/(2.*Apsi)));

        dspsidv1=Vectormkr(0,0,0);
            vtemp1=Vscalmul(c34,srtd33/Apsi);
            vtemp2=Vscalmul(dApsidv2,-1.*Vdot(v3,c42)*srtd33/(Apsi*Apsi));
        dspsidv2=Vadd(vtemp1, vtemp2);
            vtemp1=Vscalmul(c23,srtd33/Apsi);
            vtemp2=Vscalmul(dApsidv4,-1.*Vdot(v3,c42)*srtd33/(Apsi*Apsi));
        dspsidv4=Vadd(vtemp1, vtemp2);
            vtemp1=Vscalmul(c42,srtd33/Apsi);
            vtemp2=Vscalmul(v3,Vdot(v3,c42)/(srtd33*Apsi));
            vtemp3=Vscalmul(dApsidv3,-1.*Vdot(v3,c42)*srtd33/(Apsi*Apsi));
        dspsidv3=Vadd(Vadd(vtemp1, vtemp2),vtemp3);
        dspsidv5=Vectormkr(0,0,0);

        dcpsidv1=Vectormkr(0,0,0);
            vtemp1=Vadd(Vscalmul(v3,d34/Apsi),Vscalmul(v4,-1.*d33/Apsi));
            vtemp2=Vscalmul(dApsidv2,-1.*(d23*d34-d24*d33)/(Apsi*Apsi));
        dcpsidv2=Vadd(vtemp1, vtemp2);
            vtemp1=Vadd(Vscalmul(v3,d23/Apsi),Vscalmul(v2,-1.*d33/Apsi));
            vtemp2=Vscalmul(dApsidv4,-1.*(d23*d34-d24*d33)/(Apsi*Apsi));
        dcpsidv4=Vadd(vtemp1, vtemp2);
            vtemp1=Vadd(Vadd(Vscalmul(v2,d34/Apsi),Vscalmul(v4,d23/Apsi)),Vscalmul(v3,-2.*d24/Apsi));
            vtemp2=Vscalmul(dApsidv3,-1.*(d23*d34-d24*d33)/(Apsi*Apsi));
        dcpsidv3=Vadd(vtemp1, vtemp2);
        dcpsidv5=Vectormkr(0,0,0);
    }
    else        {//last amino acid has no psi
```

```
        dspsidv1=Vectormkr(0,0,0);        dspsidv2=Vectormkr(0,0,0);        dspsidv3=Vectormkr(0,0,0);
        dspsidv4=Vectormkr(0,0,0);        dspsidv5=Vectormkr(0,0,0);
        dcpsidv1=Vectormkr(0,0,0);        dcpsidv2=Vectormkr(0,0,0);        dcpsidv3=Vectormkr(0,0,0);
        dcpsidv4=Vectormkr(0,0,0);        dcpsidv5=Vectormkr(0,0,0);
    }

    // calculate the BOX guiding function (equal to real energy constraint)
    //does not apply to proline or glycine
    if (strcmp(rmalst[i].type,"gly")!=0 && strcmp(rmalst[i].type,"pro")!=0 && 1==1) {//1==1 means "on"
        ewt=.3;
        if (strcmp(rmalst[i].type,"frst")!=0){//This part only for phi
            stphi=(75./90.)*sinphi+(15./90.)*cosphi;
            cphi=-.26;
            if (stphi<cphi) ler=0.; else ler=(stphi-cphi);
            sum=sum+ewt*ler*ler;//ewt*ler^2 is the total chain energy
            rmalst[i].er+=ler*ler;//records unsatisfiedness
            info->rama_er+=ewt*ler*ler;//adds ewt*ler^2 to total ramachandrin error

