Protein Structure Refinement Algorithms

Thesis by
Mohsen Chitsaz

In Partial Fulfillment of the Requirements for the
degree of
PhD in Molecular Biophysics and Minor in Computer Science

CALIFORNIA INSTITUTE OF TECHNOLOGY
Pasadena, California
2014

(Defended March 14, 2014)

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Professor Dr. Stephen L. Mayo, for being a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I will always look back and appreciate the chance that I was given to have you as my mentor. I would also like to thank Professor William A. Goddard, Professor Pamela J. Bjorkman, and Professor Chris Umans for serving as my committee members. I would also like to thank Dr. Marie Arie for her help with writing manuscripts and her comments on the organization of the text of this thesis. I also would like to thank Rhonda DiGuisto and everyone else in Mayo Lab. I also would like to express my great appreciation and thanks to the graduate office, Dean Joseph Shepherd and Ms. Natalie Gilmore for their invaluable kindness and help with the students in tough times. I would also like to thank Daniel Yoder and ISP for their great help throughout my programs.

Words cannot express how grateful I am to my parents for all of the sacrifices that they have made to provide me with the opportunity to be able to focus on developing myself scientifically and professionally, and explore and investigate my research ideas, and pursue my education. Your prayers and inspirations for me was what made me succeed thus far. I would also like to thank my siblings and my family for all their support, help, inspirations and encouragements. I also would like to greatly thank all my friends: Alireza Mohammad Karim, Bernardo Sosa Padilla Araujo, Alborz Mahdavi, Yelena Tuzova, Niema Pahlevan, Shervin Taghavi, Aydin

Babakhani, Masoud Farivar, Asghar Aryanfar, Vanessa Heckman, and Gokjan Karakus all of whom have made my residence at Caltech very memorable, enjoyable and pleasant.

Abstract

Protein structure prediction has remained a major challenge in structural biology for more than half a century. Accelerated and cost efficient sequencing technologies have allowed researchers to sequence new organisms and discover new protein sequences. Novel protein structure prediction technologies will allow researchers to study the structure of proteins and to determine their roles in the underlying biology processes and develop novel therapeutics.

Difficulty of the problem stems from two folds: (a) describing the energy landscape that corresponds to the protein structure, commonly referred to as force field problem; and (b) sampling of the energy landscape, trying to find the lowest energy configuration that is hypothesized to be the native state of the structure in solution. The two problems are interweaved and they have to be solved simultaneously.

This thesis is composed of three major contributions. In the first chapter we describe a novel high-resolution protein structure refinement algorithm called GRID. In the second chapter we present REMC^{GRID}, an algorithm for generation of low energy decoy sets. In the third chapter, we present a machine learning approach to ranking decoys by incorporating coarse-grain features of protein structures.

Figures and Tables

Nomenclature

RMSD: Root Mean Square Deviation

GRID: A high-resolution refinement algorithm

REMC^{GRID}: Replica Exchange Monte Carlo GRID

AU: Arbitrary Units

SVM: Support Vector Machine

PDB: Protein Data Bank

Residue: an amino acid in the context of a protein structure

ϕ: Dihedral angle formed on the backbone of a residue by C, N, C-alpha, C atoms

ψ: Dihedral angle formed on the backbone of a residue by N, C-alpha, C, N atoms

Side-chain: Chemical group attached to the backbone of the protein, determining the type of amino acid

Conformation: 3-dimensional state of a protein structure in Cartesian space

Forcefield: a scoring function that takes a conformation of a protein and assigns energy to it

Decoy: a conformation of a protein, typically, perturbed state of a protein structure

TABLE OF CONTENTS

Chapter 1

GRID: A High-Resolution Protein Structure Refinement Algorithm

The text of this chapter is adapted from a published review article that was co-authored with Professor Stephen L. Mayo

Chitsaz M, Mayo SL., *J Comput Chem* 2013; **34**:445–450.

Abstract

The energy-based refinement of protein structures generated by fold prediction algorithms to atomic-level accuracy remains a major challenge in structural biology. Energy-based refinement is mainly dependent on two components: (1) sufficiently accurate force fields, and (2) efficient conformational space search algorithms. Focusing on the latter, we developed a high-resolution refinement algorithm called GRID. It takes a three-dimensional protein structure as input and, employing an all-atom force field, attempts to improve the energy of the structure by systematically perturbing backbone dihedrals and side chain rotamer conformations. We compare GRID to Backrub, a stochastic algorithm that has been shown to predict a significant fraction of the conformational changes that occur with point mutations. We applied GRID and Backrub to 10 high-resolution (≤ 2.8 Å) crystal structures from the Protein Data Bank and measured the energy improvements obtained and the computation times required to achieve them. GRID resulted in energy improvements that were significantly better than those attained by Backrub while expending about the same amount of computational resources. GRID resulted in relaxed structures that had slightly higher backbone RMSDs compared to Backrub relative to the starting crystal structures. The average RMSD was 0.25 ± 0.02 Å for GRID versus 0.14 ± 0.04 Å for Backrub. These relatively minor deviations indicate that both algorithms generate structures that retain their original topologies, as expected given the nature of the algorithms.

Key words: protein structure refinement; flexible backbone; energy-based refinement; conformational search; backrub motion

Introduction

Although much progress has been made in recent years, the ability to predict the three-dimensional structure of a protein from its amino acid sequence remains a major challenge in structural biology. Good prediction techniques would allow us to obtain the structures of novel protein sequences without having to determine them experimentally and would be particularly useful for membrane and other proteins whose structures are experimentally difficult to obtain or are highly diverged from the set of proteins with existing structures[1]. The knowledge obtained from accurate protein structure prediction could improve our understanding of how proteins function, aid in the discovery of proteins with new and improved properties, and facilitate the development of new drugs[2, 3]. Many biological applications, including studies on enzymatic activity, virtual ligand screening, structure-based protein function prediction, and structure-based drug design require knowledge of protein structures at atomic resolution[1]. In addition, the expansion of high-throughput genome sequencing projects has led to a continually widening gap between protein sequence data and the number of solved structures. Thus, there is a significant need for computational techniques that can model protein structures with atomic accuracy comparable to experimental methods.

Protein structure prediction methods can be broadly classified into three main categories: comparative modeling, threading, and template-free modeling[4]. In comparative modeling, the protein structure is built by using sequence alignments to find the best matching evolutionarily related protein that has a solved structure (the template)[4]. Most structures constructed using this approach have a root mean square deviation (RMSD) of 1-2 Å from the experimental structure[5]. In the absence of any evolutionarily related pro-

teins, threading can be used, which matches the target protein directly to the sequences of solved three-dimensional structures. Threading methods often identify correct templates and provide models with an RMSD of 2-6 Å, with inaccuracies occurring mainly in the loop regions[6]. For targets that have no structurally-related solved proteins (< 30% sequence similarity), template-free modeling methods are used; that is, the model is built from scratch[5, 7]. Successful prediction using template-free modeling is limited to small proteins (< 120 residues), with accuracy in the range of 4-8 Å[8]. Much effort has been expended in the last several decades to achieve the current levels of success in each of these three categories. However, the resulting models are still low-resolution and further refinement is needed in order to achieve the atomic-level accuracy required for most biological applications. Refinement of low-resolution structures to higher resolution models therefore remains a major challenge in the field of protein structure prediction[1, 4, 9-11]. Recognition of the importance of refinement is reflected by the addition of a refinement category in the eighth Critical Assessment of Techniques for Protein Structure Prediction (CASP8)[12].

Approaches to high-resolution refinement have included molecular mechanics energy minimization and Monte Carlo methods including replica exchange Monte Carlo[1, 13-16]. Knowledge-based statistically derived potentials have also been incorporated into these techniques[17]. Recently, a flexible backbone modeling algorithm was applied[18-20]. All these methods have attained some degree of success; however, none has emerged as clearly superior[13, 14, 16]. Current methods require improvements in terms of computational efficiency and/or accuracy[18, 20-23]. For those approaches that generate an ensemble containing a large number of low-resolution structures, refinement algorithms must be very effi-

cient in processing many structures in a reasonable amount of time. In addition, the refined structure must accurately reflect the naturally folded state. Because the minimum energy configuration is considered to be the best estimate of native structure[24,25], high-resolution refinement methods must be able to efficiently find lower energy configurations. The energy improvement attained relative to the starting structure is a measure of a method's progress towards a model consistent with the experimental data, assuming an accurate energy function. Thus, refinement procedures typically employ multiple refinement cycles in which the energy of a low-resolution structure is iteratively reduced. The energy of the structure is calculated based on molecular mechanics force fields that describe the underlying physics and chemistry of protein molecules[14, 16]. Knowledge-based force fields are also commonly used[26].

To address the problems of computational efficiency and accuracy, we devised a high-resolution refinement algorithm called GRID. The method sequentially applies perturbations to a structure's φ and ψ angles and side chain rotamers, accepting perturbations that produce the most improvement in the energy of the structure. Unlike many refinement methods that rely on stochastic techniques, GRID has been tuned to use a single pass search strategy and therefore requires less computational effort. We compared GRID to Backrub[27], a flexible backbone modeling method that is able to predict most of the small conformational changes associated with point mutations[18, 20]. GRID and Backrub were applied to a set of 10 high-resolution crystal structures from the PDB, and the algorithms were assessed in terms of energy improvement and computational efficiency.

Computational Methods

GRID refinement algorithm

GRID is a refinement method that takes a three-dimensional protein structure as input and, employing an all-atom force field, attempts to improve the energy of the structure using a move set that allows for backbone and side chain flexibility (Fig. 1). Three degrees of freedom are considered: φ and ψ angle perturbations and side-chain rotamer optimization. The algorithm starts from the first residue (N terminus) and evaluates all combinations of Δφ, Δψ, and rotamer, given a specified maximum perturbation angle and precision level (perturbation angle step size). Δφ and Δψ perturbations are applied to the C-terminal protein segment. The continuous space of Δφ, Δψ perturbations is broken down into discrete grid points around the current φ and ψ angles of the residue. Similarly, a rotamer library is used to define a discrete set of rotamers for the residue. For each combination of Δφ, Δψ, and rotamer available for a given residue, the algorithm perturbs the structure and calculates the energy of the trial configuration (the rest of the molecule is treated as a rigid body). This information is used to fill out a three-dimensional matrix that reflects the discretized energy space for the residue. The move that results in the lowest energy is then applied to the structure before proceeding to the next residue. GRID marches forward from N to C terminus, sequentially calculating the energy matrix and applying the lowest energy move on each of the residues. When the last residue is reached, the algorithm can terminate or return to the N terminus for a subsequent iteration.

Note that although the moves are restricted to local perturbations relative to a single residue, the perturbation selected influences the entire structure. Thus, relatively minor local changes can propagate down the chain resulting in larger shifts in the conformation

of the protein. In order to further improve the energies, at the end of the procedure, the algorithm walks through the structure once (from N to C terminus) and performs four steps of local conjugate gradient energy minimization on each residue while keeping the rest of the structure fixed in Cartesian space.

GRID implementation and parameterization

The GRID refinement algorithm was developed in C++ and incorporated into the Triad protein design software package[33]. The Rosetta force field with default parameters[28] was used to calculate the energy of the protein conformations. Rotamers were selected from a backbone-dependent side-chain rotamer library[29].

As noted above, GRID takes as input four main parameters: (1) the maximum perturbation angle (Δφ, Δψ) for each iteration, (2) the precision level (step size for Δφ and Δψ) for each iteration, (3) the rotamer library used for each iteration, and (4) the number of iterations. GRID was run on 2-10 test proteins using several combinations of these parameters to determine the optimal protocol for refinement. Values for maximum Δφ and Δψ ranged from ±0.1° to ±8.0° and precision levels ranged from 0.1° to 8.0°, forming Δφ × Δψ matrices containing from 9 to 441 grid points; the number of iterations ranged from 1 to 10. The same set of rotamers (from the Dunbrack backbone-dependent rotamer library[29]) was used in every case.

Backrub algorithm

The Backrub concept was developed by the Richardson lab[27] and was implemented by the Kortemme[18, 20, 30] and Donald[31] groups. Its move set was derived from the observation

that ultra-high-resolution crystal structures often show alternate conformations in which subtle local backbone motion is coupled to much larger two-state changes in side-chain conformation[18, 20, 21]. This “backrub” motion supplies a common type of local plasticity in protein backbones. A refinement algorithm based on the Backrub concept performs 10,000 steps of Monte Carlo simulation using Metropolis acceptance criteria. In every step, Backrub employs two types of moves with equal probability: (1) it replaces a side-chain rotamer (randomly chosen from a specified rotamer library) at a randomly chosen residue position, or (2) it performs a Backrub move in which a randomly chosen 2- or 3-residue fragment is rotated about the axis defined by the C_α atoms of the first and the last residue of the fragment. The bond angles extending from the C_α atoms of the first and the last residues are then optimized using a bond angle potential. We used the Rosetta version 3.1 implementation of Backrub, with parameters as described in Friedland et al[18].

Test sets and assessment criteria

GRID and Backrub were initially tested on 10 single-chain high-resolution (≤ 2.8 Å) crystal structures from the PDB representing proteins of various sizes and folds (α, β, α+β, and α/β) (Table 1). Two criteria were used to compare the algorithms: (1) energy improvement (relative to that of the starting structure), and (2) computation time. The validity of the first criterion is based on the assumption that the minimum energy configuration is the best estimate of the native state[24, 32] and that the degree of energy improvement attained reflects the method's success in moving toward the native form and, perhaps more importantly, allowing selection of the best structural models from an ensemble of predicted structures.

Results

GRID parameterization: smaller perturbations of φ and ψ allow more global structure refinement

Initial trials on 2–10 test proteins indicated that refinements done using a single iteration and the lowest precision level, a 3×3 Δφ × Δψ matrix, produced energy improvements that were at least 70% of the values obtained with larger matrices (11 × 11) and multiple iterations (up to 10). These initial studies also showed that perturbations of φ and ψ greater than ± 2.0° caused no significant energy improvement over GRID refinements performed using smaller perturbations. More extensive tests were performed using smaller perturbation angles that ranged from ± 0.1° to ± 1.9°, and all of these resulted in the same level of energy improvement (Fig. S1). Completely eliminating φ and ψ perturbations (i.e., sidechain rotamer optimization alone) was less effective at energy improvement (Fig. S2). We then conducted a post-refinement analysis of the moves that had been applied to achieve the final refined structures (Fig. 2). The moves were divided into four types that differ in their effects on protein structure: type A, the backbone is not perturbed ($\Delta\varphi = 0, \Delta\psi = 0$); type B, only one of the dihedrals is perturbed ($\Delta\varphi = 0, \Delta\psi \neq 0$) or ($\Delta\varphi \neq 0, \Delta\psi = 0$); type C, both dihedrals are perturbed by the same amount in the same direction ($\Delta\varphi = \Delta\psi$); and type D, both dihedrals are perturbed by the same amount, but in opposite directions ($\Delta\varphi = -\Delta\psi$). Side chain rotamer optimization was allowed in all cases. The frequency of occurrence of each of these different types of moves was then calculated. As seen in Figure 2, move type A (no backbone perturbation) became increasingly dominant at larger perturbation angles, representing up to 85% of all the moves ap-

plied. The next most common move was type B, followed by type D, then type C; this trend was consistent at all perturbation angles tested. At the smallest perturbation angle (0.1°), the type A move (no backbone perturbation) was no longer dominant and occurred at about the same frequency as types B and D (~27%).

In choosing the parameters for refinement with GRID, one of the criteria was that the perturbations be fairly evenly dispersed throughout the protein structure and not restricted to specific regions. We found that using the smallest perturbation (± 0.1°) for Δφ and Δψ gave the best results (i.e., the frequency of dihedral angle change was similar for all residues), whereas with larger perturbations, most of the moves caused no backbone movement (only the rotamer was changed), and moves that did perturb the backbone were restricted to residues near the termini (Fig. S3). These results led us to choose a maximum perturbation angle of ± 0.1°. To minimize computation time, the lowest values were also used for the number of iterations and for the precision level. The final GRID refinement parameters are therefore as follows: maximum perturbation angle (Δφ, Δψ) = ± 0.1°, precision level = 0.1°, and iterations = 1, with rotamers from the Dunbrack backbone-dependent rotamer library[29].

GRID outperforms Backrub

As shown in Figure 3a, refinement with GRID resulted in structures with energy improvements ranging from ~100 to ~600 kcal/mol. In every case, these values were significantly larger than those obtained with a single trajectory of Backrub. Increasing the number of Backrub trajectories to 10, 100, or 1000 caused minimal additional improvement. The four steps of local conjugate gradient minimization that is applied at the end of the GRID pro-

cedure has a significant impact on the resulting energies, increasing the overall energy improvement by about a factor of 2 (Fig. S4). The computation time expended by GRID was comparable to that required for one trajectory of Backrub (Fig. 3b); run times ranged from 45 sec to about 6 minutes, averaging ~3 minutes on a single CPU core (Intel Xeon at 2.33 GHz). As expected, computational effort increased linearly with the number of Backrub trajectories, with 1000 trajectories taking ~2400 minutes for the largest protein in the test set (155 residues).

Plots of energy improvement versus protein size show an enhanced effect (greater improvement) as protein size increases; this trend is seen for both algorithms, but is more pronounced for GRID (Fig. 3c). As expected, computation time scales linearly with the number of residues for both Backrub and GRID (Fig. 3d). Taken together, these results demonstrate that GRID outperforms Backrub for high-resolution structure refinement, achieving substantially greater energy improvements in every case while expending comparable computational effort.

Calculation of the backbone RMSD of the final refined structure with respect to the starting crystal structure for each of the ten proteins shows that GRID produces somewhat larger deviations than Backrub; the average RMSD was 0.25 ± 0.02 Å for GRID versus 0.14 ± 0.04 Å for Backrub (Fig. 4). These relatively minor deviations indicate that both algorithms generate refined structures that retain their original topologies.

An implementation of GRID that uses random residue order for applying perturbations was also run and found to produce nearly identical results to the sequential N terminus to C terminus version described above (Fig. S5).

Discussion

Our results indicate that the GRID algorithm is more effective in reducing the energy of structures than the Backrub algorithm: applying the same all-atom force field, GRID consistently yields lower energy configurations. Although stochastic algorithms have been successful in refining small proteins (< 85 residues), refinement of larger proteins has proven to be particularly difficult[3]. In contrast, GRID was effective on all the proteins tested, which included structures up to 155 residues. GRID also achieves these improvements with relatively little computational resources (equivalent to one trajectory of Backrub), thus making it feasible to use on even larger proteins. This level of computational efficiency is essential for the last stage of protein structure prediction, since low-resolution prediction techniques typically produce a large number of structures that must be refined in order to select the best structural model. GRID's high performance capabilities stem from using a strategy that is straightforward, but effective.

To achieve structures that are closer to the native state, refinement methods must be able to find lower energy configurations. This can be problematic for algorithms that use local perturbations to achieve refinement, as they are more likely to fall into local minima. It is possible that GRID minimizes this problem because it applies local perturbations that are propagated down the whole protein chain. These perturbations, which are effectively more global in nature, have a higher probability of explicitly moving the structure out of a local minimum. Thus, GRID may allow for a more thorough exploration of the energy landscape, making it easier to find lower energy structures.

Acknowledgments

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and a Department of Defense National Security Science and Engineering Faculty Fellowship (S.L.M). I would like to thank Marie Ary for her comments and assistance in writing the manuscript.

References

[1] G. Chopra, C. M. Summa, M. Levitt, *Proc. Natl. Acad. Sci. USA* **2008**, *105*, 20239-20244.

[2] D. Baker, A. Sali, *Science* **2001**, *294*, 93-96.

[3] P. Bradley, K. M. S. Misura, D. Baker, *Science* **2005**, *309*, 1868-1871.

[4] Y. Zhang, *Curr. Opin. Struct. Biol.* **2009**, *19*, 145-155.

[5] V. Mariani, F. Kiefer, T. Schmidt, J. Haas, T. Schwede, *Proteins* **2011**, *79 Suppl 10*, 37-58.