            grd[a1]=Vadd(grd[a1],Vadd(Vscalmul(Vsub(dsphidv0,dsphidv1),2.*ewt*ler*(75./90.)),Vscalmul(Vsub(dcphidv0,dcphidv1),2.*ewt*ler*(15./90.))));
            grd[a2]=Vadd(grd[a2],Vadd(Vscalmul(Vsub(dsphidv1,dsphidv2),2.*ewt*ler*(75./90.)),Vscalmul(Vsub(dcphidv1,dcphidv2),2.*ewt*ler*(15./90.))));
            grd[a3]=Vadd(grd[a3],Vadd(Vscalmul(Vsub(dsphidv2,dsphidv3),2.*ewt*ler*(75./90.)),Vscalmul(Vsub(dcphidv2,dcphidv3),2.*ewt*ler*(15./90.))));
            grd[a4]=Vadd(grd[a4],Vadd(Vscalmul(Vsub(dsphidv3,dsphidv4),2.*ewt*ler*(75./90.)),Vscalmul(Vsub(dcphidv3,dcphidv4),2.*ewt*ler*(15./90.))));
        }
        if(strcmp(rmalst[i].type,"lst")!=0){//This part only for psi
            stpsi=(50./90.)*sinpsi+(40./90.)*cospsi;
            cpsi=-.5;
            if (stpsi>cpsi) ler=0.; else ler=(stpsi-cpsi);
            sum=sum+20.*ewt*ler*ler;//ewt*ler^2 is the total chain energy
            rmalst[i].er+=ler*ler;//records unsatisfiedness
            info->rama_er+=20.*ewt*ler*ler;//adds ewt*ler^2 to total ramachandrin error

            grd[a2]=Vadd(grd[a2],Vadd(Vscalmul(Vsub(dspsidv1,dspsidv2),2.*20.*ewt*ler*(50./90.)),Vscalmul(Vsub(dcpsidv1,dcpsidv2),2.*20.*ewt*ler*(40./90.))));
            grd[a3]=Vadd(grd[a3],Vadd(Vscalmul(Vsub(dspsidv2,dspsidv3),2.*20.*ewt*ler*(50./90.)),Vscalmul(Vsub(dcpsidv2,dcpsidv3),2.*20.*ewt*ler*(40./90.))));
            grd[a4]=Vadd(grd[a4],Vadd(Vscalmul(Vsub(dspsidv3,dspsidv4),2.*20.*ewt*ler*(50./90.)),Vscalmul(Vsub(dcpsidv3,dcpsidv4),2.*20.*ewt*ler*(40./90.))));
            grd[a5]=Vadd(grd[a5],Vadd(Vscalmul(Vsub(dspsidv4,dspsidv5),2.*20.*ewt*ler*(50./90.)),Vscalmul(Vsub(dcpsidv4,dcpsidv5),2.*20.*ewt*ler*(40./90.))));
        }

    }

    // sequence continuity guiding function
    if (strcmp(rmalst[i].type,"pro")!=0 && strcmp(rmalst[i].type,"gly")!=0 && 1==0) {//note 1==1 means "on"
        if (strcmp(rmalst[i].type,"frst")!=0 && strcmp(rmalst[i].type,"lst")!=0) {//only applies to non-first, non-last
            tar=(rmalst[i-1].score+2*rmalst[i].score+rmalst[i+1].score)/4.;//target is weighted average of itself, and neighbors
            if (tar==0.)      {sphio=0;    spsio=0;    cphio=0;     cpsio=0;  ewt=0.;}
            else if (tar<0.)  {sphio=-.8387; spsio=-.7314; cphio=.5446;  cpsio=.6820;  ewt=.04;}   //it wants to be a alpha helix
            else              {sphio=-.7071; spsio=.7071;  cphio=-.7071; cpsio=-.7071; ewt=.002;}//it wants to be a beta sheet
            if (strcmp(rmalst[i-1].type,"gly")==0) {sphio=-.0; spsio=.0;  cphio=-.0; cpsio=-.0;  ewt=.00;}//does not apply if after glycine
            if (strcmp(rmalst[i+1].type,"gly")==0) {sphio=-.0; spsio=.0;  cphio=-.0; cpsio=-.0;  ewt=.00;}//does not before if after glycine
            if (strcmp(rmalst[i-1].type,"pro")==0) {sphio=-.0; spsio=.0;  cphio=-.0; cpsio=-.0;  ewt=.00;}//does not apply if after proline
            if (strcmp(rmalst[i+1].type,"pro")==0) {sphio=-.7071; spsio=.7071;  cphio=-.7071; cpsio=-.7071;  ewt=.05;} //pre proline always beta sheet
        }
        else {ewt=.00;}

        // Guiding energy
        ler=(sinphi-sphio)*(sinphi-sphio)+(cosphi-cphio)*(cosphi-cphio)+(sinpsi-spsio)*(sinpsi-spsio)+(cospsi-cpsio)*(cospsi-cpsio);

        sum=sum+ewt*ler;//adds ewt*ler to the total chain energy
        rmalst[i].er+=ewt*ler;//records unsatisfiedness

        if(strcmp(rmalst[i].type,"frst")!=0){
            grd[a1]=Vadd(grd[a1],Vadd(Vscalmul(Vsub(dsphidv0,dsphidv1),2.*(sinphi-sphio)*ewt),Vscalmul(Vsub(dcphidv0,dcphidv1),2.*(cosphi-cphio)*ewt)));
            grd[a2]=Vadd(grd[a2],Vadd(Vscalmul(Vsub(dsphidv1,dsphidv2),2.*(sinphi-sphio)*ewt),Vscalmul(Vsub(dcphidv1,dcphidv2),2.*(cosphi-cphio)*ewt)));
```