[6] R. Jauch, H. C. Yeo, P. R. Kolatkar, N. D. Clarke, *Proteins* **2007**, *69 Suppl 8*, 57-67.

[7] J. Xu, J. Peng, F. Zhao, *Proteins* **2009**, *77 Suppl 9*, 133-137.

[8] J. Kopp, L. Bordoli, J. N. Battey, F. Kiefer, T. Schwede, *Proteins* **2007**, *69 Suppl 8*, 38-56.

[9] I. H. Park, V. Gangupomu, J. Wagner, A. Jain, N. Vaidehi, *J. Phys. Chem. B* **2012**, *116*, 2365-2375.

[10] O. Graña, D. Baker, R. M. MacCallum, J. Meiler, M. Punta, B. Rost, M. L. Tress, A. Valencia, *Proteins* **2005**, *61 Suppl 7*, 214-224.

[11] C. Venclovas, A. Zemla, K. Fidelis, J. Moult, *Proteins* **2003**, *53 Suppl 6*, 585-595.

[12] J. L. MacCallum, L. Hua, M. J. Schnieders, V. S. Pande, M. P. Jacobson, K. A. Dill, *Proteins* **2009**, *77 Suppl 9*, 66-80.

[13] S. Kannan, M. Zacharias, *Proteins* **2007**, *66*, 697-706.

[14] S. Kannan, M. Zacharias, *J. Struc. Biol.* **2009**, *166*, 288-294.

[15] M. Levitt, S. Lifson, *J. Mol. Biol.* **1969**, *46*, 269-279.

[16] A. Shmygelska, M. Levitt, *Proc. Natl. Acad. Sci. USA* **2009**, *106*, 1415-1420.

[17] C. M. Summa, M. Levitt, *Proc. Natl. Acad. Sci. USA* **2007**, *104*, 3177-3182.

[18] G. D. Friedland, A. J. Linares, C. A. Smith, T. Kortemme, *J. Mol. Biol.* **2008**, *380*, 757-774.

[19] M. S. Lin, T. Head-Gordon, *J. Comput. Chem.* **2011**, *32*, 709-717.

[20] C. A. Smith, T. Kortemme, *J. Mol. Biol.* **2008**, *380*, 742-756.

[21] T. E. Creighton, *Biochem. J.* **1990**, *270*, 1-16.

[22] K. A. Dill, *Biochemistry* **1990**, *29*, 7133-7155.

[23] H. Venselaar, R. P. Joosten, B. Vroling, C. A. B. Baakman, M. L. Hekkelman, E. Krieger, G. Vriend, *Eur. Biophys. J.* **2010**, *39*, 551-563.

[24] G. I. Makhatadze, P. L. Privalov, *Adv. Protein. Chem.* **1995**, *47*, 307-425.

[25] O. B. Ptitsyn, *Adv. Protein. Chem.* **1995**, *47*, 83-229.

[26] K. T. Simons, C. Kooperberg, E. Huang, D. Baker, *J. Mol. Biol.* **1997**, *268*, 209-225.

[27] I. W. Davis, W. B. Arendall, D. C. Richardson, J. S. Richardson, *Structure* **2006**, *14*, 265-274.

[28] C. A. Rohl, C. E. Strauss, K. M. Misura, D. Baker, *Methods Enzymol.* **2004**, *383*, 66-93.

[29] R. L. Dunbrack, F. E. Cohen, *Protein Sci.* **1997**, *6*, 1661-1681.

[30] E. L. Humphris, T. Kortemme, *Structure* **2008**, *16*, 1777-1788.

[31] I. Georgiev, D. Keedy, J. S. Richardson, D. C. Richardson, B. R. Donald, *Bioinformatics* **2008**, *24*, i196-204.

[32] C. B. Anfinsen, *Science* **1973**, *181*, 223-230.

[33] Protabit, www.protabit.com.

[34] W. Kabsch, C. Sander, *Biopolymers* **1983**, *22*, 2577-2637.

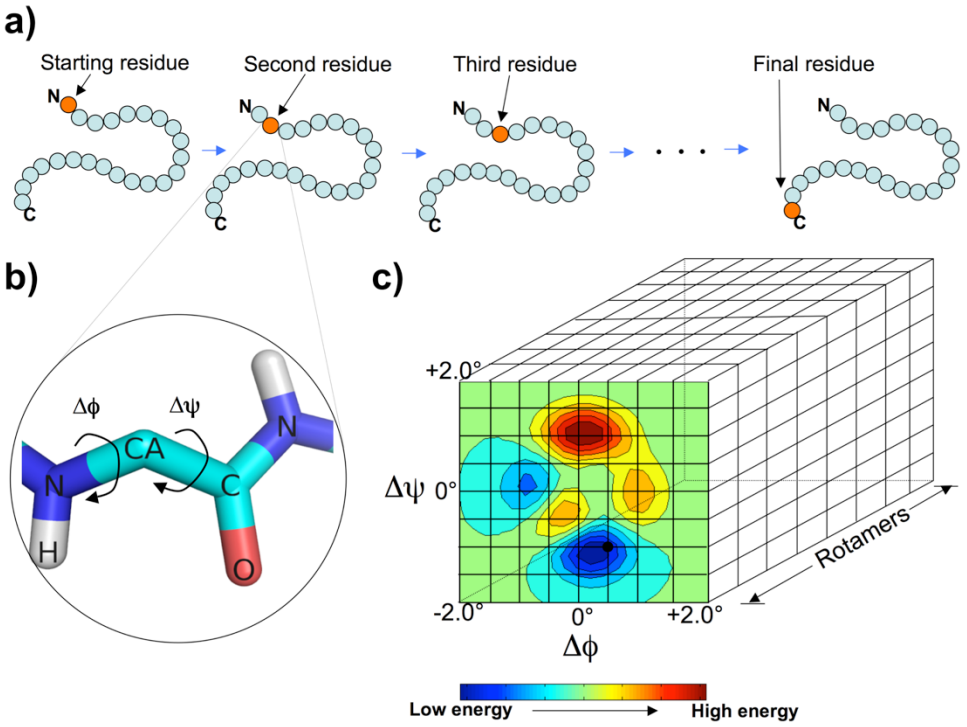


Figure 1. Schematic showing the GRID algorithm. a) In each iteration, GRID starts from the N-terminal residue and marches forward to the C terminus, sequentially applying the lowest energy move on the structure before proceeding to the next residue. b) For each residue, GRID evaluates all the possible perturbations (combinations of Δφ, Δψ, and rotamer) given a specified maximum perturbation angle and precision level, then applies the move that results in the lowest energy. c) Three-dimensional energy matrix (grid) that corresponds to all the possible perturbations (Δφ, Δψ, and rotamer) relative to the current configuration of a residue. In this example, the maximum perturbation angle for Δφ and Δψ is ± 2.0°, the precision level is 0.5°, and 8 rotamers are specified from the backbone dependent rotamer library.

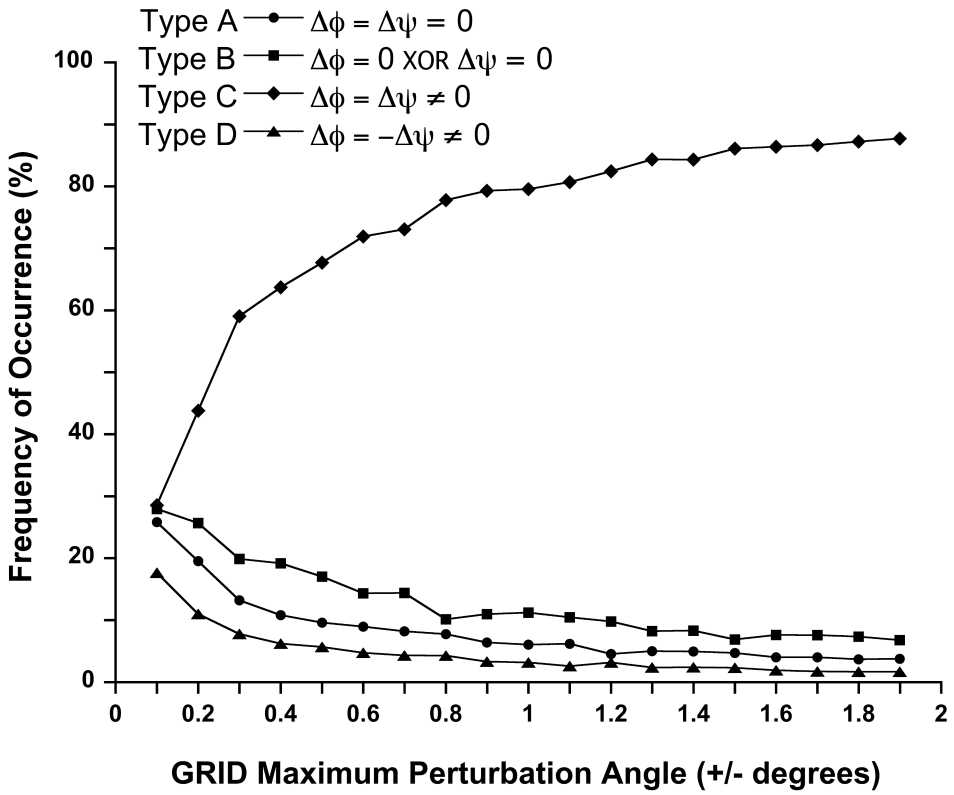


Figure 2. Post-refinement analysis of GRID moves for different maximum dihedral angle perturbations. A single iteration of GRID was run on each of the 10 test proteins using maximum perturbation angles ranging from ± 0.1° to ± 1.9°; the precision level was always set to form the smallest Δφ × Δψ matrix (9 grid points). GRID moves were divided into four types: no perturbation of the backbone (type A), perturbation of φ or ψ, but not both (type B), perturbation of both φ and ψ by the same amount in the same direction (type C), and perturbation of both φ and ψ by the same amount, but in opposite directions (type D). The frequency of occurrence of each of the four different types of moves was calculated; points are the average for the 10 test proteins.

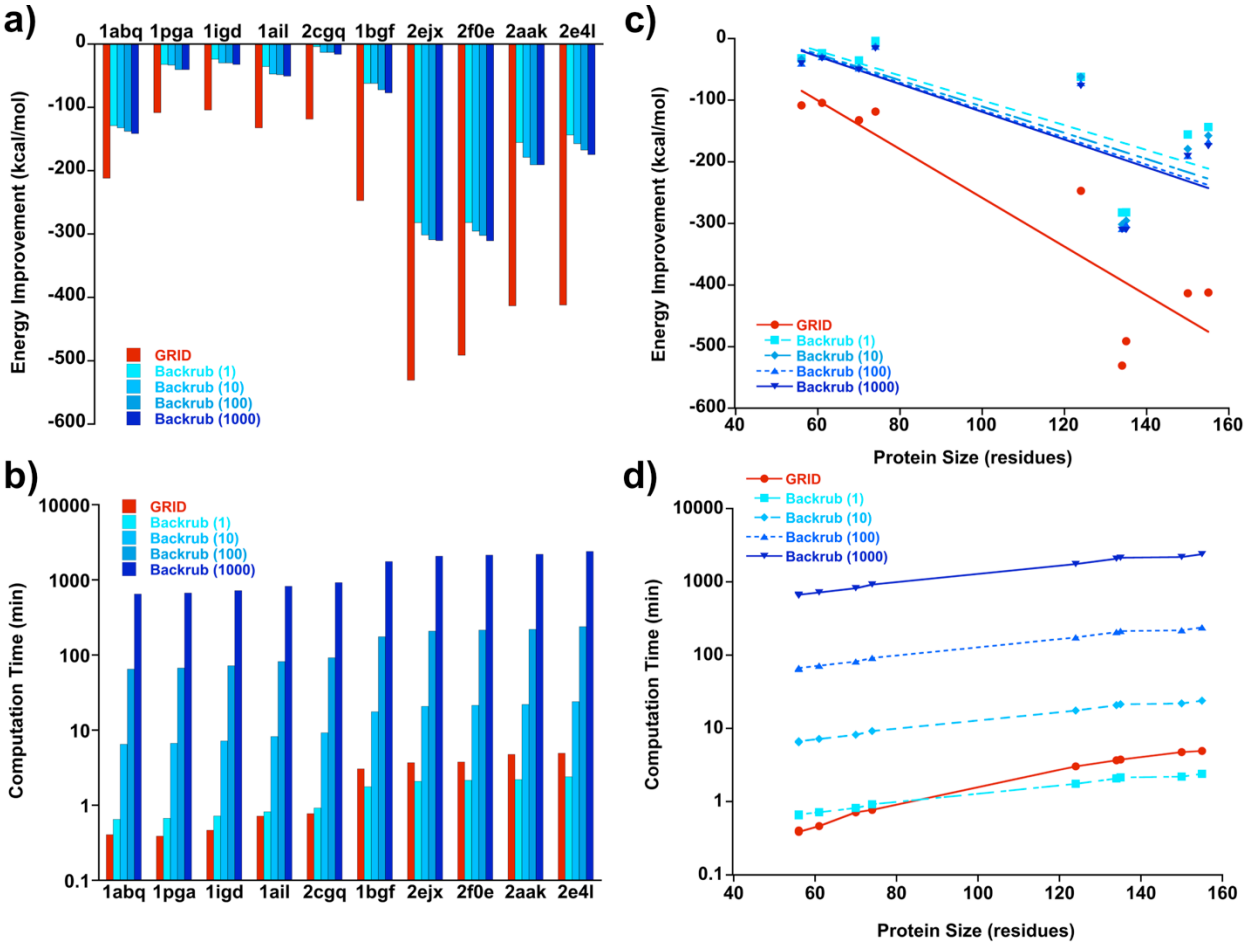


Figure 3. Performance of GRID versus Backrub in refining 10 high-resolution crystal structures from the PDB. Energy improvement (a) and computation time (b) are shown for each of the 10 proteins. For Backrub, which is a stochastic algorithm, values are given for 1, 10, 100, and 1000 trajectories. Energy improvement (c) and computation time (d) are also plotted as a function of protein size.

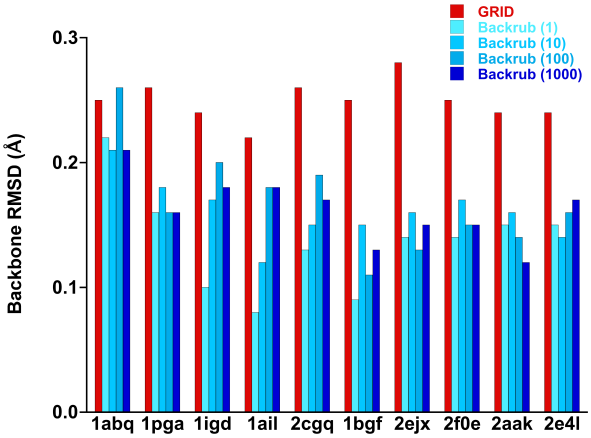


Figure 4. Backbone RMSD with respect to starting crystal structure following refinement with GRID and Backrub. Both algorithms keep the refined structures close to their original topologies.

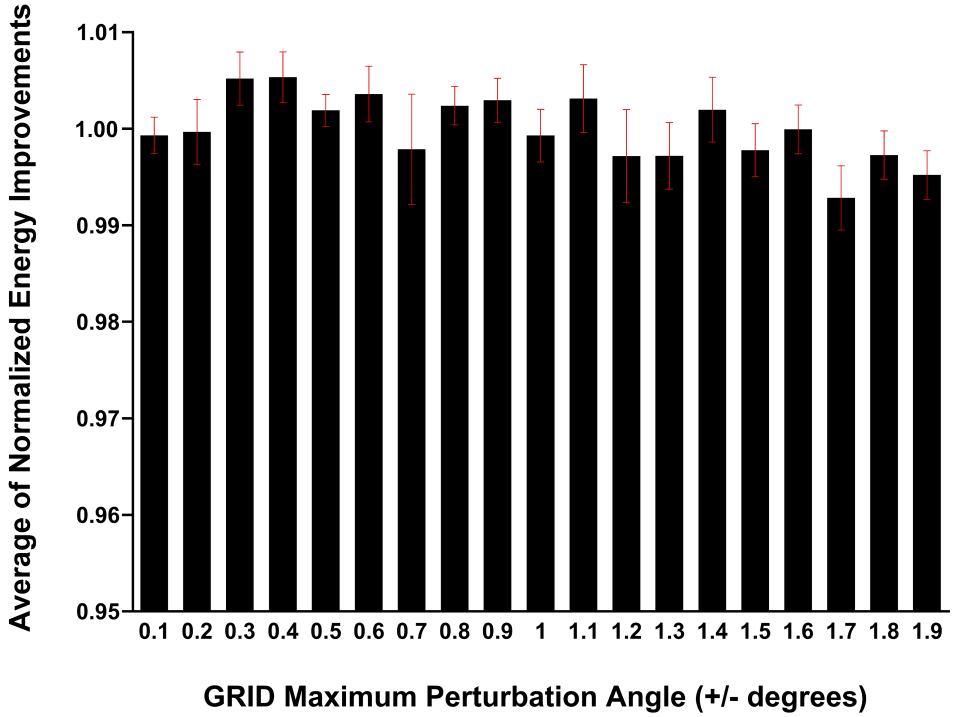
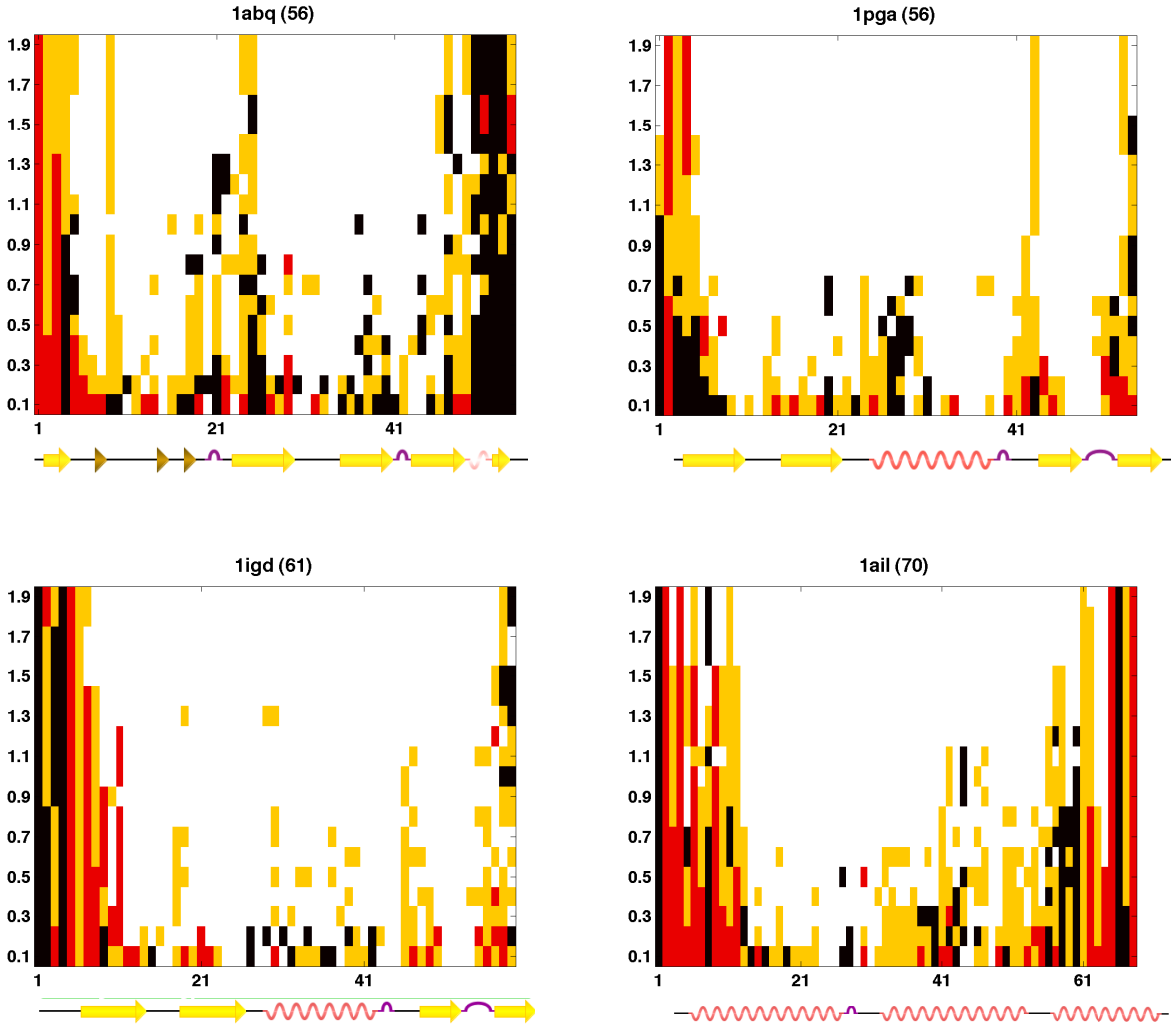
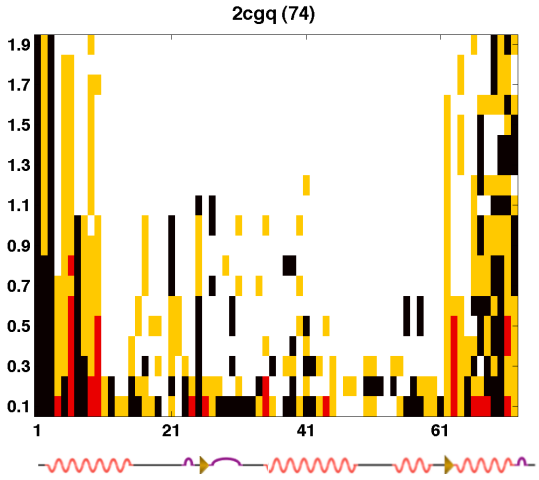
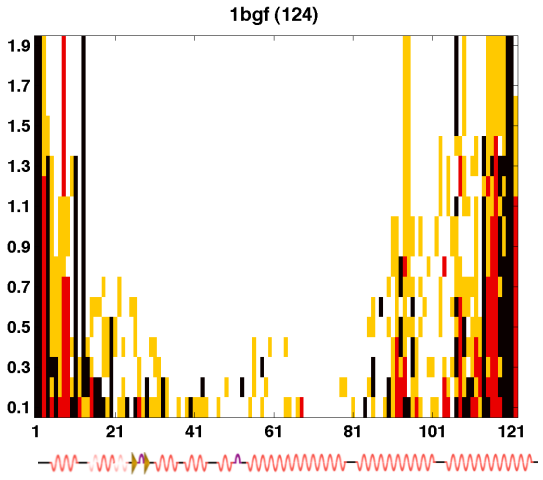
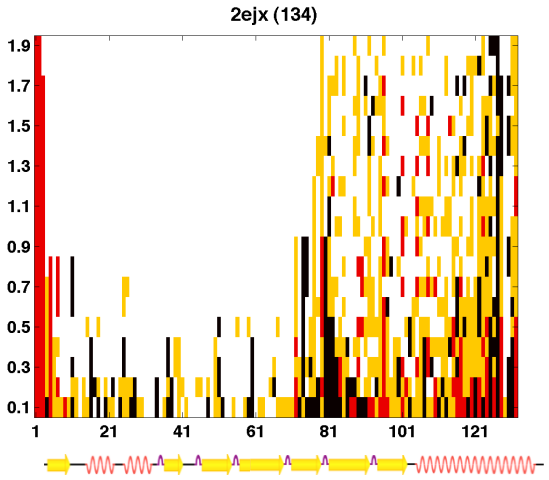


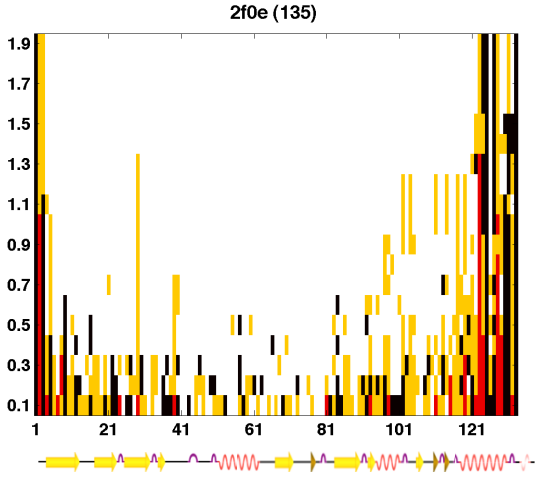
Figure S1. Energy improvement following refinement with GRID using maximum dihedral angle perturbations ranging from ± 0.1° to ± 1.9°; a single iteration was performed and the precision level was set to produce a 9-point Δφ × Δψ matrix (3 × 3). For each of the 10 proteins in the test set, the energy improvement (kcal/mol) was averaged over all the perturbation angles and normalized. Averages of these normalized values for all 10 proteins ± S.E. are shown.

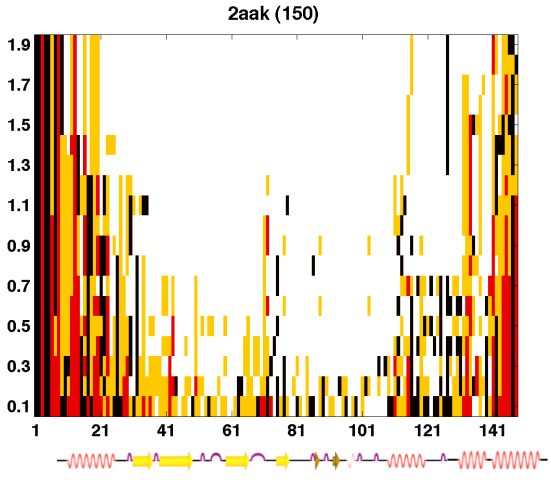

1abq (56)
1pga (56)
1igd (61)
1ail (70)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61
Δϕ = Δψ = 0
Δϕ = 0 XOR Δψ = 0
Δϕ = Δψ ≠ 0
Δϕ = −Δψ ≠ 0


2cgq (74)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61


1bgf (124)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61
81
101
121


2ejx (134)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61
81
101
121


2f0e (135)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61
81
101
121


2aak (150)
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
0.3
0.1
1
21
41
61
81
101
121
141

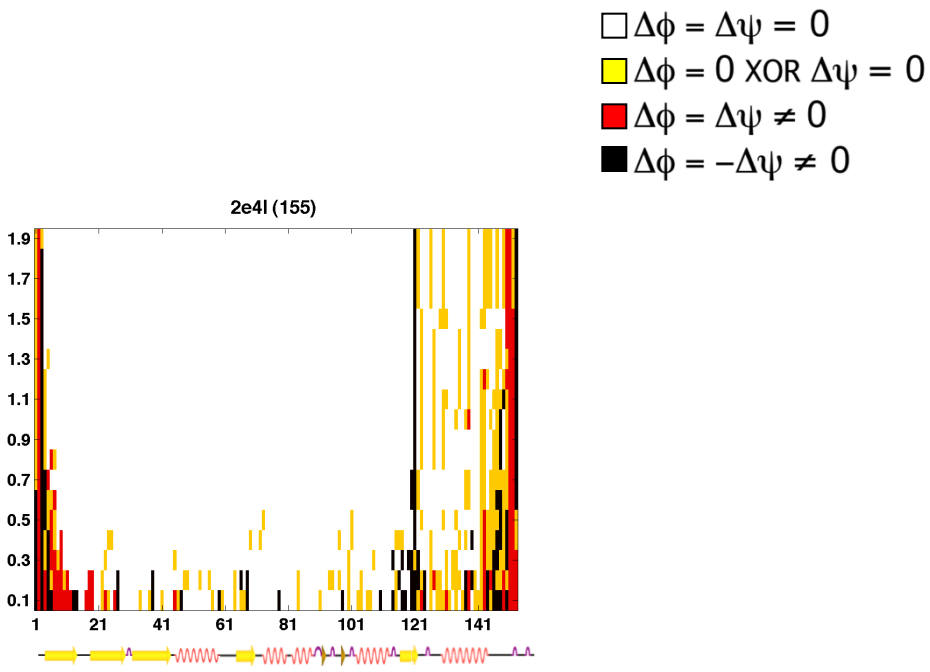


Figure S2. GRID moves plotted as a function of residue number (x axis) and maximum dihedral angle perturbation (y axis) for each of the 10 proteins tested (PDB ID and length of protein are given at top of each graph). The precision level was always set to form the smallest Δφ × Δψ matrix (9 grid points) and a single iteration was performed. The four move types are as follows: type A (white), type B (yellow), type C (red), and type D (black). At larger perturbation angles, moves that cause no backbone movement (type A) become increasingly dominant, and moves that actually perturb the backbone (types B–D) are restricted to residues near the termini. Smaller perturbation angles allow movement to occur more uniformly throughout the entire protein structure.

Table 1. Proteins used as test set for structure refinement.

PDB ID	No. of residues	Resolution (Å)	Class	% α	% β
1abq	56	2.80	β	4	43
1pga	56	2.07	α+β	25	42
1igd	61	1.10	α+β	22	42
1ail	70	1.90	α	80	0
2cgq	74	1.83	α	38	1
1bgf	124	1.45	α	79	1
2ejx	134	1.79	α+β	35	41
2f0e	135	1.95	α+β	24	28
2aak	150	2.40	α+β	35	19
2e4l	155	2.00	α/β	34	29

[a]Classifications based on Dictionary of Secondary Structure of Proteins (DSSP)

Chapter 2

REMC^{GRID}: A Sampling Algorithm for Protein Structure Refinement and Decoy Set Generation

Abstract

We present replica exchange Monte Carlo (REMC)^{GRID}, an algorithm that incorporates the core strategies of GRID[1], a previously developed refinement algorithm, with REMC sampling. GRID uses a deterministic approach to high-resolution refinement, improving the energy of a protein structure by systematically perturbing the backbone dihedrals and side-chain conformations. In this study, REMC^{GRID} was used for two purposes: (1) structure refinement, and (2) generation of decoy structures. In both applications, REMC^{GRID} takes a protein structure as input and generates a conformationally diverse set of low-energy perturbed structures. One can specify the level of dihedral angle perturbations applied and the probability of accepting a perturbed conformation using θ^{max} and kT_f parameters. For refinement, we used 10 high-resolution (≤2.8 Å) crystal structures from the Protein Data Bank (PDB) [2]. The energy improvements obtained and the computational effort required to achieve them were compared for REMC^{GRID} and GRID alone. When longer compute times are allowed, REMC^{GRID} results in energy improvements that are significantly better than those attained by GRID. For decoy generation, we used 59 high-resolution (≤2.8 Å) crystal structures from the PDB of proteins ranging in size from 50-150 residues. We applied REMC^{GRID} with 25 different parameter sets specifying different φ/ψ perturbation angles and acceptance probabilities to generate 472,000 decoys. We computed the backbone RMSD of each decoy with respect to its crystal structure and found that REMC^{GRID} can generate a uniform distribution of perturbed structures covering a broad range of RMSDs (0–10+ Å) that is independent of protein size and class. These characteristics (uniformity and broadness) are important for the proper performance of machine learning algorithms.

Key words: decoy set generation; protein structure refinement; flexible backbone; energy-based refinement; conformational search; replica exchange Monte Carlo

Introduction

Computational modeling of protein structures continues to be essential for advancing research in biology and drug discovery, especially given that experimental structures for many biologically important targets are difficult to obtain. Although investigators have focused efforts on protein structure prediction for over half a century, the accurate and reliable prediction of protein structures still remains a major challenge [3]. A successful structure prediction algorithm must solve two problems: (1) inaccuracies in the force field, and (2) inefficient sampling of the conformational space [4]. The two problems are inter-related; for example, one needs an efficient sampling algorithm in order to validate improvements in a force field [5]. Several groups have worked on developing force fields [6-15], and creating better sampling methods [16-22].

One of the major problems with current force fields is that they are too sensitive[6,23]. Small perturbations in the configuration of a structure can result in large changes in its energy. For instance, a few atomic clashes can result in an unfavorable energy for the entire structure, even though the overall fold might be very close to the native state. This imposes ruggedness on the energy landscape and makes it hard for sampling algorithms to search the conformational space and find the lowest energy states of the structure without getting trapped in local minima. Coarse-grained approaches aim to decrease the sensitivity of current force fields by using a simpler model to describe proteins and their interatomic interactions[24]. To test the validity of any coarse-grained model, one needs a decoy set that covers a broad range of RMSDs so that the model's ability to properly rank the decoys can be tested. We therefore set out to create a decoy-generating algorithm that could provide a similar distribution of perturbed structures spanning a broad range of

RMSD's for proteins of different sizes and classes. We were able to achieve this by incorporating a previously developed deterministic refinement algorithm (GRID) with replica exchange Monte Carlo sampling in a novel algorithm called REMC^{GRID}.

Results and Discussion

REMC^{GRID} algorithm

REMC^{GRID} is a sampling algorithm that takes a three-dimensional protein structure as input and, employing an all-atom force field, perturbs the structure using a move set that allows for backbone and side chain flexibility (see Fig. 1). Three degrees of freedom are considered: φ and ψ dihedral perturbations and side-chain conformation. The algorithm runs multiple Monte Carlo simulations (replicas) in parallel at different temperatures. After a specified number of steps of Monte Carlo simulation, replicas join and stochastically swap configurations based on their energies and the temperatures they are running at according to the Metropolis-Hastings criterion[25] (see Fig. 6). Each step of Monte Carlo simulation consists of four stages: (1) picking a residue at random, (2) perturbing φ and ψ backbone dihedrals of that residue, given a specified maximum perturbation angle range, (3) scanning through all the possible side chain conformations (from a predefined rotamer library) and picking the one that results in the lowest energy for the perturbed structure, and (4) accepting or rejecting the move based on the Metropolis acceptance criterion, given a specified temperature. A rotamer library is used to define a discrete set of conformations for each residue. Although the moves are restricted to local perturbations relative to a single residue, the perturbation selected influences the entire structure. Thus, relatively minor local changes can propagate down the chain resulting in large shifts in the conformation of the protein.

REMC^{GRID} Improves Structure Refinement

REMC^{GRID} was used to refine a set of 10 high-resolution (≤2.8 Å) crystal structures from the PDB (Table 1). We calculated the energy improvements attained and the compute times required to achieve them for REMC^{GRID} and for GRID alone. Given enough computational resources, REMC^{GRID} results in structures with significantly better energies than GRID alone (Fig. 2 and Fig. 3). As seen in a plot showing the ratio of computational resources required for REMC^{GRID} to match the level of energy improvement obtained by GRID (see Fig. 4), with longer compute times (number of moves) REMC^{GRID} achieves energy improvements that are up to ~15% better than GRID alone. Since REMC^{GRID} runs multiple trajectories in parallel and the compute time of each trajectory is comparable to that of GRID (GRID takes ~1-2 minutes, since it is deterministic and therefore just runs once), on a large cluster, REMC^{GRID} can get this 15% improvement with an overall compute time that is ~3-fold that of GRID; i.e., in ~2-6 minutes. Low score decoys generated for one of the 10 protein structures (pdbID 1pga) is shown in Fig. 7 and Fig. 8.

REMC^{GRID} Generates Uniform Distributions of Diversely Perturbed Decoys

We used REMC^{GRID} to generate a set of decoys based on 59 high-resolution (≤2.8 **Å**) crystal structures from the PDB. These 59 structures included proteins ranging in size from 50-150 residues and is the same dataset used by Das et al. for decoy generation with Rosetta [26]. We applied REMC^{GRID} with 25 different parameter sets specifying 5 different φ/ψ perturbation angles and 5 different acceptance probabilities. For each pair of parameters, we generated 320 decoys for each of the 59 proteins, for a total of 472,000 decoys. We found that smaller perturbations of φ and ψ result in smaller deviations in structure. Moreover, higher simulation temperatures caused the acceptance of more moves that worsen the energy of the structure, resulting in more diverse and more perturbed decoy sets. The parameter set with the largest φ/ψ perturbations and the least restricting kT_f values resulted in the most perturbed decoy structures. These decoys covered the broadest range of RMSD's (including 10+ Å) and showed uniform broadness across protein size. These characteristics (uniformity and broadness) are important for the proper performance of machine learning algorithms. We planned to use this decoy set to train a machine learning classifier (SVM), which would be used as a scoring function for structure prediction (see Chapter 3).

REMC^{GRID} generates more uniform decoys than Rosetta

In generating decoys, Rosetta starts from an unfolded polypeptide chain and after several rounds of full structure prediction, produces a set of low energy conformations [27]. In contrast, REMC^{GRID} starts from the crystal structure and systematically and gradually increases the energy of the structure to generate near-native, partially-folded, and unfolded

decoy structures. REMC^{GRID} uses an all-atom force field to ensure that structures with severe clashes are not generated.

We compared our most perturbed REMC^{GRID} decoys (those produced with the largest φ/ψ perturbations and the least restricting kT_f values) with those obtained using Rosetta for broadness (range of backbone RMSDs) and uniformity (distribution of RMSDs across different protein structures). The REMC^{GRID} decoys are very uniform: the median value is ~1 Å and is very close to 3 percentile value, and the most perturbed structures (98 percentile) rarely surpass 12 Å RMSD (Fig. 10). In contrast, the Rosetta decoys are quite varied: the median typically ranges from 5-10 Å, and the 98 percentile values are much higher, with some decoys reaching > 30 Å RMSD. REMC^{GRID} decoys show adequate broadness (0 to 10+ Å) and have distributions that are fairly consistent, whereas Rosetta decoys show very large differences in their distributions. Thus, REMC^{GRID} can generate a uniform distribution of perturbed structures covering a broad range of RMSDs that is independent of protein size and class.

Methods

REMC^{GRID} implementation and parameterization

The REMC^{GRID} algorithm was developed in C++ and incorporated into the phoenix protein design software package. Standard Message Passing Interface (MPI) was used for multiprocessor parallelization. The phoenix force field with default parameters was used to calculate the energy of the protein conformations. Side-chain conformations were selected from the Dunbrack backbone-dependent rotamer library [28].

REMC^{GRID} takes as input six main parameters: (1) the maximum perturbation angle (Δφ, Δψ), (2) number of parallel replicas, (3) number of Monte Carlo steps before

swapping configurations, (4) number of rounds of running Monte Carlo simulations, (5) lowest temperature replica, and (6) highest temperature replica. Values for maximum $\Delta\varphi$ and $\Delta\psi$ ranged from ± 2° to ± 10°; the number of parallel replicas was 80; the number of Monte Carlo steps before swapping configuration was 100; number of rounds of Monte Carlo was 20; lowest temperature was $kT_i = 0.5$; and highest temperature ranged from $kT_f = 1$ to $kT_f = 9$. The final set of parameters was obtained after trying $REMC^{GRID}$ with several different values for each parameter and computing the RMSDs of the resulting decoys.

Acknowledgments

This work was supported by the Defense Advanced Research Projects Agency (DARPA) and a Department of Defense National Security Science and Engineering Faculty Fellowship (S.L.M).

References

1. Chitsaz, M.; Mayo, S. L. J Comput Chem 2013, 34(6), 445-450.

2. Bernstein, F. C.; Koetzle, T. F.; Williams, G. J.; Meyer, E. F., Jr.; Brice, M. D.; Rodgers, J. R.; Kennard, O.; Shimanouchi, T.; Tasumi, M. Eur J Biochem 1977, 80(2), 319-324.

3. Dill, K. A.; MacCallum, J. L. Science 2012, 338(6110), 1042-1046.

4. Summa, C. M.; Levitt, M. Proc Natl Acad Sci U S A 2007, 104(9), 3177-3182.

5. Best, R. B.; Zhu, X.; Shim, J.; Lopes, P. E.; Mittal, J.; Feig, M.; Mackerell, A. D., Jr. J Chem Theory Comput 2012, 8(9), 3257-3273.