```
                grd[a3]=Vadd(grd[a3],Vadd(Vscalmul(Vsub(dsphidv2,dsphidv3),2.*(sinphi-sphio)*ewt),Vscalmul(Vsub(dcphidv2,dcphidv3),2.*(cosphi-cphio)*ewt)));
                grd[a4]=Vadd(grd[a4],Vadd(Vscalmul(Vsub(dsphidv3,dsphidv4),2.*(sinphi-sphio)*ewt),Vscalmul(Vsub(dcphidv3,dcphidv4),2.*(cosphi-cphio)*ewt)));
            }
            if(strcmp(rmalst[i].type,"lst")!=0){
                grd[a2]=Vadd(grd[a2],Vadd(Vscalmul(Vsub(dspsidv1,dspsidv2),2.*(sinpsi-spsio)*ewt),Vscalmul(Vsub(dcpsidv1,dcpsidv2),2.*(cospsi-cpsio)*ewt)));
                grd[a3]=Vadd(grd[a3],Vadd(Vscalmul(Vsub(dspsidv2,dspsidv3),2.*(sinpsi-spsio)*ewt),Vscalmul(Vsub(dcpsidv2,dcpsidv3),2.*(cospsi-cpsio)*ewt)));
                grd[a4]=Vadd(grd[a4],Vadd(Vscalmul(Vsub(dspsidv3,dspsidv4),2.*(sinpsi-spsio)*ewt),Vscalmul(Vsub(dcpsidv3,dcpsidv4),2.*(cospsi-cpsio)*ewt)));
                grd[a5]=Vadd(grd[a5],Vadd(Vscalmul(Vsub(dspsidv4,dspsidv5),2.*(sinpsi-spsio)*ewt),Vscalmul(Vsub(dcpsidv4,dcpsidv5),2.*(cospsi-cpsio)*ewt)));
            }
        }
    }
    return (sum);//this returned ramachandrin energy is added to the total in chainengrad function above
}


//This function figures out who are the neighbors of each atom
void nghbrmaker(atom *atoms, proteininfo *info)
{
    int i, j, kn, igtmp, tmplst[info->numatoms] ;
    fvector v;

    for(i=0; i<info->numatoms; i++) atoms[i].num_nghbrs=0;//start out with no neighbors

    for(j=0; j<info->numatoms; j++){
    //for the jth atom, consider everybody as possible neighbor
        for(i=0; i<info->numatoms; i++) tmplst[i]=1;//everybody's a suspect
        //throw out the bonded list members
        for(i=0; i<atoms[j].num_bonded; i++){
            igtmp=atoms[j].bonded[i];
            tmplst[igtmp]=0;//discard those that are ignored
        }
        //if not thrown out already, throw out if too far away
        for(i=0; i<info->numatoms; i++){
            if(tmplst[i]==1){//still a possible neighbor
                v=Vsub(atoms[i].r, atoms[j].r);
                if(Vnormsquare(v)>7.*7.) tmplst[i]=0;//if too far away, discard
            }
        }
        //who's left?  add to neighbor list
        for(i=j+1; i<info->numatoms; i++)
        {
            if(tmplst[i]==1)
            {
                kn=atoms[j].num_nghbrs;
                atoms[j].nghbrs[kn]=i;
                atoms[j].num_nghbrs+=1;
            }
        }
    }
}

//calculates the potential of an atomic configuration, and fills in the gradient array
double potengrad(atom *atmlst, proteininfo *info, fvector *grd)
{
    int i,j, a1, a2, an, ah, ao, ac;
    fvector v, vnh, voc;
    double sum, rs, rtarsq, ratio, en, den, cbs, cas, fr, dfr, vnhv, vocv, ewt, bondlengthsquared;
    fvector dedvnh, dedvoc, dedv;

    for(i=0; i<info->numatoms; i++) grd[i]=Vectormkr(0,0,0);//clear the gradient
    sum=0;//clear the total energy.  This is returned in the end
```