6. Ponder, J. W.; Case, D. A. Adv Protein Chem 2003, 66, 27-85.

7. Vanommeslaeghe, K.; Hatcher, E.; Acharya, C.; Kundu, S.; Zhong, S.; Shim, J.; Darian, E.; Guvench, O.; Lopes, P.; Vorobyov, I.; Mackerell, A. D., Jr. J Comput Chem 2010, 31(4), 671-690.

8. Hsieh, M. J.; Luo, R. J Phys Chem B 2010, 114(8), 2886-2893.

9. Marrink, S. J.; Risselada, H. J.; Yefimov, S.; Tieleman, D. P.; de Vries, A. H. J Phys Chem B 2007, 111(27), 7812-7824.

10. Periole, X.; Marrink, S. J. Methods Mol Biol 2013, 924, 533-565.

11. Yesylevskyy, S. O.; Schafer, L. V.; Sengupta, D.; Marrink, S. J. PLoS Comput Biol 2010, 6(6), e1000810.

12. Li, X.; Wu, X.; Wu, G. J Theor Biol 2014.

13. Berhanu, W. M.; Jiang, P.; Hansmann, U. H. Phys Rev E Stat Nonlin Soft Matter Phys 2013, 87(1), 014701.

14. Vendruscolo, M. J Biol Phys 2001, 27(2-3), 205-215.

15. Koga, N.; Takada, S. J Mol Biol 2001, 313(1), 171-180.

16. Daidone, I.; Amadei, A.; Roccatano, D.; Nola, A. D. Biophys J 2003, 85(5), 2865-2871.

17. Chen, J.; Im, W.; Brooks, C. L., 3rd. J Comput Chem 2005, 26(15), 1565-1578.

18. Frembgen-Kesner, T.; Elcock, A. H. J Mol Biol 2006, 359(1), 202-214.

19. Stumpff-Kane, A. W.; Maksimiak, K.; Lee, M. S.; Feig, M. Proteins 2008, 70(4), 1345-1356.

20. Jiang, P.; Yasar, F.; Hansmann, U. H. J Chem Theory Comput 2013, 9(8).

21. Yang, Y.; Liu, H. J Comput Chem 2006, 27(13), 1593-1602.

22. Zheng, Q.; Rosenfeld, R.; DeLisi, C.; Kyle, D. J. Protein Sci 1994, 3(3), 493-506.

23. Price, D. J.; Brooks, C. L., 3rd. J Comput Chem 2002, 23(11), 1045-1057.

24. Tozzini, V. Curr Opin Struct Biol 2005, 15(2), 144-150.

25. Haario, H.; Saksman, E.; Tamminen, J. Bernoulli 2001, 7(2), 223-242.

26. Das, R.; Qian, B.; Raman, S.; Vernon, R.; Thompson, J.; Bradley, P.; Khare, S.; Tyka, M. D.; Bhat, D.; Chivian, D.; Kim, D. E.; Sheffler, W. H.; Malmstrom, L.; Wollacott, A. M.; Wang, C.; Andre, I.; Baker, D. Proteins 2007, 69 Suppl 8, 118-128.

27. Das, R.; Baker, D. Annu Rev Biochem 2008, 77, 363-382.

28. Dunbrack, R. L., Jr.; Karplus, M. J Mol Biol 1993, 230(2), 543-574.

input: kT_i, kT_f, $replicas$, $MCsteps$, θ^{max}, $rounds$, $struct$

$kT \leftarrow \frac{kT_f - kT_i}{replicas}\text{rank}() + kT_i$;
for $round \leftarrow 1$ **to** $rounds$ **do**
  $E_{prev} \leftarrow \text{Eng}(struct)$;
  for $step \leftarrow 1$ **to** $MCsteps$ **do**
    $pos \leftarrow \text{rand}(1, numRes(struct) + 1)$;
    $\Delta\phi \leftarrow \text{rand}(-\theta^{max}, \theta^{max})$;
    $\Delta\psi \leftarrow \text{rand}(-\theta^{max}, \theta^{max})$;
    $\text{perturb}(struct, pos, \Delta\phi, \Delta\psi)$;
    $\text{ReAlignRots}(pos)$;
    $E_{min} \leftarrow +\infty$;
    for $r \leftarrow 1$ **to** $numRots(pos)$ **do**
      $\text{setRot}(struct, r)$;
      if $E_{min} > Eng(struct, pos)$ **then**
        $E_{min} \leftarrow \text{Eng}(struct)$;
      end
    end
    $E_{current} \leftarrow E_{min}$;
    $\Delta E \leftarrow E_{current} - E_{prev}$;
    $p \leftarrow \text{rand}(0, 1)$;
    if $p < e^{-\frac{\Delta E}{kT}}$ **then**
      $\text{Accept}(struct)$;
      $E_{prev} \leftarrow E_{current}$;
    else
      $\text{Reject}(struct)$;
    end
  end
  $E \leftarrow \text{Eng}(struct)$;
  $kT \leftarrow \text{SyncSwapReplicas}(\text{rank}(), E)$;
end

Fig 1. REMC[GRID] algorithm pseudocode; the algorithm takes a crystal structure and a parameter set as input and generates a set of low energy decoys as output. Level of perturbation depends on the parameter set.

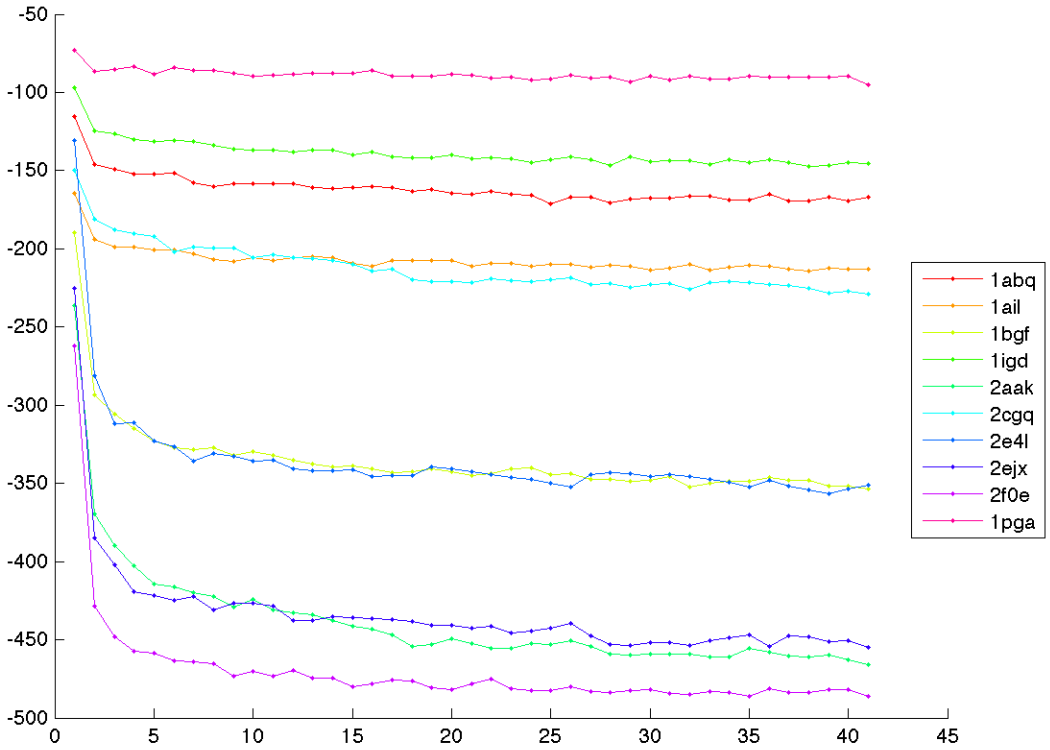


Fig 2. The energy improvement of 10 single-chain high-resolution crystal structures using REMC^{GRID} employing phoenix energy force field.

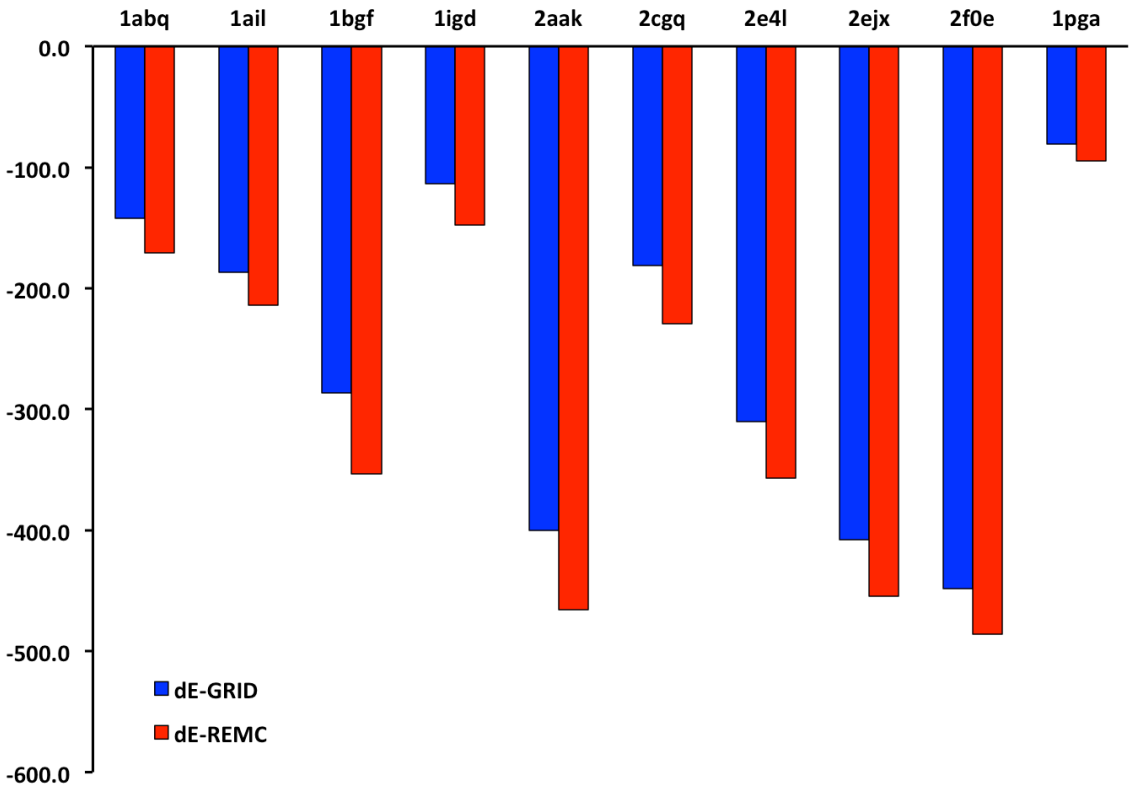


Fig 3. REMC^{GRID} energy improvement compared to GRID energy improvement; REMC^{GRID} results in significantly more energy improvements than GRID.

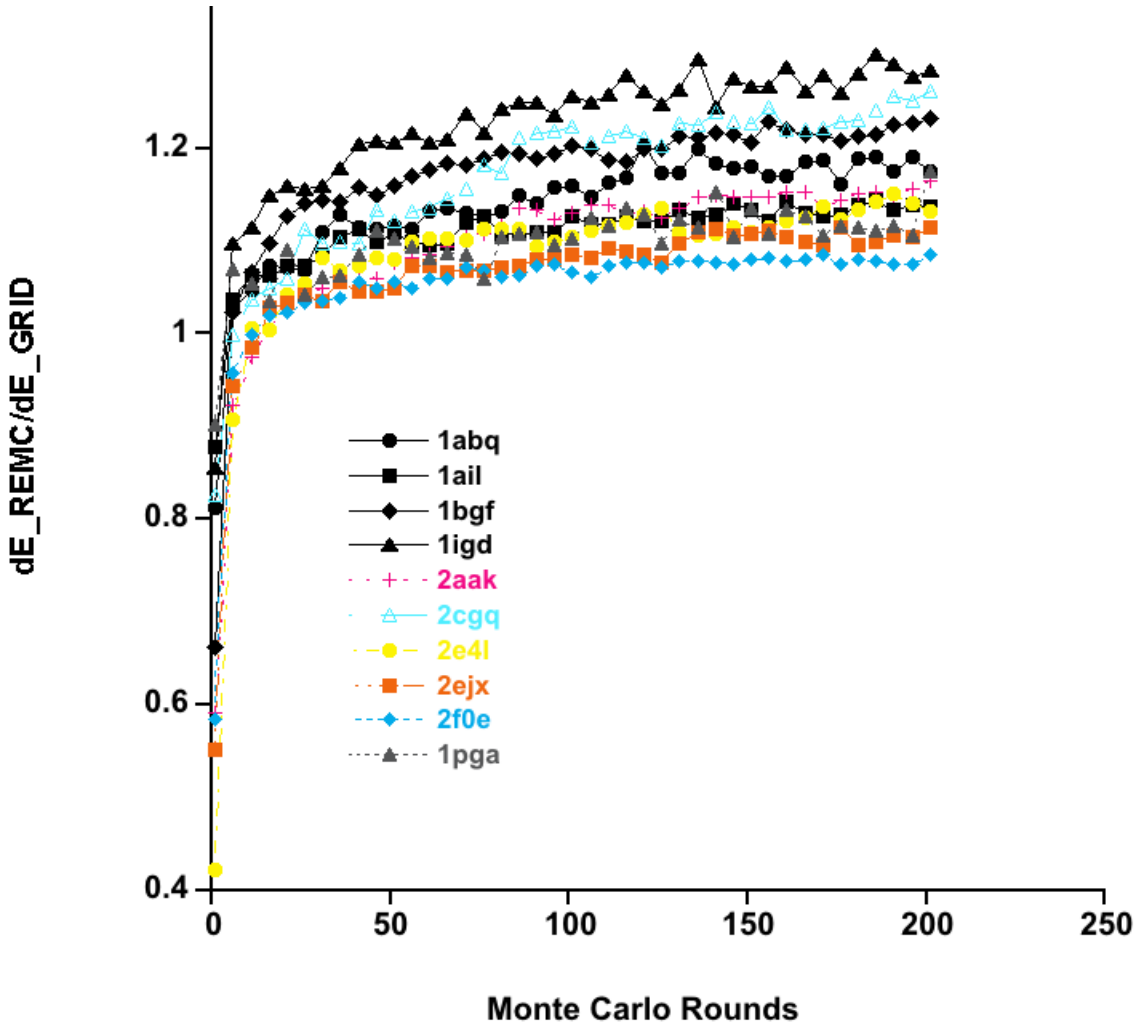


Fig 4. The ratio of energy improvement that REMC^{GRID} achieves compared to the energy improvement attained by GRID based on the number of steps of Monte Carlo simulation.

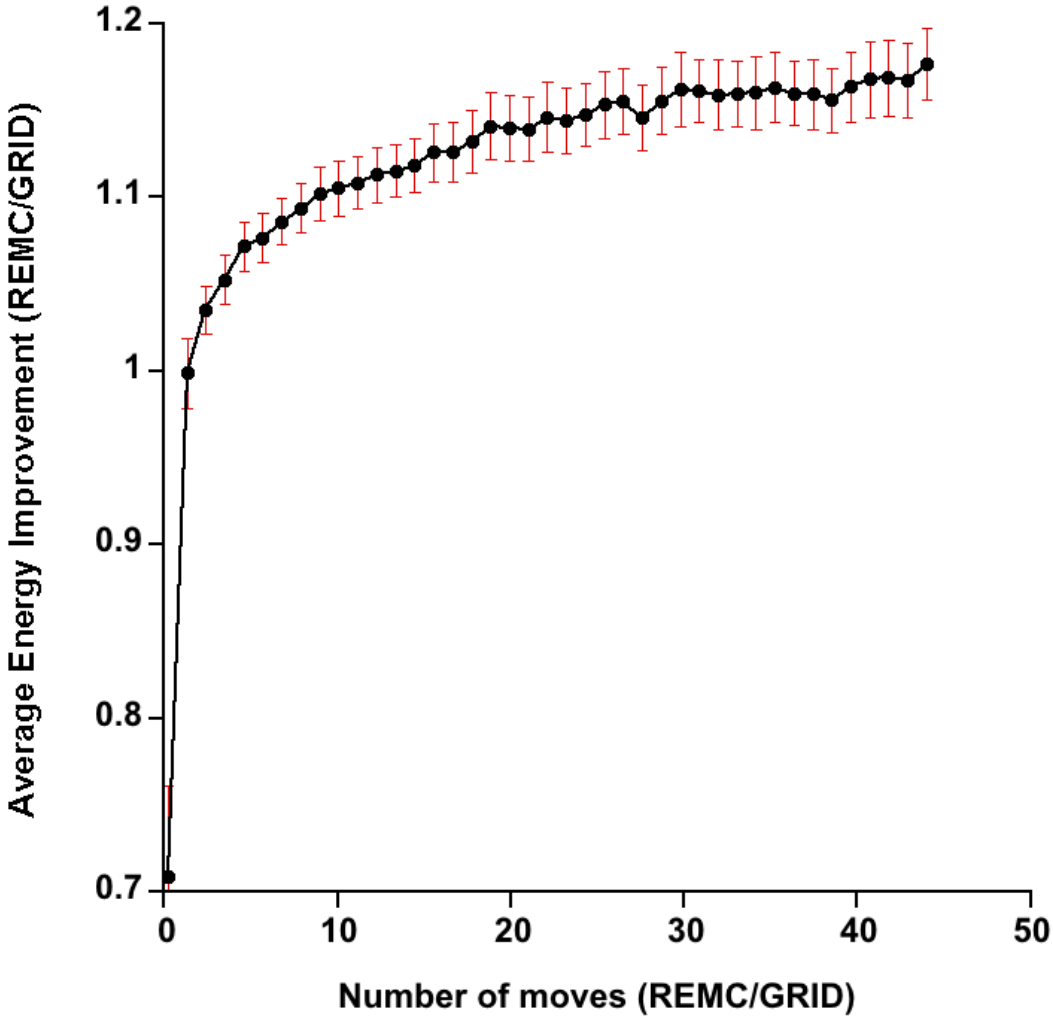


Fig 5. Computational effort comparison between REMC^{GRID} and GRID; REMC^{GRID} by spending more computational effort can achieve higher energy improvements. REMC^{GRID} continues to improve the energy improvement by being allowed to spend more computational resources.

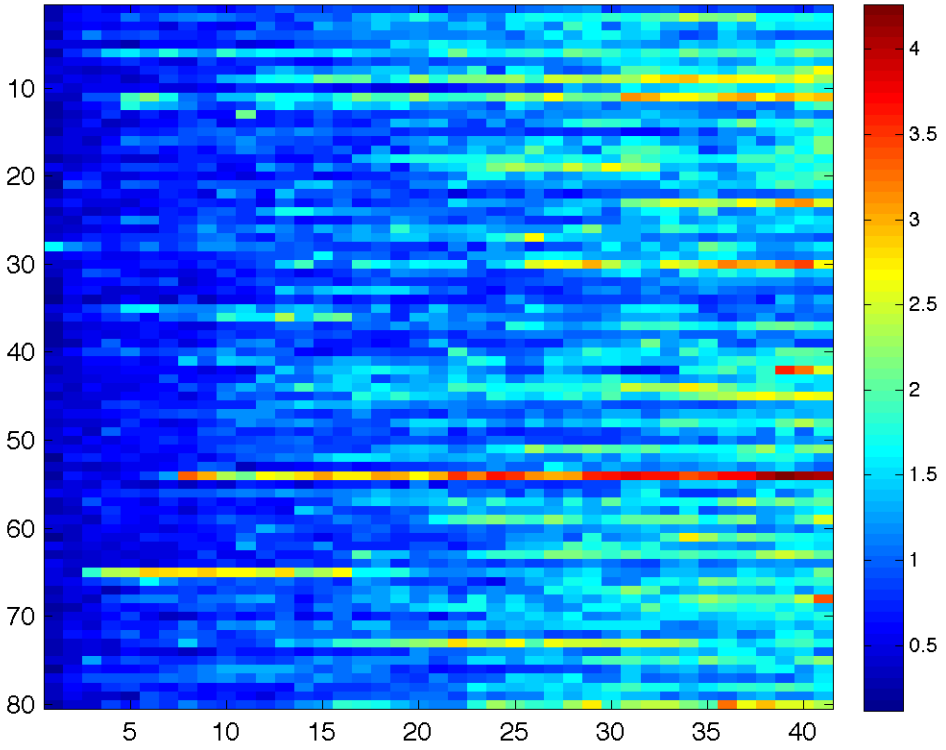


Fig 6. Backbone rmsds of 80 parallel replicas; 40 rounds of swaps between replicas is shown; protein G with pdbID "1pga" is the protein we used in this simulation.