```
    for(i=0; i<info->numatoms; i++) {
        a1=i;
        for(j=0; j<atmlst[i].num_nghbrs; j++){ //For the ith atom, run over neighbors...
            a2=atmlst[i].nghbrs[j];
            v=Vsub(atmlst[a1].r,atmlst[a2].r);
            rs=Vnormsquare(v);//distance between ith atom, and jth neighbor

            //Self avoiding potential.  Steric repulsion controlled here
            rtarsq=(atmlst[a1].radius+atmlst[a2].radius)*.76;//This is how close they can come for free
            rtarsq=rtarsq*rtarsq;
            ewt=1.;
            ratio=rs/rtarsq;
            if (ratio<1.) {//check if too close
                en=(1.-ratio)*(1.-ratio);
                den=2.*(1.-ratio)*-1./rtarsq;
                sum=sum+ewt*en;//if too close, add steric energy to total
                grd[a1]=Vadd(grd[a1],Vscalmul(v,2.*ewt*den));
                grd[a2]=Vadd(grd[a2],Vscalmul(v,-2.*ewt*den));
                if (ratio<.8 && info->prg >=4){
                    printf(" a1=%d  %s in %s num %d  a2=%d  %s in %s num %d   ratio=%f \n",a1, atmlst[a1].pdbcode, atmlst[a1].aminoacidtype, atmlst[a1].aminoacidnum, a2, atmlst[a
2].pdbcode, atmlst[a2].aminoacidtype, atmlst[a2].aminoacidnum, ratio);
                }
            }

            //hydrophobic energy
            if (atmlst[a1].hydro!=0 && atmlst[a2].hydro!=0 && atmlst[a1].aminoacidnum!=atmlst[a2].aminoacidnum) {
//////////////////////////
//HYDROPHOBIC ENERGIES!!!
//////////////////////////
                if (atmlst[a1].hydro==1 && atmlst[a2].hydro==1) ewt=-.364;
                else if (atmlst[a1].hydro==2 && atmlst[a2].hydro==2) ewt=-.272;
                else if (atmlst[a1].hydro==3 && atmlst[a2].hydro==3) ewt=.42;
                else if (atmlst[a1].hydro==4 && atmlst[a2].hydro==4) ewt=-.42;
                else if ((atmlst[a1].hydro==1 && atmlst[a2].hydro==2) || (atmlst[a1].hydro==2 && atmlst[a2].hydro==1)) ewt=-.248;
                else if ((atmlst[a1].hydro==1 && atmlst[a2].hydro==3) || (atmlst[a1].hydro==3 && atmlst[a2].hydro==1)) ewt=.238;
                else if ((atmlst[a1].hydro==1 && atmlst[a2].hydro==4) || (atmlst[a1].hydro==4 && atmlst[a2].hydro==1)) ewt=-.212;
                else if ((atmlst[a1].hydro==2 && atmlst[a2].hydro==3) || (atmlst[a1].hydro==3 && atmlst[a2].hydro==2)) ewt=.314;
                else if ((atmlst[a1].hydro==2 && atmlst[a2].hydro==4) || (atmlst[a1].hydro==4 && atmlst[a2].hydro==2)) ewt=-.156;
                else if ((atmlst[a1].hydro==3 && atmlst[a2].hydro==4) || (atmlst[a1].hydro==4 && atmlst[a2].hydro==3)) ewt=.28;
                else printf(" shouldn't see this message!!!  genpot energy function   a1.hydro=%d  a2.hydro=%d\n",atmlst[a1].hydro,atmlst[a2].hydro);

                ewt=ewt*.2;//overall rescaling, to get HP vs HB energy scale balanced correctly
                ratio=rtarsq/rs;
                if (ratio<1.) {en=2.*ratio*ratio-ratio*ratio*ratio*ratio;  den=(2.*2.*ratio-4.*ratio*ratio*ratio)*-1.*ratio/rs; }
                else {en=1.;  den=0; }
                sum=sum+ewt*en;//add hydrophobic energy to total
                grd[a1]=Vadd(grd[a1],Vscalmul(v,2.*ewt*den));
                grd[a2]=Vadd(grd[a2],Vscalmul(v,-2.*ewt*den));
            }

            //Hydrogen bonds
            if ( (strcmp(atmlst[a1].pdbcode,"H")==0 && strcmp(atmlst[a2].pdbcode,"O")==0) || (strcmp(atmlst[a2].pdbcode,"H")==0 && strcmp(atmlst[a1].pdbcode,"O")==
0) ){//are atom i, j, an oxygen and a hydrogen?  check here.
                if (abs(atmlst[a1].aminoacidnum-atmlst[a2].aminoacidnum)>2. ){//Are atom i an j 3 amino acids apart?
                    if (strcmp(atmlst[a1].pdbcode,"H")==0){//whose the hydrogen?  whose the oxygen?
                        an=atmlst[a1].bonded[0]; ah=a1; ao=a2; ac=atmlst[a2].bonded[0];}
                    else if (strcmp(atmlst[a2].pdbcode,"H")==0){//whose the hydrogen?  whose the oxygen?
                        an=atmlst[a2].bonded[0]; ah=a2; ao=a1; ac=atmlst[a1].bonded[0];}
                    else printf(" shouldn't see this message!!!  genpot energy function \n");
                    vnh=Vsub(atmlst[an].r,atmlst[ah].r);
                    v=  Vsub(atmlst[ah].r,atmlst[ao].r);
                    voc=Vsub(atmlst[ao].r,atmlst[ac].r);
```

```
                    bondlengthsquared=1.9*1.9;
                    ewt=-1.5; //HYDROGEN BOND ENERGY SCALE

                    vnhv=Vdot(v,vnh)/(Vdot(vnh,vnh)*Vdot(v,v));
                    vocv=Vdot(v,voc)/(Vdot(voc,voc)*Vdot(v,v));
                    if (vnhv>0. && vocv>0.) {
                        cbs=Vdot(v,vnh)*vnhv;//cos^2 theta_b
                        cas=Vdot(v,voc)*vocv;//cos^2 theta_a
                        ratio=bondlengthsquared/rs;
                        if (ratio<1.) fr=2.*ratio*ratio-ratio*ratio*ratio*ratio;  else fr=1.;
                        if (ratio<1.) dfr=(2.*2.*ratio-4.*ratio*ratio*ratio)*-1.*ratio/rs;  else dfr=0;

                        en=ewt*fr*(cas*cbs);//here's the energy function
                        sum=sum+en;//add hyrdogen bond energy to total

                        dedvnh=Vadd(Vscalmul(v,ewt*fr*cas*2.*vnhv),Vscalmul(vnh,ewt*fr*cas*cbs*-2./Vdot(vnh,vnh)));
                        dedvoc=Vadd(Vscalmul(v,ewt*fr*cbs*2.*vocv),Vscalmul(voc,ewt*fr*cbs*cas*-2./Vdot(voc,voc)));
                        dedv=Vadd(Vadd(Vscalmul(vnh,ewt*fr*cas*2.*vnhv),Vscalmul(voc,ewt*fr*cbs*2.*vocv)),Vscalmul(v,ewt*2.*dfr*cas*cbs-ewt*2.*fr*cbs*cas/Vdot(v,v)-
ewt*2.*fr*cas*cbs/Vdot(v,v)));

                        grd[an]=Vadd(grd[an],Vscalmul(dedvnh, 1.));
                        grd[ah]=Vadd(grd[ah],Vadd(Vscalmul(dedvnh, -1.),Vscalmul(dedv,1.)));
                        grd[ao]=Vadd(grd[ao],Vadd(Vscalmul(dedv,-1.),Vscalmul(dedvoc,1.)));
                        grd[ac]=Vadd(grd[ac],Vscalmul(dedvoc,-1.));
                    }
                }
            }//end hydrogen bond loop
        }//end jth neighbor loop

        //atoms that are bonded are allowed to get pretty close together.  .3*(rad1+rad2) then repulsion kicks in
        for(j=0; j<atmlst[i].num_bonded; j++)//now j goes over atoms ignored by atom i
        {
            a2=atmlst[i].bonded[j];
            v=Vsub(atmlst[a1].r,atmlst[a2].r);
            rs=Vnormsquare(v);//This is how far apart atom i and atom j are
            rtarsq=(atmlst[a1].radius+atmlst[a2].radius)*.3;//minimum distance
            rtarsq=rtarsq*rtarsq;