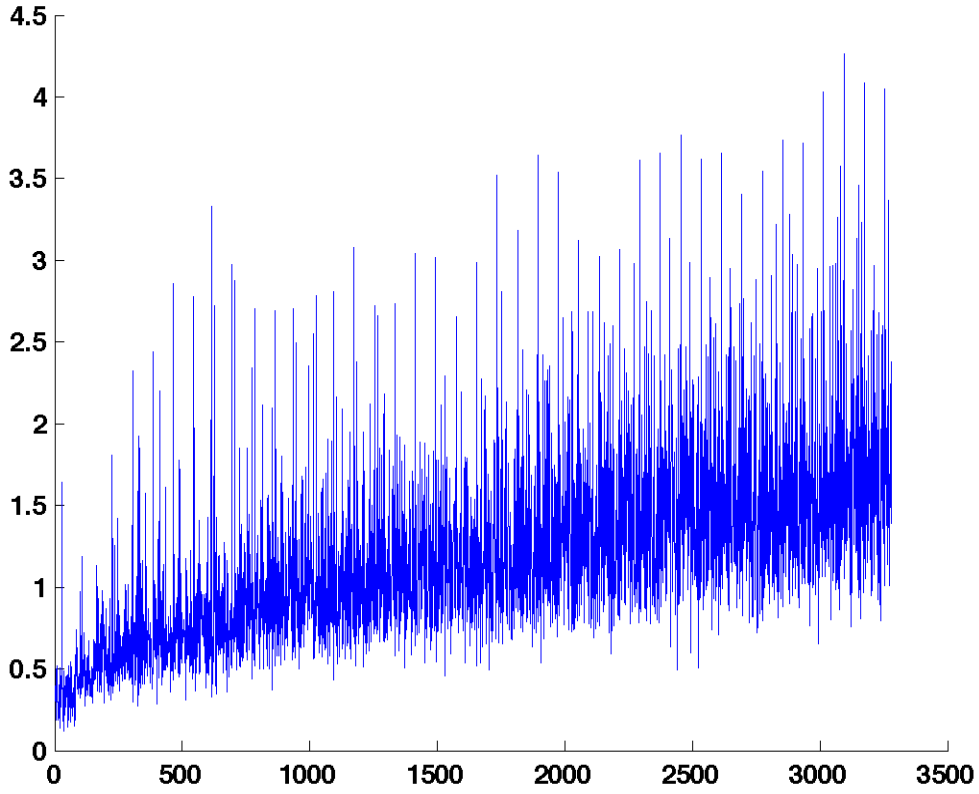


Fig 7. Backbone rmsds of decoys (protein G, 1pga); backbone rmsd increases as the number of iterations increases in this simulation. Parameters are set to generate a diverse set of decoys.

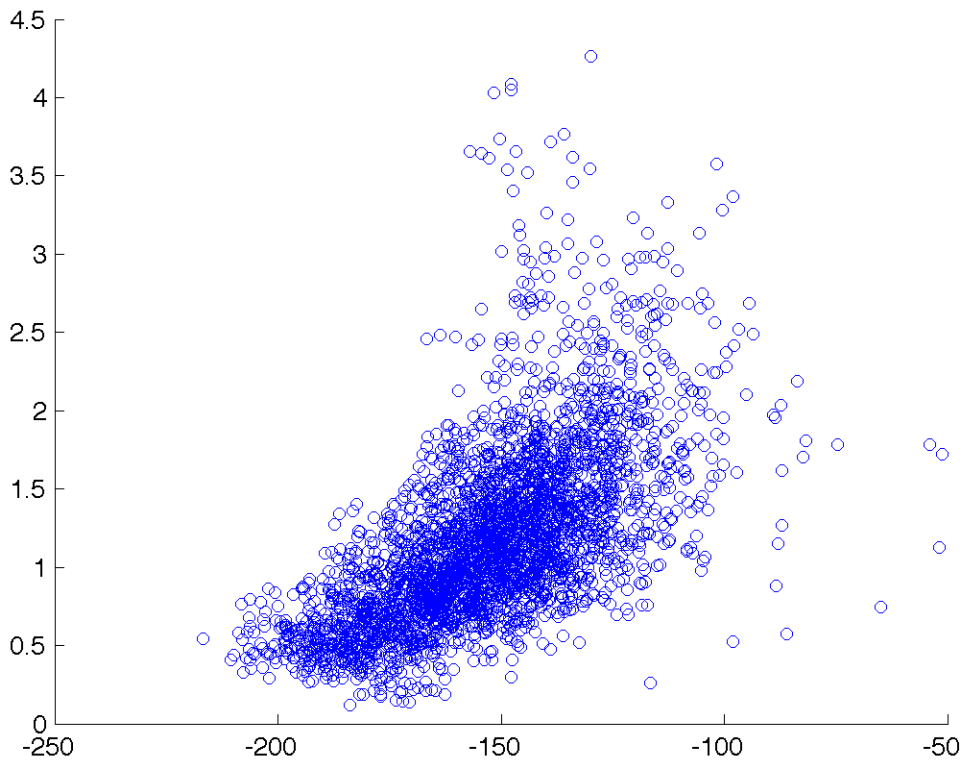


Fig 8. Backbone rmsd vs. energy of decoys generated; gradual increase in the energy of the structure results in a funnel like distribution of backbone RMSDs vs. energies (pdbID 1pga).

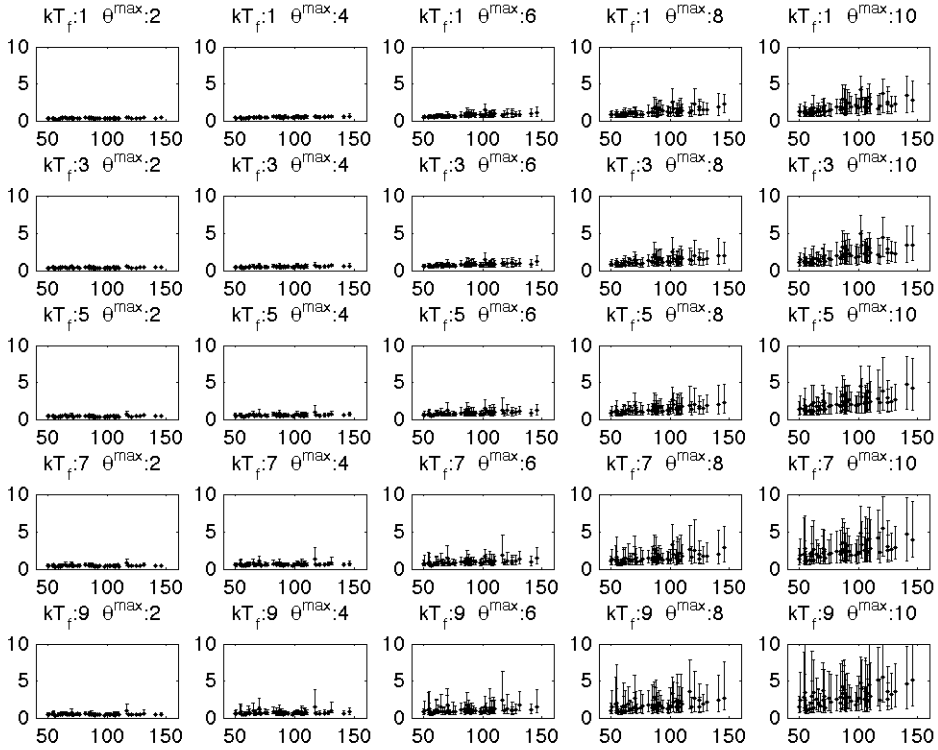


Fig 9. Backbone rmsd distribution of decoy sets generated by REMC^{GRID}. In this figure we have generated 25 different decoy sets by varying two parameters of the algorithm (kT_f and θ). Each decoy set is depicted in a box; the x-axis is the size of protein in terms of number of residues, and the y-axis is the backbone rmsd. Each line corresponds to one single protein structure and the decoys generated for that specific structure. The line is marked by three points of 25, 50, and 75 percentiles.

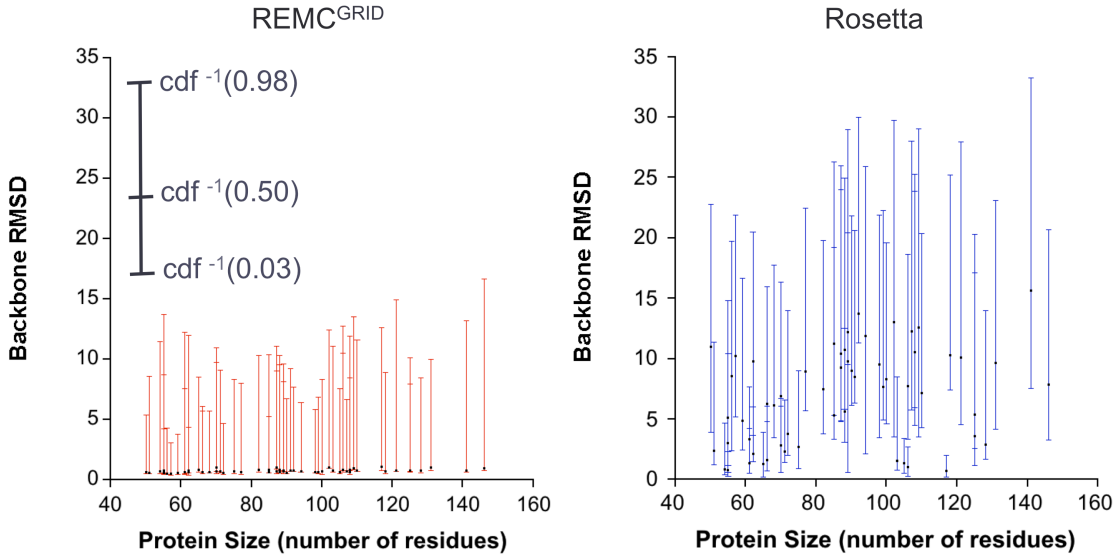


Fig 10. Comparison of Rosetta decoys and REMC^{GRID} decoys; REMC^{GRID} decoys are much more uniformly distributed due to gradual increase of the energy of the structures; Rosetta on the other hand produces set of decoys by performing a full structure prediction and picking the lowest energy decoys. Each line is marked at three points, the lowest point corresponds to 3 percentile, the middle mark corresponds to 50 percentile and the top most mark corresponds to 98 percentile.

PDB ID	No. of residues	Resolution (Å)	Class	% α	% β
1abq	56	2.80	β	4	43
1pga	56	2.07	α+β	25	42
1igd	61	1.10	α+β	22	42
1ail	70	1.90	α	80	0
2cgq	74	1.83	α	38	1
1bgf	124	1.45	α	79	1
2ejx	134	1.79	α+β	35	41
2f0e	135	1.95	α+β	24	28
2aak	150	2.40	α+β	35	19
2e4l	155	2.00	α/β	34	29

[a]Classifications based on Dictionary of Secondary Structure of Proteins (DSSP)

Table 1. Ten high-resolution crystal structures that were used in testing REMC^{GRID} refinement.

Chapter 3

A Machine Learning Approach to Improving Structure Prediction by Incorporating Coarse-Grained Features

Abstract

Although considerable progress has been made in the last few decades, the ability to predict the three-dimensional structure of a protein from its amino acid sequence remains a major challenge in structural biology. A successful energy-based protein structure prediction algorithm must address two problems: (1) inaccuracies in the force field, and (2) inefficient sampling of conformational space. The all-atom force fields currently available are too sensitive—even very minor perturbations in the coordinates of a protein can result in enormous energy fluctuations. This results in a very rugged energy landscape that is difficult to explore. One approach to solving this problem is to incorporate coarse-grained features into the force field. Here, we develop a coarse-grained model and use it to calculate a coarse-grained (CG)-score for a set of decoy structures. Three additional features were also calculated: surface area, volume, and all-atom energy. Decoys were classified into four groups based on their backbone RMSD relative to the crystal structure. A support vector machine (SVM) classifier was then trained to predict the class of each decoy in three different training experiments. The first two used either all-atom energy or CG-score as the sole feature for training and the third combined all four features (all-atom energy, CG-score, surface area, and volume). The CG-score alone produced better results than the all-atom energy alone, but the combination of all four features significantly improved the ability of the SVM to correctly classify the decoys.

Introduction

Efforts aimed at predicting the structure of a protein from its amino acid sequence have been ongoing for over 50 years, and although much progress has been made, the accurate and reliable prediction of protein structures still presents many challenges [1]. For an energy-based structure prediction algorithm to be successful, it must deal with two problems: (1) inaccuracies in the force field, and (2) inefficient sampling of the conformational space [2]. Force field development has been ongoing for many years, and several research groups are attempting to develop simple yet relevant models that describe the physics of protein folding [3 4 5 6 7 8]. The simplicity of the models is crucial as compute power is limited and a simpler model can reduce the calculation time required to adequately sample the possible conformations [7].

One of the problems with current force fields is that they are too sensitive; that is, small perturbations in the configuration of a structure can result in large differences in its energy [9 5]. This results in a very rugged energy landscape, which makes it difficult for sampling algorithms to search conformational space to find the lowest energy structure without being trapped in local minima. A smoother energy landscape is therefore desirable and would in fact mimic that of natural proteins, whose folding rate suggests a smooth landscape [10]. Coarse-grained models aim to alleviate the sensitivity of current force fields [11 7 12]. Here, we describe a simple coarse-grained model that was devised for use in training a machine learning classifier. This coarse-grained model was used to calculate a CG-score for a set of decoy structures. Similarly, we computed three other features for the decoys: surface area, volume, and all-atom energy. We classified the decoys into four

groups based on backbone rmsd from the crystal structure. We then trained a support vector machine (SVM) classifier on these features to predict the class of each decoy. Three SVM training experiments were conducted: (1) with all-atom energy as the sole feature, (2) with CG-score as the sole feature, and (3) with all four features used simultaneously (CG-score, all-atom energy, surface area, and volume). We found that the combination of all four features was more predictive than either all-atom energy or CG-score alone.

Results and Discussion

A simple coarse-grained model

To overcome the sensitivity of an all-atom force field, we devised a coarse-grained model for use in training a machine-learning scheme. Our coarse-grained model uses a very simple representation of protein structure, simplifies the calculation of inter-residue interactions by grouping amino acid types, and calculates derivative energies explicitly, all of which speed up the sampling of conformational space. Unlike other coarse-grained models, it also favors stabilizing long-range interactions, which are typically not taken into account.

The backbone of the protein structure is modeled by connecting the C-alpha (CA) atoms with straight lines, and side chains are modeled as spheres connected to the corresponding CA atoms on the backbone (see Fig. 1). The center of mass of the side chain is considered to be the center of the sphere. For each residue type, we calculated the average sphere radius by processing more than 3,500 high-resolution crystal structures extracted from the Protein Data Bank [13]. Similarly, we calculated the average length of the

bond (straight line) connecting two consecutive CA atoms on the backbone. This model can be used to score any three-dimensional configuration of a protein structure. The CG-score for a given protein structure is computed by summing all the pairwise energies of all the residues. The pairwise energy of two residues is the sum of three energies: (a) the backbone-backbone energy (E_{bb}), (b) the sidechain-backbone energy (E_{sb}), and (c) the sidechain-sidechain energy (E_{ss}). The E_{bb} for two residues is based on the distance between their CA atoms, with a penalty applied if the distance is less than 4.2 Å (Fig. 2A). The E_{sb} is similarly based on the distance between the center of the sphere of the first residue and the CA atom of the other residue. A clash between the sphere and the backbone results in a penalty (Fig. 2B). E_{ss} is based on the distance between the centers of mass of the two residues, their radii, and which of 7 amino acid groups they belong to (categorized by size and biophysical properties) (see Table 1 and Fig. 2C). Basically, if the two residues are clashing, a penalty is applied; if they are far away from each other, there is no penalty; and if they are close but not clashing, a score is computed that depends on their amino acid groups (see Table 2 and Fig. 3). Note that E_{ss} is scaled up by sqrt(i-j) factor, to favor long-range stabilizing contacts. Long-range interactions that are not stabilizing are not favored (this factor is simply +1).

Generation of Decoy Structures

As described in Chapter 2, to generate decoys, we chose 59 high-resolution single chain crystal structures from the PDB, ranging in size from 50 to 150 residues. These same crystal structures were used to generate decoys in a study of Rosetta@home in full structure prediction[14]. We used REMC^{GRID} algorithm with 25 different parameter sets; for each parameter set, we generated 320 decoys for each of the 59 proteins by randomly choos-

ing a residue, perturbing its phi/psi angles, and applying the best rotamer for the perturbed configuration (see Fig. 4). We computed the backbone rmsd of each decoy with respect to its crystal structure. The parameter set with the largest phi/psi perturbations and the least restricting kT_f values resulted in the most perturbed decoy structures. We chose these decoys as our dataset because they covered the broadest range of rmsd's (including 10+ Å) and showed uniform broadness across protein size. These characteristics (uniformity and broadness) are important for the proper performance of machine learning algorithms.

Classification of Decoys and Features Used in SVM Training

The decoys were then grouped into four classes based on backbone rmsd from the crystal structure, with classes A, B, C, and D corresponding to rmsd's of 0-1, 1-3, 3-10, and 10+ Å. For every decoy, we computed four features: (i) all-atom energy, (ii) volume, (iii) surface area, and (iv) CG-score as described above.

Support Vector Machine Training

We trained our support vector machine classifier to predict the class of each decoy using three different feature sets: (1) with all-atom energy as the sole feature, (2) with CG-score as the sole feature, and (3) with all four features used simultaneously (CG-score, all-atom energy, surface area, and volume). Table 3 and Fig. 5 show the distribution of correct predictions for the four different classes. Fig. 5A shows the distribution of classified decoys that are 0-1 Å backbone rmsd away from their native state. The "all-4" classifier predicts about 80% of the decoys correctly, whereas the classifier trained using all-atom en-

ergy alone predicts only about 25% of the decoys correctly. This trend is consistent in all four different classes of A, B, C, and D.

Methods

Coarse-grained parameterization

In the coarse-grained model, the CG-score of a protein structure is calculated by summing the pairwise energies of all the residues. The value of the sidechain-sidechain term, Ess, depends on the amino acid groups of the two residues (see Fig. 3). Five levels of energy were used to represent the different pairwise interactions: -1, -0.5, 0, +0.5, and +1. A positive score corresponds to an unfavorable interaction and a negative score corresponds to a favorable interaction. The five energy levels were chosen based on the biophysical characteristics of protein molecules.

Computing the features of the decoys

We computed the volume and surface area of the decoys using vossvolvox package [15]. The coarse-grained score was computed as described in the Results, and the all-atom energies were computed using the PHOENIX software package developed previously in the Mayo lab.

Support vector machine training

We used libsvm-3.17 open source library[16] as our support vector machine-training scheme. All input and output data were scaled to -1.0 to +1.0. We trained the classifier as a multi-class classification (C-SVC) with linear basis function kernel. We used 6-fold cross validation, and on average, generalization was around 80%.

References

1. Dill, K. A.; MacCallum, J. L. Science 2012, 338(6110), 1042-1046.

2. Summa, C. M.; Levitt, M. Proc Natl Acad Sci U S A 2007, 104(9), 3177-3182.

3. Vanommeslaeghe, K.; Hatcher, E.; Acharya, C.; Kundu, S.; Zhong, S.; Shim, J.; Darian, E.; Guvench, O.; Lopes, P.; Vorobyov, I.; Mackerell, A. D., Jr. J Comput Chem 2010, 31(4), 671-690.