            //Self avoiding potential
            ewt=2.;
            if(a2==a1) ewt=0.;
            ratio=rs/rtarsq;
            if (ratio<1.) {//too close?
                en=(1.-ratio)*(1.-ratio);
                den=2.*(1.-ratio)*-1./rtarsq;
                sum=sum+ewt*en;//add this steric energy to total
                grd[a1]=Vadd(grd[a1],Vscalmul(v,2.*ewt*den));
                grd[a2]=Vadd(grd[a2],Vscalmul(v,-2.*ewt*den));
                if (ratio<.8 && info->prg >=4)        printf("bonded atoms too close a1=%d  %s in %s num %d   a2=%d  %s in %s num %d    ratio=%f \n",a1, atmlst[a1].pdbcode, atmlst[a1].am
inoacidtype, atmlst[a1].aminoacidnum, a2, atmlst[a2].pdbcode, atmlst[a2].aminoacidtype, atmlst[a2].aminoacidnum, ratio);
            }
        }
    }//end ith atom loop

    for(i=0; i<info->numatoms; i++) grd[i]=Vscalmul(grd[i],1./atmlst[i].rscl);//rescale gradient based on inertia
    return(sum);
}
```

```
#ifndef VECTOR
#define VECTOR

#include <math.h>

typedef struct FVECTOR {double x,y,z;} fvector;

fvector Vadd(fvector a, fvector b);
fvector Vsub(fvector a, fvector b);
fvector Vscalmul(fvector a, double b);
fvector Vectormkr(double xc, double yc, double zc);
double Vdot(fvector a, fvector b);
fvector Vcross(fvector a, fvector b);
double Vnorm(fvector a);
double Vnormsquare(fvector a);

#endif // VECTOR
```

```
#include <math.h>

//defin what a vector is (three doubles)
typedef struct FVECTOR {double x,y,z;} fvector;
//adding vectors
fvector Vadd(fvector a, fvector b){
    fvector c;
    c.x=a.x+b.x; c.y=a.y+b.y; c.z=a.z+b.z;
    return c;}
//subtracting vectors
fvector Vsub(fvector a, fvector b){
    fvector c;
    c.x=a.x-b.x; c.y=a.y-b.y; c.z=a.z-b.z;
    return c;}
//scalar multiplication
fvector Vscalmul(fvector a, double b){
    fvector c;
    c.x=a.x*b; c.y=a.y*b; c.z=a.z*b;
    return c;}
//MAKE a vector
fvector Vectormkr(double xc, double yc, double zc){
    fvector c;
    c.x=xc; c.y=yc; c.z=zc;
    return c;}
//dot product of two vectors
double Vdot(fvector a, fvector b) { return a.x*b.x+a.y*b.y+a.z*b.z; }
//cross product of two vectors
fvector Vcross(fvector a, fvector b){
  fvector c;
  c.x=a.y*b.z-a.z*b.y; c.y=-a.x*b.z+a.z*b.x; c.z=a.x*b.y-a.y*b.x;
  return c;}
//the magnitude of a vector
double Vnorm(fvector a) { return sqrt(a.x*a.x+a.y*a.y+a.z*a.z);}
//the magnitude squared
double Vnormsquare(fvector a) { return a.x*a.x+a.y*a.y+a.z*a.z;}
```

```
#include "./vector.h"


typedef struct {
    double beta;
    double min_dm_err;
    int dm_iters;
    int refine_iters;
    int restartevery;
    int outputprogevery;
    int numaminos;
    int numatoms;
    int numbonds;
    double bond_er;
    int numangles;
    double angle_er;
    int numdets;
    double det_er;
    int numramas;
    double rama_er;
    int chain_proj_iter_max;
    double chain_max_step_size;
    double chain_min_step_size;
    double chain_min_nrg;
    int pot_proj_iter_max;
    double pot_max_step_size;
    double pot_min_step_size;
    double pot_min_nrg;
    int prg;
} proteininfo;

typedef struct {
  int aminoacidnum;
  fvector r;
  char *atomtype;
  char *genpotcode;
  char *pdbcode;
  char *aminoacidtype;
  int hydro;
  double rscl;
  double radius;
  double charge;
  int num_nghbrs;
  int *nghbrs;
  int num_bonded;
  int *bonded;
} atom;

typedef struct {
    int atom1;
    int atom2;
    double tar;
    double act;
    char *type;
    double er;
    double ewt;
} bond;

typedef struct {
  int atom1;
```

```
  int atom2;
  int atom3;
  double tar;
  double act;
  char *type;
  double er;
  double ewt;
} angl;

typedef struct {
  int atom1;
  int atom2;
  int atom3;
  int atom4;
  double tar;
  double act;
  char *type;
  double er;
  double ewt;
} det;

typedef struct RAMA {
  int atom1;
  int atom2;
  int atom3;
  int atom4;
  int atom5;
  int aminoacidnum;
  char *type;
  double phi;
  double psi;
  double sinphi;
  double cosphi;
  double Aphi;
  double sinpsi;
  double cospsi;
  double Apsi;
  double score;
  double er;
} rama;
```

```
all:    clean
        python Setup.py build_ext --inplace

clean:
        @rm -f pyconverter.c *.o *.so *~ core
        @rm -rf build
```

BIBLIOGRAPHY

[1] C. Anfinsen. Principles that govern the folding of protein chains. *Science*, 181: 223–230, 1973.

[2] H. Arkin and T. lik. Comparison of the elp and multicanonical methods in simulation of the heptapeptide deltorphin. *Journal The European Physical Journal B*, 30 (4):577–580, 2002.

[3] M. Bachmann, M. Arkn, and W. Janke. Multicanonical study of coarse-grained off-lattice models for folding heteropolymers. *PHYSICAL REVIEW E*, 71:031906, 2005.

[4] D. Baker and A. Sali. Protein structure prediction and structural genomics. *Science*, 294(5540):93–96, 2001.

[5] H. H. Bauschke, P. L. Combettes, and D. Russel Luke. Hybrid projection-reflection method for phase retrieval. *J. Opt. Soc. Am. A*, pages 1025–1034, 2003.

[6] C. Chothia. One thousand families for the molecular biologist. *Nature*, 357:543 – 544, 1992.

[7] S. Dalal, S. Balasubramanian, and L. Regan. Transmuting alpha helices and beta sheets. *Fold Des*, 2(5):R71–9, 1997.

[8] S. Dalal, S. Balasubramanian, and L. Regan. Protein alchemy: changing beta-sheet into alpha-helix. *Nat Struct Biol*, 4(7):548–52, 1997.

[9] M. W. N. Deininger and B. J. Druker. Specific targeted therapy of chronic myelogenous leukemia with imatinib. *Pharmacol Rev*, 55:401–423, 2003.

[10] D. J. Earl and M. W. Deem. Parallel tempering: Theory, applications, and new perspectives. *Phys. Chem. Chem. Phys.*, 7:3910 – 3916, 2005.

[11] R. Elber and D. Tobi. Distance dependent pair potential for protein folding: Results from linear optimization. *Proteins Structure Function and Genetics*, 41:40–46, 2000.

[12] V. Elser. Phase retrieval by iterated projections. *J. Opt. Soc. Am. A*, 20(1):40–55, 2003.

[13] V. Elser. Solution of the crystallographic phase problem by iterated projections. *Acta Cryst. A*, 59(3):201–209, May 2003.

[14] V. Elser and I. Rankenburg. Deconstructing the energy landscape. *Phys. Rev. E*, 73(026702), February 2006.

[15] V. Elser, I. Rankenburg, and P. Thibault. Searching with iterated maps. *PNAS*, 104: 418–423, 2007.

[16] R. A. Engh and R. Huber. Accurate bond and angle parameters for x-ray protein structure refinement. *Acta Crystallographica*, pages 392–400, 1991.

[17] J. R. Fienup. Reconstruction of an object from the modulus of its fourier transform. *Opt. Lett.*, 3(1):27–29, 1978.

[18] J. R. Fienup. Phase retrieval algorithms: a comparison. *AO*, 21(15):2758–2769, August 1982.

[19] J. R. Fienup, J. C. Marron, T. J. Schulz, and J. H. Seldin. Hubble space telescope characterized by using phase-retrieval algorithms. *Appl. Opt.*, 32:1747, 1993.

[20] R. W. Gerchberg and W. . Saxton. A practical algorithm for the determination of phase from image and diffraction plane pictures. *Optik*, 35:237, 1972.

[21] U. H. E. Hansmann and L. T. Wille. Global optimization by energy landscape paving. *Phys. Rev. Lett.*, 88:068105, 2002.

[22] A. Irbäck, F. Sjunnesson, and S. Wallin. Three-helix-bundle protein in a ramachandran model. *Proc. Natl. Acad. Sci.*, 97:13614–13618, 2000.

[23] G. K. Comparative modeling for protein structure prediction. *Curr Opin Struct Biol*, 16(2):172–7, 2006.

[24] M. Karplus1 and A. McCammon. Molecular dynamics simulations of biomolecules. *Nature Structural Biology*, 9:646 – 652, 2002.

[25] J. Kendrew. Myoglobin and the structure of proteins. *Science*, 139(3561):1259, 1963.

[26] J. Kendrew, G. Bodo, H. Dintzis, R. Parrish, H. Wyckoff, and D. Phillips. A three-dimensional model of the myoglobin molecule obtained by x-ray analysis. *Nature*, 181:662–6, 1958.