4. Best, R. B.; Zhu, X.; Shim, J.; Lopes, P. E.; Mittal, J.; Feig, M.; Mackerell, A. D., Jr. J Chem Theory Comput 2012, 8(9), 3257-3273.

5. Price, D. J.; Brooks, C. L., 3rd. J Comput Chem 2002, 23(11), 1045-1057.

6. Hsieh, M. J.; Luo, R. J Phys Chem B 2010, 114(8), 2886-2893.

7. Marrink, S. J.; Risselada, H. J.; Yefimov, S.; Tieleman, D. P.; de Vries, A. H. J Phys Chem B 2007, 111(27), 7812-7824.

8. Vendruscolo, M. J Biol Phys 2001, 27(2-3), 205-215.

9. Ponder, J. W.; Case, D. A. Adv Protein Chem 2003, 66, 27-85.

10. Jewett, A. I.; Pande, V. S.; Plaxco, K. W. J Mol Biol 2003, 326(1), 247-253.

11. Tozzini, V. Curr Opin Struct Biol 2005, 15(2), 144-150.

12. Berhanu, W. M.; Jiang, P.; Hansmann, U. H. Phys Rev E Stat Nonlin Soft Matter Phys 2013, 87(1), 014701.

13. Bernstein, F. C.; Koetzle, T. F.; Williams, G. J.; Meyer, E. F., Jr.; Brice, M. D.; Rodgers, J. R.; Kennard, O.; Shimanouchi, T.; Tasumi, M. J Mol Biol 1977, 112(3), 535-542.

14. Das, R.; Qian, B.; Raman, S.; Vernon, R.; Thompson, J.; Bradley, P.; Khare, S.; Tyka, M. D.; Bhat, D.; Chivian, D.; Kim, D. E.; Sheffler, W. H.; Malmstrom, L.; Wollacott, A. M.; Wang, C.; Andre, I.; Baker, D. Proteins 2007, 69 Suppl 8, 118-128.

15. Voss, N. R.; Gerstein, M.; Steitz, T. A.; Moore, P. B. J Mol Biol 2006, 360(4), 893-906.

16. Chang, C. C.; Lin, C. J. Acm T Intel Syst Tec 2011, 2(3).

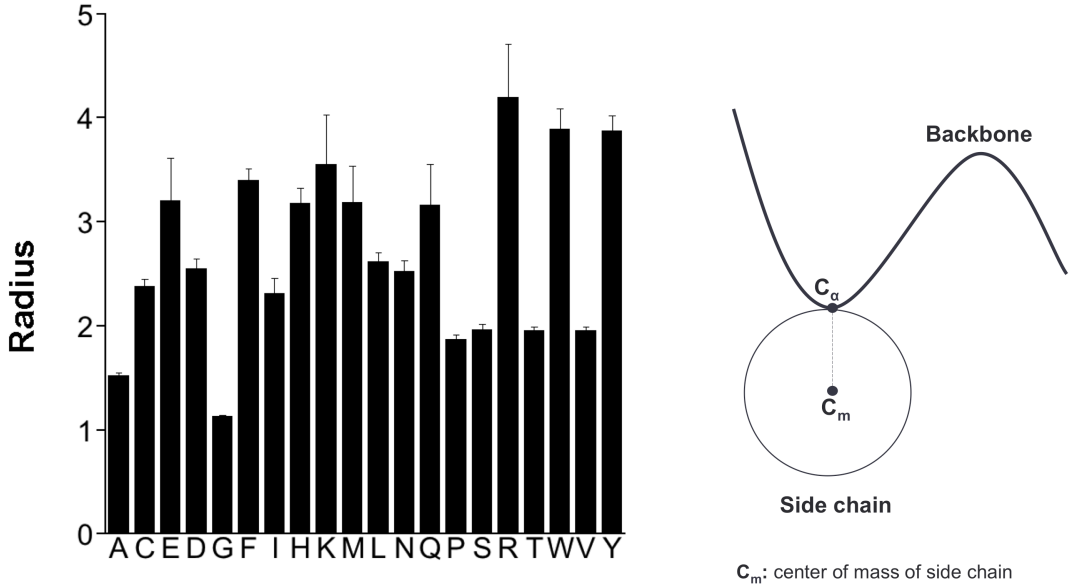


Fig 1. Coarse-grained model of protein structures; backbone is modeled as sticks connecting C-alpha (CA) atoms and sidechain is modeled as a sphere connected to the backbone. The average radius of each amino acid was extracted from 3,507 crystal structures from the PDB. Similarly, the average length of sticks connecting CA atoms was extracted from the same set of PDBs.

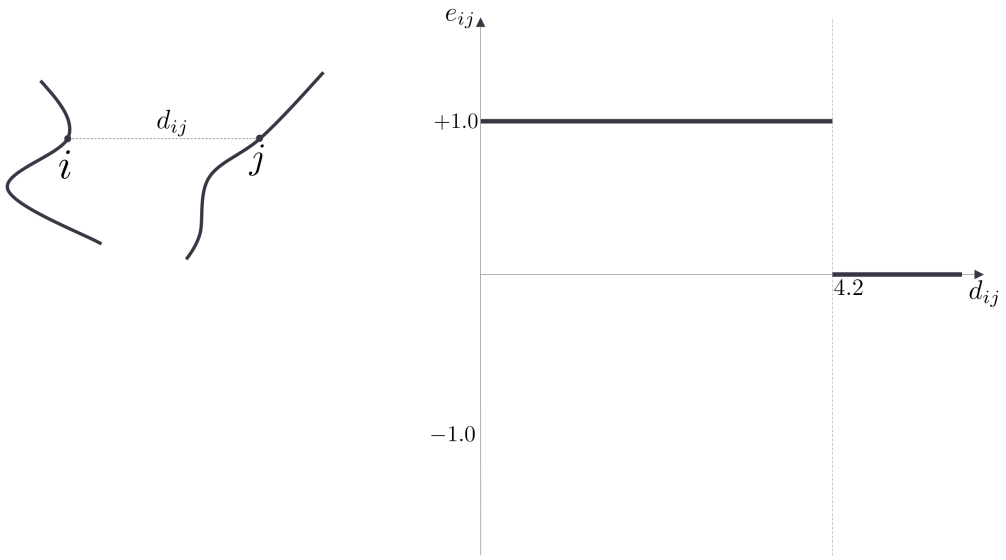


Fig 2. (A) Backbone-backbone energy, E_{bb}; If the distance between the two CAs of residue *i* and residue j (dij) < 4.2 Å, a penalty of +1 is applied; at greater distances, no penalty is applied.

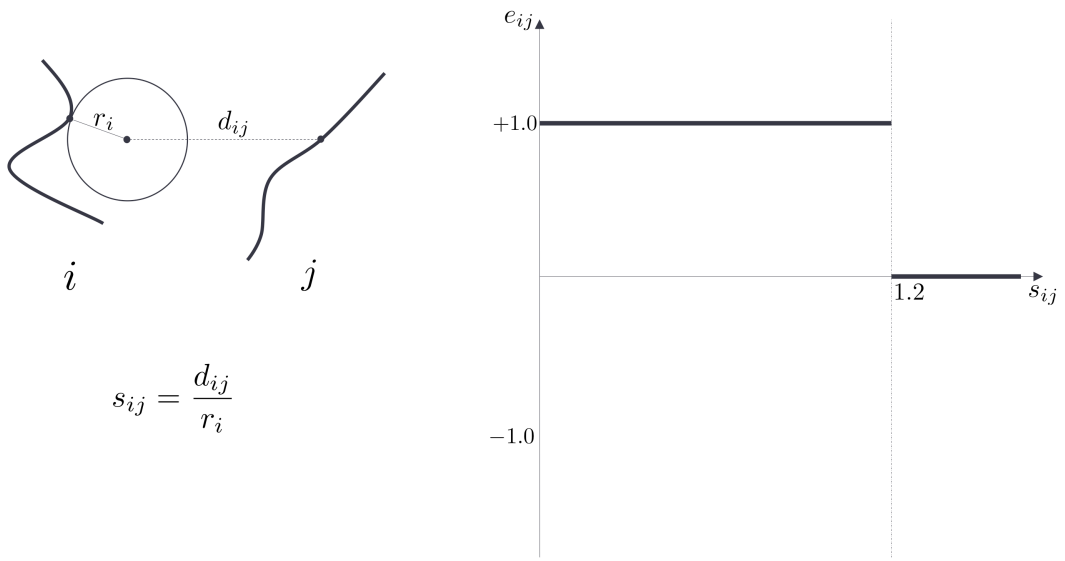


Fig 2. (B) Sidechain backbone energy term; distance between the center of mass of the sidechain of residue *i* and the backbone of residue *j* is denoted by d_{ij}. The effective closeness is computed by dividing the distance by radius of residue *i.* Based on closeness, there is either a penalty term or no energy contribution. If $s_{ij} \leq 1.2$ there is a penalty of +1; if $s_{ij} > 1.2$ there is no penalty.

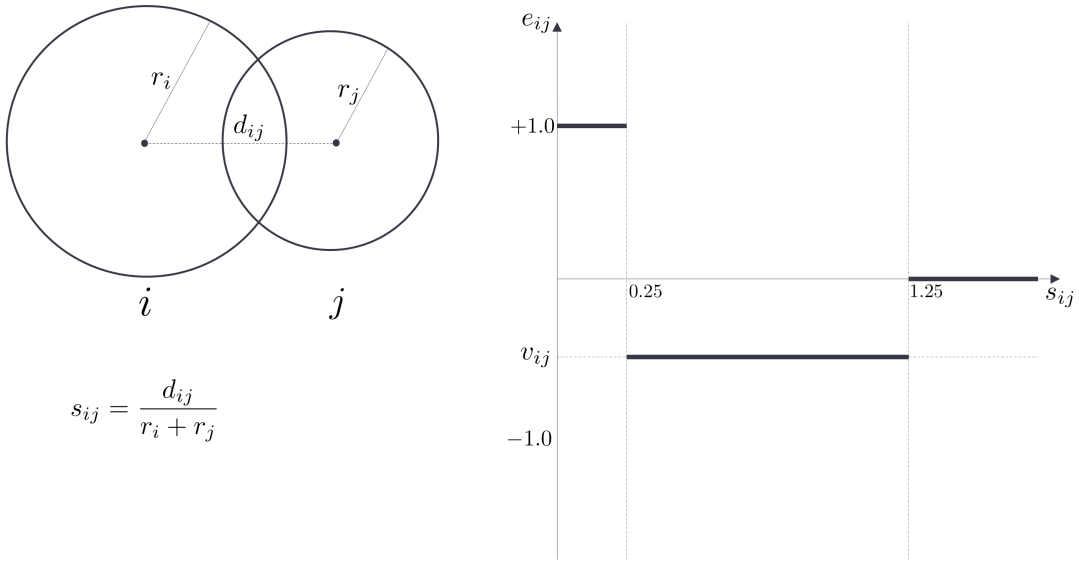


Fig 2. (C) Sidechain sidechain energy term; residue *i* and residue *j* are modeled as spheres. The distance between two centers of mass of the two sidechains is referred to as d_{ij}. Effective closeness is computed by dividing d_{ij} by the sum of two radii denoted by s_{ij}. If $s_{ij} \geq 1.25$ there is no interaction, thus no penalty or stabilizing contribution; if $s_{ij} \leq 0.25$ the two sidechains are too close and it's considered a clash and there is a penalty of +1; if $0.25 < s_{ij} < 1.25$ the two side-chains are in contact and the corresponding stabilizing or penalty term is determined based on the amino acid groups.

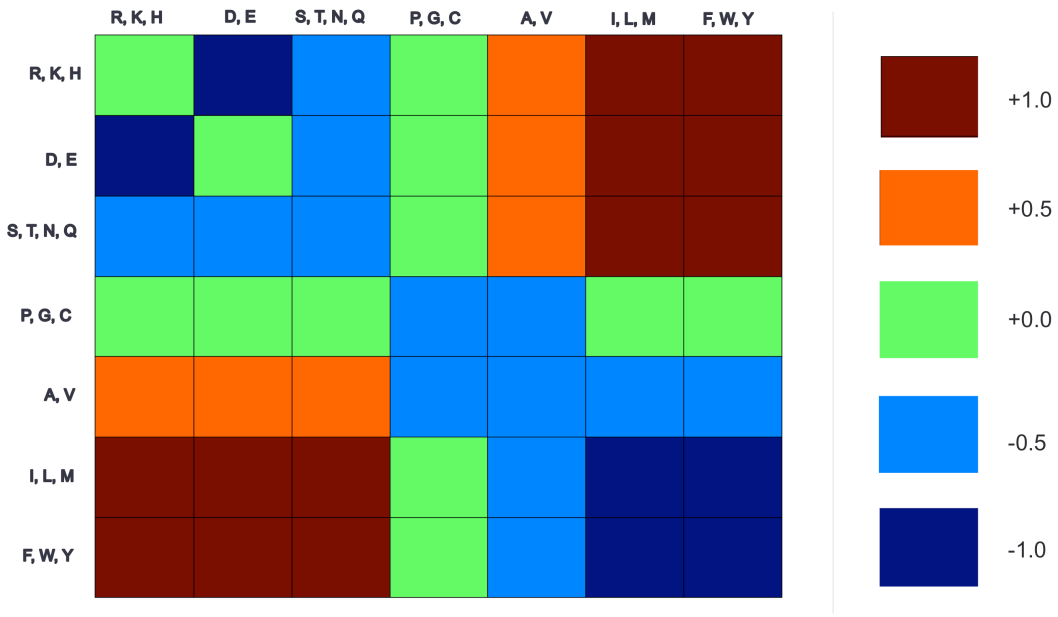


Fig 3. Pairwise energies among amino acid groups; amino acids are categorized into 7 different groups based on their biophysical properties and also their sizes. There are five energy levels: -1, -0.5, 0.0, +0.5 and +1 for simplicity and generalization. Negative energies are stabilizing and positive energies are not stabilizing. Numbers assigned to pairwise energies are derived based on biophysical properties.

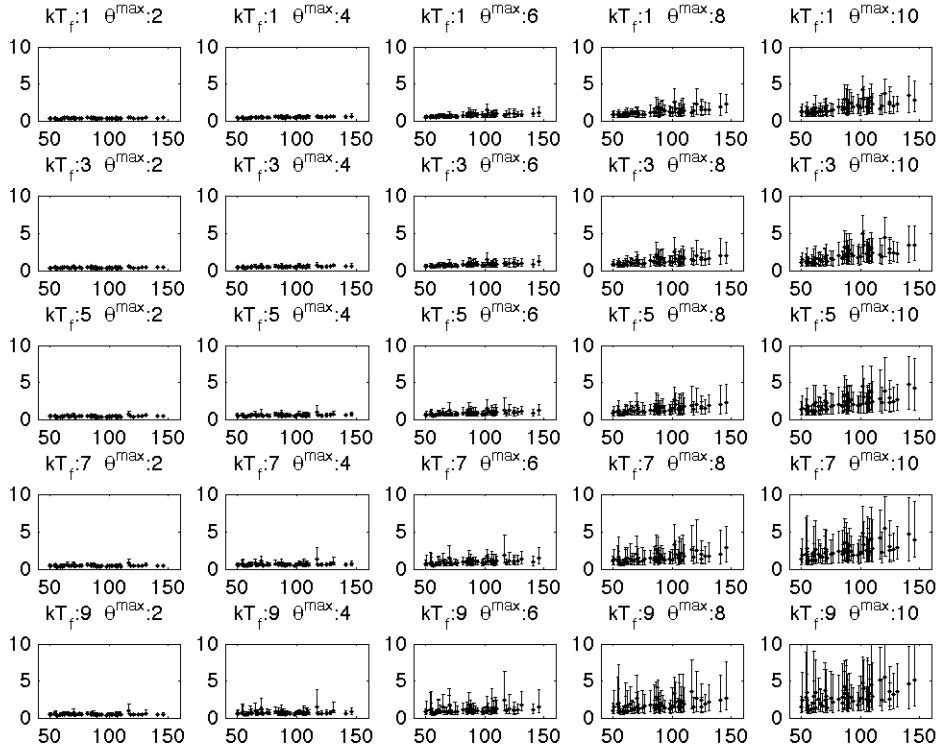


Fig 4. Backbone rmsd distribution of decoy sets generated by REMC^{GRID}. In this figure, we have generated 25 different decoy sets by varying two parameters of the algorithm (kT_f and θ). Each decoy set is depicted in a box; the x-axis is the size of protein in terms of number of residues, and the y-axis is the backbone rmsd. Each line corresponds to one single protein structure and the decoys generated for that specific structure. The line is marked by three points of 25, 50, and 75 percentiles.

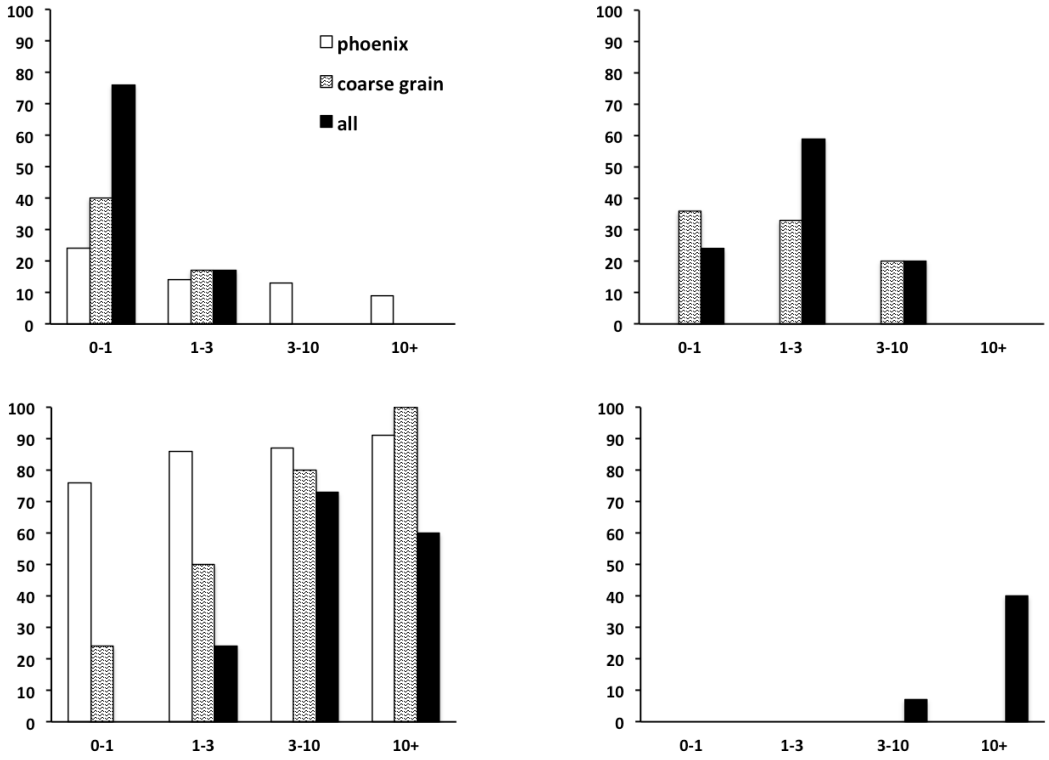


Fig 5. Comparison of three different classifiers; each classifier is trained using a different set of features, correspondingly they are trained using phoenix energy, coarse grain score, and all features composed of volume, surface area, coarse grain score and phoenix energy. (A) it shows how each of the three schemes classify decoys that are 0-1 Å backbone rmsd away from their native states; (B) it shows how decoys that are 1-3 Å backbone rmsd away from their native state are ranked; (C) it shows ranking of decoys that are between 3-10 Å rmsd away from their native state; (D) it shows ranking of decoys that were 10+ Å rmsd from their native state.