[27] M. Khalili, A. Liwo, A.and AJagielska, and H. A. Scheraga. Molecular dynamics with the united-residue model of polypeptide chains. ii. langevin and berendsen-bath dynamics and tests on model helical systems. *J. Phys. Chem. B*, 109:13798 –13810, 2005.

[28] G. J. Kleywegt and T. A. Jones. Phi/psi-chology: Ramachandran revisited. *Structure*, 4(12):1395–1400, 1996.

[29] E. E. Lattman. Sixth meeting on the critical assessment of techniques for protein structure prediction. *Proteins: Structure, Function, and Bioinformatics*, 61:1–2, 2005.

[30] T. Lazaridis and M. Karplus. Discrimination of the native from misfolded protein models with an energy function including implicit solvation. *J. Mol. Biol.*, 288: 477–487, 1999.

[31] C. Levinthal. Are there pathways for protein folding? *Journal de Chimie Physique et de Physico-Chimie Biologique*, 44:65, 1968.

[32] A. L. Lomize, M. Y. Reibarkh, and I. D. Pogozheva. Interatomic potentials and solvation parameters from protein engineering data for buried residues. *Protein Science*, 11:1984–2000, 2002.

[33] J. A. Lukin, G. Kontaxis, V. Simplaceanu, Y. Yuan, A. Bax, and C. Ho. Quaternary structure of hemoglobin in solution. *PNAS*, pages 517–520, 2003.

[34] V. N. Maiorov and G. M. Crippen. Contact potential that recognizes the correct folding of globular proteins. *J Mol Biol.*, 227(3):876–88, 1992.

[35] M. A. Marti-Renom, A. C. Stuart, A. Fiser, R. Sanchez, F. Melo, and A. Sali. Comparative protein structure modeling of genes and genomes. *Annu Rev Biophys Biomol Struct*, 29:291–325, 2000.

[36] J. Miao and D. Sayre. On possible extensions of x-ray crystallography through diffraction-pattern oversampling. *Acta Cryst. A*, 56:596–506, November 2000.

[37] S. Miyazawa and R. L. Jernigan. Estimation of effective interresidue contact energies from protein crystal structures: Quasi-chemical approximation. *Macromolecules*, 18:534–552, 1985.

[38] H. Muirhead and M. Perutz. Structure of haemoglobin. a three-dimensional fourier synthesis of reduced human haemoglobin at 5.5 a resolution. *Nature*, 199:633–8, 1963.

[39] J. N. Onuchic. Theory of protein folding: The energy landscape perspective. *Annual Review of Physical Chemistry*, 48:545–600, 1997.

[40] C. A. Orengo and J. M. Thornton. Protein families and their evolution-a structural perspective. *Annu Rev Biochem.*, 74:867–900, 2005.

[41] A. Orlowski and H. Paul. Phase retrieval in quantum mechanics. *Phys. Rev. A*, 50: R921 – R924, 1994.

[42] I. I. Rabi, J. R. Zacharias, S. Millman, and P. Kusch. A new method of measuring nuclear magnetic moment. *Physical Review*, 53:318–318, 1938.

[43] I. C. Rankenburg and V. Elser. Protein structure prediction by an iterative search method.

[44] H. A. Scheraga, A. A. Liwo, S. Odziej, C. Czaplewski, M. Khalili, J. A. Vila, and D. R. Ripoll. The two aspects of the protein folding problem. *ISBN-13: 978-3-9810843-0-6*, pages 37–44, 2006.

[45] A. Schug, T. Herges, A. Verma, and W. Wenzel1. Investigation of the parallel tempering method for protein folding. *J. Phys.: Condens. Matter*, 17:S1641S1650, 2005.

[46] M. y. Shen and A. Sali. Statistical potential for assessment and prediction of protein structures. *Protein Science*, 15:2507–2524, 2006.

[47] A. Verma, A. Schug, K. H. Lee, and W. Wenzel. Basin hopping simulations for all-atom protein folding. *J. Chem. Phys*, 124:44515, 2006.

[48] A. R. Williamson. Creating a structural genomics consortium. *Nat Struct Biol*, page 953, 2000.

[49] K. Wuthrich. Protein structure determination in solution by nmr spectroscopy. *THE JOURNAL OF BIOLOGICAL CHEMISTRY*, 265(36):22059–22062, 1990.

[50] Y. Zhang and J. Skolnick. The protein structure prediction problem could be solved using the current pdb library. *Proc. Natl. Acad. Sci.*, 102(4):1029–34, 2005.

INDEX