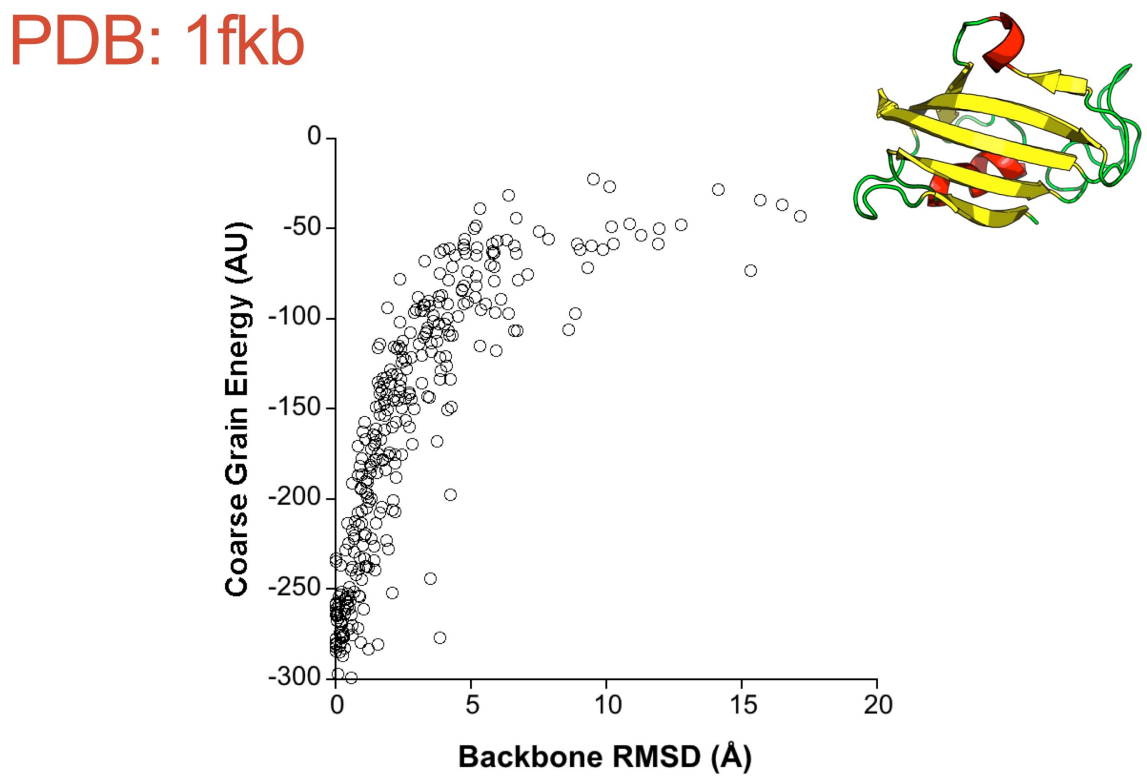


Fig S1. Coarse-grained score vs. backbone RMSD on 320 decoys generated from protein structure with pdbID 1fkb.

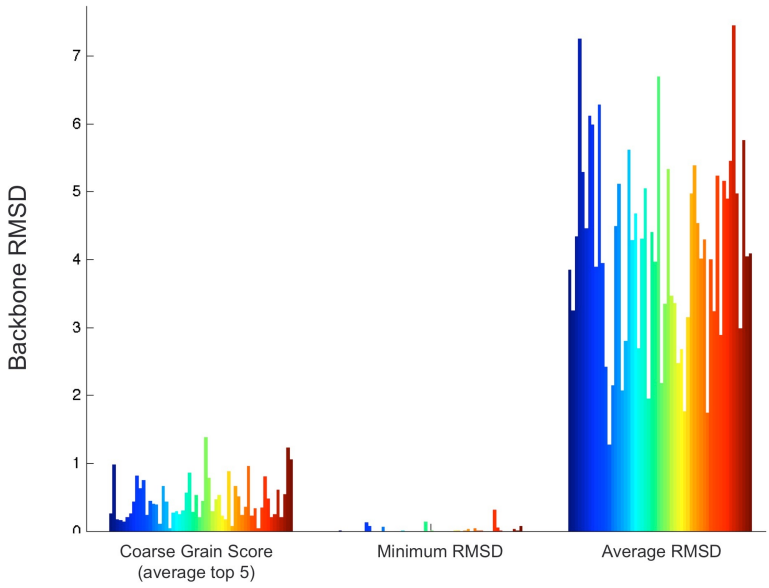


Fig S2. Comparison of coarse-grained score of structures with average RMSD and minimum RMSD in the pool of decoys, composed of 320 decoys for each of 59 protein structures; Average RMSD of top 5 structures according to coarse-grained scoring function is depicted in the first column; color coding corresponds to different proteins, ranked by number of residues.

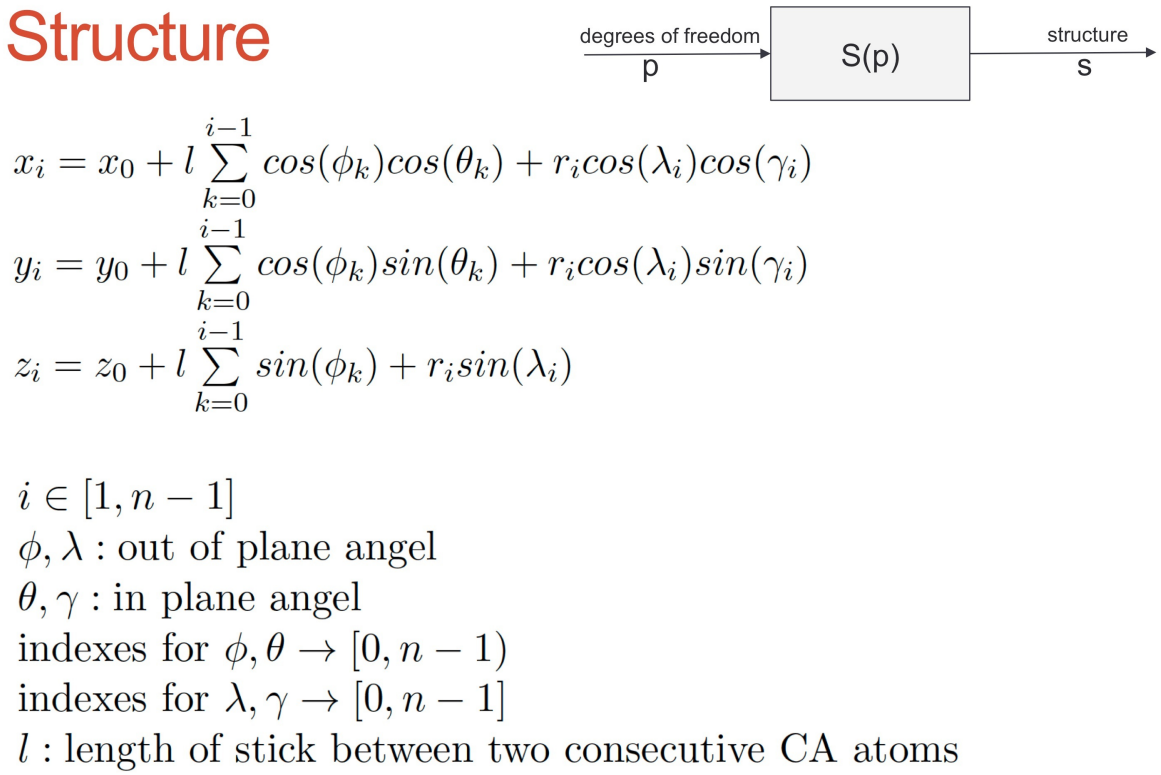
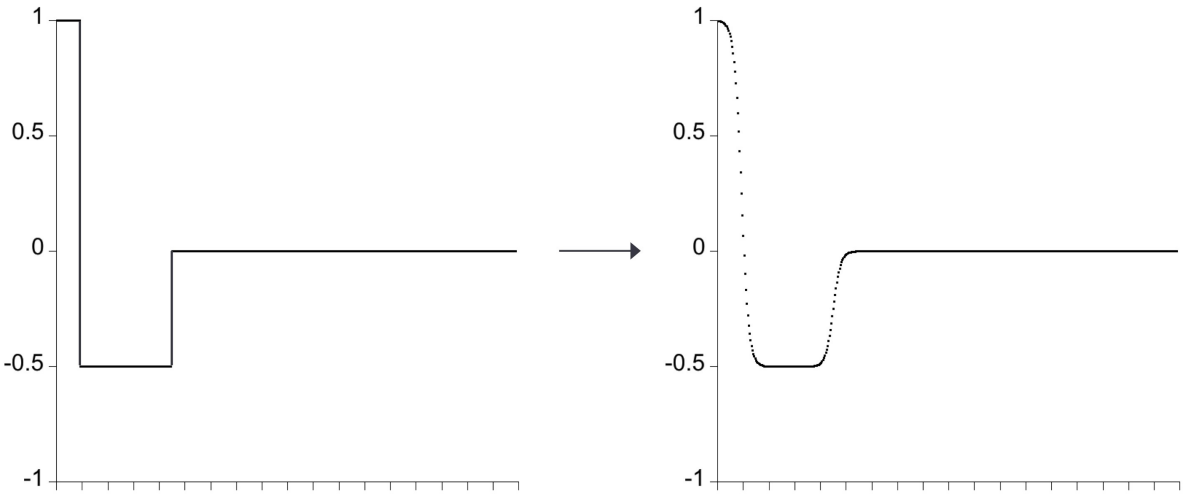
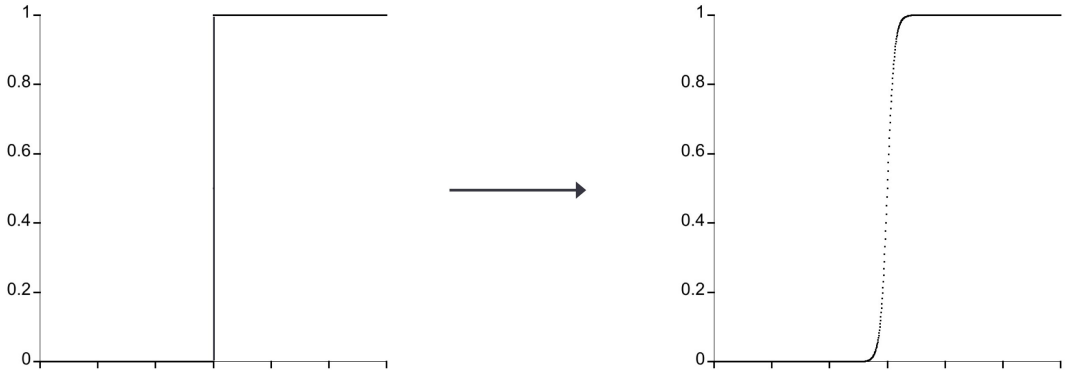


Fig S3. Protein structure can be described in p degrees of freedom in polar space; the above mentioned formula turns the structure from polar space into Cartesian space.

Smoothed Pairwise Energy



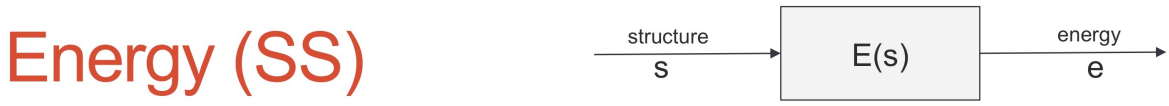
Sigmoid



$$y = step(x - a)$$

$$y = \theta(k(x - a)) = \frac{1}{1 + exp(-k(x - a))}$$

Fig S4. Modeling the scoring function using soft sigmoid functions; major benefit of sigmoid functions over step functions is that their derivative exists and easily can be calculated, whereas in the case of step functions, the derivative is an impulse function which is not easy to work with.



$$E = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} e_{ij} \cdot \text{coeff}(e_{ij})$$

$$\text{coeff}(e_{ij}) = \left(\sqrt{i-j} - 1\right) \cdot \theta(-k \cdot e_{ij}) + 1; \text{ where } k = 25$$

$$e_{ij} = 1 - (1 - v_{ij})\theta(k(s_{ij} - a)) - v_{ij} \cdot \theta(k(s_{ij} - b)); \text{ where } k = 25, a = 0.0625, b = 1.5625$$

$$s_{ij} = \frac{1}{(r_i + r_j)^2}\left[\Delta x_{ij}^2 + \Delta y_{ij}^2 + \Delta z_{ij}^2\right]$$

where:

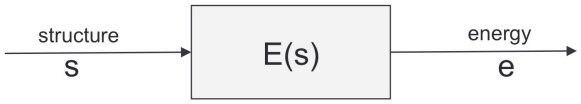
$$\Delta x_{ij} = \left(l \sum_{k=j}^{i-1} cos(\phi_k)cos(\theta_k) + r_i cos(\lambda_i)cos(\gamma_i) - r_j cos(\lambda_j)cos(\gamma_j)\right)$$

$$\Delta y_{ij} = \left(l \sum_{k=j}^{i-1} cos(\phi_k)sin(\theta_k) + r_i cos(\lambda_i)sin(\gamma_i) - r_j cos(\lambda_j)sin(\gamma_j)\right)$$

$$\Delta z_{ij} = \left(l \sum_{k=j}^{i-1} sin(\phi_k) + r_i sin(\lambda_i) - r_j sin(\lambda_j)\right)$$

Fig S5. Sidechain-sidechain energy term calculated based on Cartesian coordinates in the structure.

Energy (BB)



$$\Delta x_{ij}^{bb} = l \sum_{k=j}^{i-1} cos(\phi_k) cos(\theta_k)$$
$$\Delta y_{ij}^{bb} = l \sum_{k=j}^{i-1} cos(\phi_k) sin(\theta_k)$$
$$\Delta z_{ij}^{bb} = l \sum_{k=j}^{i-1} sin(\phi_k)$$

$$d_{ij}^{bb^2} = \Delta x_{ij}^{bb^2} + \Delta y_{ij}^{bb^2} + \Delta z_{ij}^{bb^2}$$
$$E^{bb} = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} e_{ij}^{bb}$$
$$e_{ij}^{bb} = \theta(-k^{bb}(d_{ij}^{bb^2} - a^{bb^2})); \text{ where } a^{bb} = 3.25 \text{ and } k^{bb} = 3$$

Fig S6. Backbone-backbone energy term calculated based on Cartesian coordinates in the structure.

Energy (SB)

for $i > j$ and

$$\Delta x_{ij}^{sb} = l \sum_{k=j}^{i-1} cos(\phi_k)cos(\theta_k) + r_i cos(\lambda_i)cos(\gamma_i)$$
$$\Delta y_{ij}^{sb} = l \sum_{k=j}^{i-1} cos(\phi_k)sin(\theta_k) + r_i cos(\lambda_i)sin(\gamma_i)$$
$$\Delta z_{ij}^{sb} = l \sum_{k=j}^{i-1} sin(\phi_k) + r_i sin(\lambda_i)$$

for $j > i$

$$\Delta x_{ij}^{sb} = -l \sum_{k=i}^{j-1} cos(\phi_k)cos(\theta_k) + r_i cos(\lambda_i)cos(\gamma_i)$$
$$\Delta y_{ij}^{sb} = -l \sum_{k=i}^{j-1} cos(\phi_k)sin(\theta_k) + r_i cos(\lambda_i)sin(\gamma_i)$$
$$\Delta z_{ij}^{sb} = -l \sum_{k=i}^{j-1} sin(\phi_k) + r_i sin(\lambda_i)$$

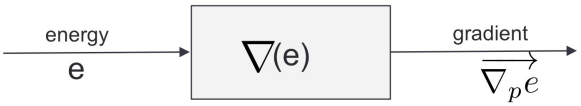
$$s_{ij}^{sb} = \frac{1}{r_i^2}\left[\Delta x_{ij}^{sb^2} + \Delta y_{ij}^{sb^2} + \Delta z_{ij}^{sb^2}\right]$$

$$E^{sb} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} e_{ij}^{sb}; \text{ where } e_{ii}^{sb} = 0$$

$$e_{ij}^{sb} = \theta(-k^{sb}(s_{ij}^{sb} - a^{sb^2})); \text{ where } a^{sb} = 1.05 \text{ and } k^{sb} = 25$$

Fig S7. Sidechain-backbone energy term calculated based on Cartesian coordinates in the structure.

Gradient



$$p^r = (\phi_0^r, \theta_0^r, \lambda_0^r, \gamma_0^r, \ldots, \phi_{n-2}^r, \theta_{n-2}^r, \lambda_{n-2}^r, \gamma_{n-2}^r, \lambda_{n-1}^r, \gamma_{n-1}^r)$$

$$p^{r+1} = p^r - \epsilon \nabla E(p^r)$$

$$\frac{\partial E}{\partial \phi_m} = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \frac{\partial e_{ij}}{\partial \phi_m} \Big[(\sqrt{i-j} - 1)\theta(-k e_{ij}) \big(1 - k\theta(k e_{ij}) e_{ij}\big) + 1 \Big]$$

$$\frac{\partial E}{\partial \phi_m} = \sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \frac{\partial e_{ij}}{\partial s_{ij}} \cdot \frac{\partial s_{ij}}{\partial \phi_m} \Big[(\sqrt{i-j} - 1)\theta(-k e_{ij}) \big(1 - k\theta(k e_{ij}) e_{ij}\big) + 1 \Big]$$

$$\frac{\partial e_{ij}}{\partial s_{ij}} = -k \Big[(1 - v_{ij})\theta\big(k(s_{ij} - a)\big)\theta\big(-k(s_{ij} - a)\big) + v_{ij}\theta\big(k(s_{ij} - b)\big)\theta\big(-k(s_{ij} - b)\big) \Big]$$

$$\frac{\partial s_{ij}}{\partial \phi_m} = \frac{2}{(r_i + r_j)^2} \Big[\Delta x_{ij} \cdot \frac{\partial \Delta x_{ij}}{\partial \phi_m} + \Delta y_{ij} \cdot \frac{\partial \Delta y_{ij}}{\partial \phi_m} + \Delta z_{ij} \cdot \frac{\partial \Delta z_{ij}}{\partial \phi_m} \Big]$$

$$\frac{\partial \Delta x_{ij}}{\partial \phi_m} = -l \cdot sin(\phi_m) cos(\theta_m)$$

$$\frac{\partial \Delta y_{ij}}{\partial \phi_m} = -l \cdot sin(\phi_m) sin(\theta_m)$$

$$\frac{\partial \Delta z_{ij}}{\partial \phi_m} = l \cdot cos(\theta_m)$$

Fig S8. Gradient of the energy can be directly calculated from the energy (score function). It can be used to explore the energy landscape and minimize the score and find the lowest energy structure.

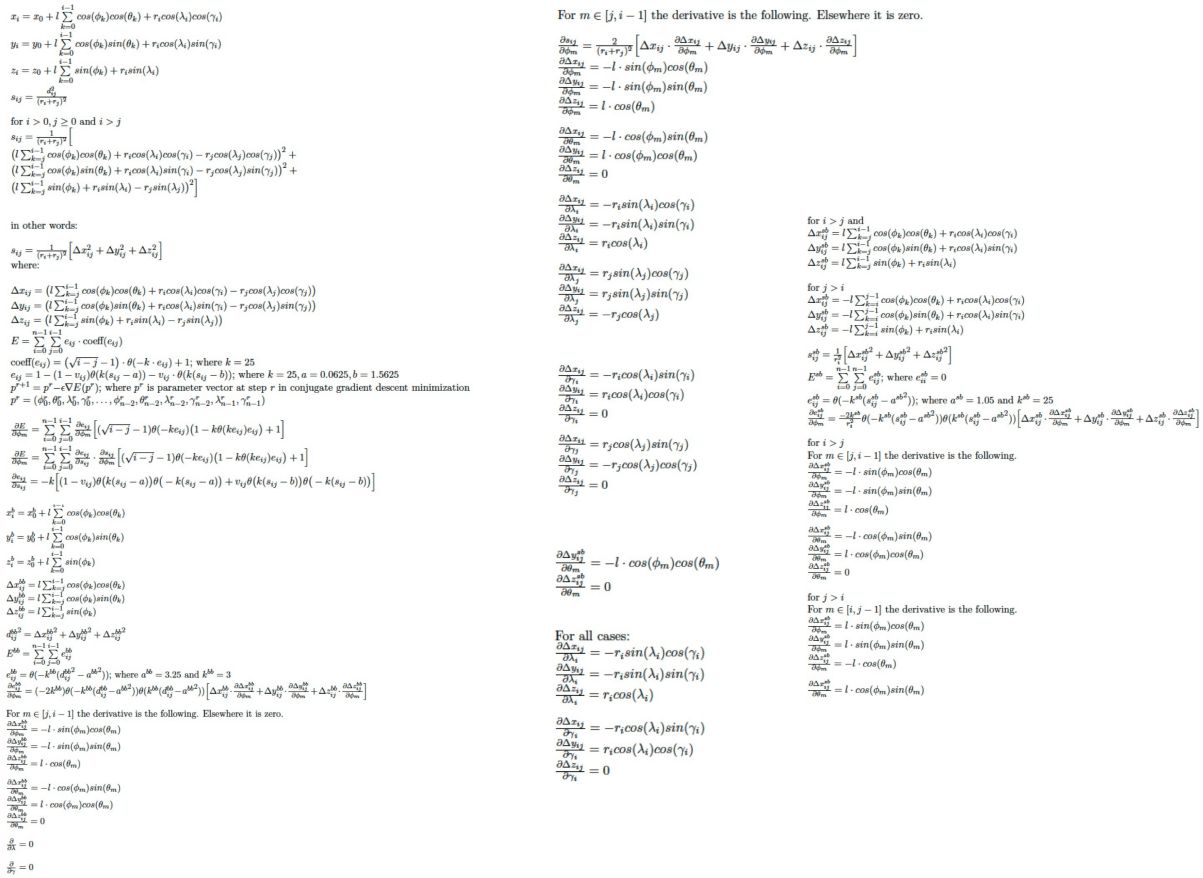


Fig S9. Coarse-grained model and scoring function and gradient of the score.

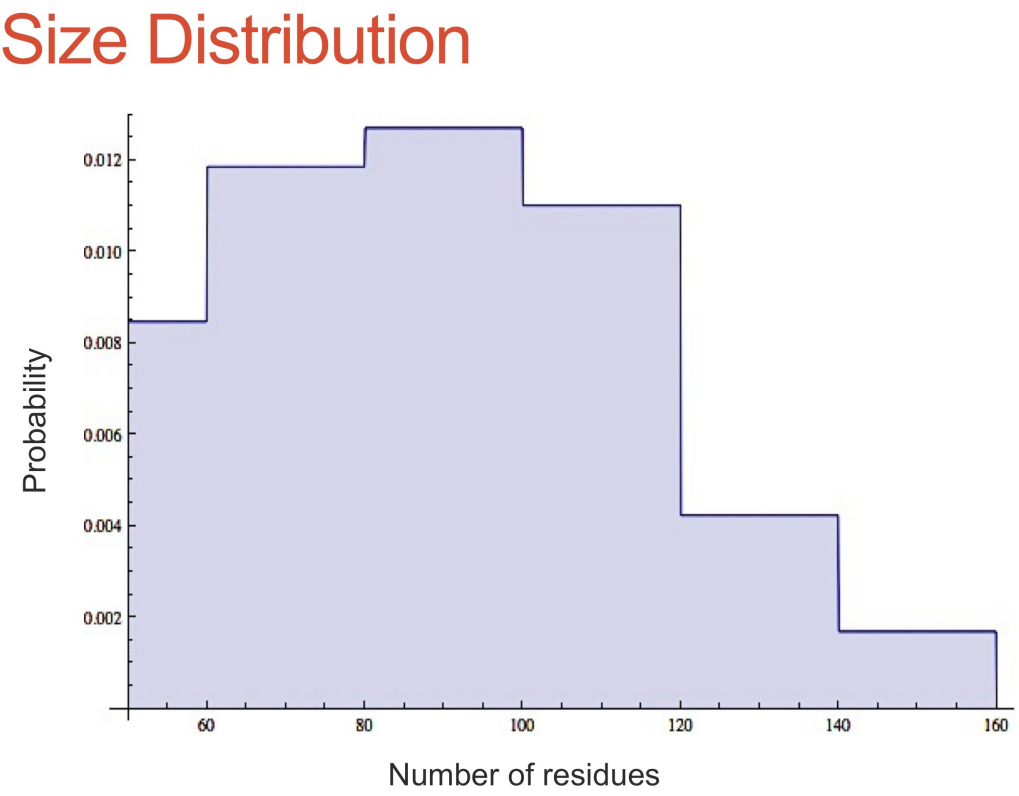


Fig S10. Size(number of amino acids) distribution of 59 high-resolution protein structures used to generate decoys.

Group	Amino Acids	Description
1	R, K, H	Positive
2	D, E	Negative
3	S, T, N, Q	Polar uncharged
4	P, G, C	Special
5	A, V	Small hydrophobic
6	I, L, M	Large hydrophobic
7	F, W, Y	Aromatic hydrophobic

Table 1. Amino acids are categorized into seven different groups based on their size and biochemical properties.

	1	2	3	4	5	6	7
1	+0.0	-1.0	-0.5	+0.0	+0.5	+1.0	+1.0
2	-1.0	+0.0	-0.5	+0.0	+0.5	+1.0	+1.0
3	-0.5	-0.5	-0.5	+0.0	+0.5	+1.0	+1.0
4	+0.0	+0.0	+0.0	-0.5	-0.5	+0.0	+0.0
5	+0.5	+0.5	+0.5	-0.5	-0.5	-0.5	-0.5
6	+1.0	+1.0	+1.0	+0.0	-0.5	-1.0	-1.0
7	+1.0	+1.0	+1.0	+0.0	-0.5	-1.0	-1.0

Table 2. Pairwise energy for residue *i* and *j* based on their amino acid group. Amino acids are categorized based on their biophysical properties. The energy corresponding two amino acids is given in this table. There are five energy levels for simplicity.

(a) Phoenix energy		Actual class			
		A	**B**	**C**	**D**
Predicted class (%)	**A**	24	14	13	9
	B	0	0	0	0
	C	76	86	87	91
	D	0	0	0	0

(b) Coarse grain		Actual class			
		A	**B**	**C**	**D**
Predicted class (%)	**A**	40	17	0	0
	B	36	33	20	0
	C	24	50	80	100
	D	0	0	0	0

(c) All features		Actual class			
		A	**B**	**C**	**D**
Predicted class (%)	**A**	76	17	0	0
	B	24	59	20	0
	C	0	24	73	60
	D	0	0	7	40

Table 3. Comparison of phoenix energy, coarse grain score, and all features in training the support vector machine in classifying decoys. Every individual decoy is classified into one of the four classes of A, B, C, and D that corresponds to 0-1, 1-3, 3-10, 10+ backbone rmsds. In all of the tables we demonstrate predicted class of decoys against the actual class of a decoy. Each column presents the distribution of decoys predicted in each of four classes. For instance, in table (a), 76% of decoys that belonged to class A were predicted to belong to class C (wrong prediction). As another example, in table (c), 59% of decoys that belonged to class B were predicted to belong to class B (correct prediction).

Appendix

Appendix A

top7.hh

```
#ifndef BDA_TOP7
#define BDA_TOP7
#include"../mpi/mpi_include.hh"

#ifdef BDA_PYTHON
          #ifdef BDA_XLC
          #include<iostream>
          #undef _POSIX_C_SOURCE
          #endif
#include<Python.h>
#endif

#include<iostream>
#include"../cpds-misc/random.hh"
#include"../cpds-misc/cpdsError.hh"
#include"../cpds-misc/cpdsOut.hh"
#include"../cpds/scoreData.hh"
#include"../misc/util.hh"
#include"../misc/Annealing.hh"
#include"../flexopt/energiesflex.hh"
#include"../cpds/mole_interface.hh"

#define INFTY 10E+33

using namespace std;

struct PositionRotamerPair
{
          int Pos;
          int Rotamer;
};

struct classcomp{
          bool operator() (const int& lhs, const int& rhs) const
          {
           return lhs < rhs;
          }
};

class mapStruct
{
public:
          mapStruct()
          {
           m_inv_is_updated = false;
          }
          ~mapStruct()
          {
          }
          void set(int first, int second)
          {
           m[first] = second;
           m_inv_is_updated = false;
          }
          int get(int first)
          {
           return m[first];
          }
          int get_inv(int second)
          {
           if(m_inv_is_updated==false)
             inverse();
           return m_inv[second];
          }
          void inverse()
          {
           m_inv.clear();
           for(It=m.begin();It!=m.end();It++)
           {
             m_inv[(*It).second] = (*It).first;
           }
           m_inv_is_updated = true;
          }
          void show()
          {
           for(It=m.begin();It!=m.end();It++)
           {
             OUT << (*It).first << ": " << (*It).second << endl;
           }
          }
          map<int,int> m;
          map<int,int> m_inv;
          bool m_inv_is_updated;
          map<int,int>::iterator It;
};


/*
class ReplicaConfig
{
public:
          ReplicaConfig()
```

```
        {
         degree = 0;
         precision = 0;
         kT = 0;
         E = INFTY;
         index = -1;
         rank = -1;
        }

        ~ReplicaConfig()
        {}

        double degree; //degree
        double precision; //precision
        double kT;
        double E; // energy
        int index;
        int rank; //mpi_rank

        vector<double> exportData()
        {
         vector<double> temp;
         temp.push_back(degree);
         temp.push_back(precision);
         temp.push_back(kT);
         temp.push_back(E);
         temp.push_back(double(index)+0.1);
         temp.push_back(double(rank)+0.1);
         return temp;
        }
        void importData(vector<double> data)
        {
         assert(data.size()==6);
         degree = data[0];
         precision = data[1];
         kT = data[2];
         E = data[3];
         index = int(data[4]);
         rank = int(data[5]);
        }
};
*/

class Top7
{
 public:
        Top7(molecules* _m,forceField* _f);
        ~Top7();
        vector<int> BackBonePerturb(int NumOfPosToPerturb,float MaxPerturb,float Precision);
        //typical values: NumPos = 3 to 5, Max =  2 deg,Preci = 0.1 deg
        int GreedyRotamerOptimization();//returns number of rotamers changed
        float EnergyCurrentConfig();
        void UpdateResidueEnergies();
        void WindowOptimization(int StartPos,
                        int StopPos,
                        float AcceptProbability,
                        int MinNumOfSimultanousPerturb,
                        int MinNumOfIterations,
                        int NumOfRepeatedEngToStop);
        //Excluding StopPos. If new config is better accept else accept with
        //AcceptProbability probability. It starts with Maximum number of positions
        //to start perturbing and then but not less than MinNum.. of simultanous
        //perturbations
        float FlexBkBn(int MaxNumOfBkBnPosToPerturb,
                 int NumOfIterations,
                 float MaxDegree,
                 float DegreePrecision,
                 float ThreshHold,
                 float AcceptProbability);

        void RossetaMove(int NumPos,
                   int Steps,
                   float MaxDegree,
                   float PrecisionDegree,
                   float KTi,
                   float KTf);

        float FlexBkBnAnnealing(int MaxNumOfBkBnPosToPerturb,
                        int NumOfSteps,
                        float MaxDegree,
                        float DegreePrecision,
                        float KTi,
                        float KTf);

        float DisruptStructure(int MaxNumOfBkBnPosToPerturb,
                        int NumOfIterations,
                        float MaxDegree,
                        float DegreePrecision,
                        float EnergyStepIncrease,
                        float ThreshHold,
                        float AcceptProbability);

   void ShuffleBkBn(int MaxNumOfBkBnPosToPerturb,
                    int NumOfIterations,
                    float MaxDegree,
                    float DegreePrecision);

        pair<int,float> NormalizeAngle(float alpha);
        float DeNormalizeAngle(pair<int,float> CaseAlpha);
        float CorrectedN_Ca_Ha(int p, int c);
        float CorrectedN_Ca_Ha(int p);
        float CorrectedN_Ca_Chain(int p, int c);
        float CorrectedN_Ca_Chain(int p);
        float CorrectedC_N_Ca_Ha(int p, int c);
        float CorrectedC_N_Ca_Ha(int p);
        float CorrectedC_N_Ca_Chain(int p, int c);
        float CorrectedC_N_Ca_Chain(int p);
```

```
void FixHa(int p, int c);
void FixHa(int p);
void FixChain(int p, int c);
void FixChain(int p);

void BackrubMove(int p1, int p2, float theta,int FixAtoms=0);
//0: Interpolation, -1:nothing, 1:real minimization
void Minimization(int pos, int rot);
void PreMinimization(int p, int c);
void PreMinimization(int p);
void FlexBackrubMonteCarlo(float MaxAngle,
                float Precision,
                float KTi,
                float KTf,
                int Steps,
                int Length=3,
                float BiasProbability=0.5,
                bool domin=false);

void FlexOstrichMonteCarlo(float MaxAngle,
                float Precision,
                float KTi,
                float KTf,
                int Steps,
                int Length,
                int MaxNumOfBkBnPosToPerturb,
                int Top7Steps,
                float SCProbability,
                float SCMinProbability,
                float BackrubProbability,
                float Top7Probability);

vector<vector<float> > PhiPsiTable(int Pos,
                      int Rot,
                      float Degree,
                      float Precision);

vector<vector<float> > PhiPsiTableMPI(int Pos,
                        int Rot,
                        float Degree,
                        float Precision);

map<string,float> BestPhiPsiPos(int Pos,
                    float Degree,
                    float Precision,
                    bool RotamerOptimization=true);
map<string,float> BestPhiPsiPosMPI(int Pos, float Degree, float Precision);
vector<vector<pair<float,int> > > BestPhiPsiPosMPI_HighPerformance(int Pos,
                                          int StartPos,
                                          int StopPos,
                                          float Degree,
                                          float Precision,
                                          bool RotamerOptimization=true);
//Note: There is sth wrong with enerygWindow in it consider revision....
//HighPerformance function combines PhiPsiTableMPI and BestPhiPsiPosMPI to gain some
//performance: Each thread checks all possible rotamers without recalcing
//eneryTerms which are not necessary to achieve some performance.

void TinyGrid(float Degree);

void TinyGrid3_Move_MPI(int pos1, float deg1, int pos2, float deg2, int pos3, float deg3);
pair<float,float> PhiPsiPair(float deg, int offset);
void GradientBestPhiPsi(float Degree,
              float Precision,
              bool  RotamerOptimization=true,
              int RecordSnapShots=false);
void GradientBestPhiPsiMPI(float Degree,
                float Precision,
                bool ExcludeAlphaHelices=false);
void GradientBestPhiPsiPosMPI(int Pos,
                  float Degree,
                      float Precision);

void GradientBestPhiPsiMPI_HighPerformance(int StartPos,
                           int StopPos,
                           float Degree,
                           float Precision,
                           bool ExcludeAlphaHelices=false,
                           int RecordSnapShots=false);
//RecordSnapShots: if false or 0 it doesn't write middle steps (pdb files)
//otherwise:
//SnapShot_RecordSnapShots_Rank_Pos.pdb

void GradientBestPhiPsiBackrubMPI_HighPerformance(int StartPos,
                                int StopPos,
                                float Degree,
                                float Precision,
                                bool ExcludeAlphaHelics=false,
                                int BackrubLength=3,
                                float BackrubDegree=4.0,
                                float BackrubPrecision=0.02);

void AllAtOnceGradientMPI_HighPerformance(int StartPos,
                          int StopPos,
                          float Degree,
                          float Precision,
                          bool ExcludeAlphaHelices=false,
                          bool RotamerOptimization=true);

void AllAtOnceGradientMPI_56Nodes(float Degree,
                     int Steps=1,
                     bool ExcludeAlphaHelices=false);
void AllAtOnceGradientMPI_xNodes(int numberOfNodes,
                    vector<float> Degrees,
                    bool ExcludeAlphaHelices=false);
```

```
void SomeAtOnceGradientMPI_xNodes(vector<int> Positions,
                     vector<float> Degrees);

void BestResidueGradientMPI_HighPerformance(int StartPos,
                            int StopPos,
                            float Degree,
                            float Precision,
                            bool ExcludeAlphaHelices=false);

void SomeAtOnceGradientMPI_HighPerformance(int StartPos,
                           int StopPos,
                           float Degree,
                           float Precision,
                           bool ExcludeAlphaHelices,
                           vector<int> Positions);

void StochasticGradientBestPhiPsiMPI(int NumOfIterations,
                       int MaxNumOfPositions,
                       float Degree,
                       float Precision,
                       bool ExcludeAlphaHelices);

void MonteCarloGradientBestPhiPsiMPI(float Degree,
                       float KTi,
                       float KTf,
                       int Steps,
                       bool fixedRange=false,
                       bool recordSnapshots=false,
                       char* Filename=NULL);

void StochasticPerturbationGridRefinement(vector<float> degrees,
                          vector<float> precision,
                          int NumOfStructures,
                          float MaxPerturbationAngle,
                          string path,
                          string name,
                          int Rank,
                          bool UseRamaCode=true,
                          bool UniformRama=false,
                          bool RecordSnapShots=false,
                          bool Code2=false,
                          bool doGRID=true);

void StochasticPerturbationBackrubRefinement(float MaxAngle,
                            float Precision,
                            float MaxPerturbationAngle,
                            float KTi,
                            float KTf,
                            int Steps,
                            int Length,//=3
                            float BiasProbability,//=0.5
                            bool domin,//=false
                            int NumOfStructures,
                            string path,
                            string name,
                            int Rank,
                            bool RecordSnapShots);


void StochasticRamaPerturbationDecoyGeneration(int NumOfStructures,
                              string path,
                              string name,
                              float PerturbationRate=0.1);


void ReplicaGRID(double degree,
          double precision,
          size_t steps,
          double kT);


void ReplicaExchangeMonteCarloGRID(vector<float> degrees,
                      vector<float> precisions,
                      int rounds,
                      int steps_sync,
                      vector<float> kTs,
                      int round=1);

void SimulatedAnnealingReplicaExchangeMonteCarloGRID(float kT_high_i,
                                  float kT_high_f,
                                  float kT_low_i,
                                  float kT_low_f,
                                  float degree_scale,
                                  int rounds,
                                  int rounds_of_rounds,
                                  int steps_sync);

void StochasticPDBFragmentRefinement();
void SnapShotRecorder(int SnapShot, int Pos);
vector<float> PhiPsiRepresentation();

void GreedyMinimization();
bool ByPassRank(float MaxDegree,float DegreePrecision);
map<string,float> CorrelationTwoPos(int Pos1, int Pos2);
void RotamerPerturb(int Position);
void RandomStructurePerturbation(float MaxAngle,bool Sequential=true);
void RamaStructurePerturbation(bool Uniform=false,
                   bool Code2=false,
                   float PerturbationRate=0.1);
//PerturbationRate is probability of a position to be perturbed

vector<vector<int> > GetIntMatrix(string filename);
vector<vector<float> > GetFloatMatrix(string filename);
template <class T> pair<T,pair<int,int> > max(vector<vector<T> > vec);
template <class T> pair<T,pair<int,int> > min(vector<vector<T> > vec);

//Walks through the structure residue by residue and perturbs each phi angle
```

```
          //and psi angle +/-rand(MaxAngle)

          void LoadRamaCode2DataStructures();
          void LoadRamaCodeDataStructures();
          vector<int> RamaCode2OfStructure();
          vector<char> RamaCodeOfStructure();

          pair<float,float> RamaRandomPhiPsi(int Code,
                                string Residue,
                                bool Uniform=false,
                                bool Code2=false);

          float Move();
          bool IsInAlphaHelix(int Position);
          void AddRotamerModification(int Position);
          void RestoreRotamerModification();
          void ClearRotamerModification();
          void SaveState();
          void RestoreState();

          template <class T> void ShowTable(vector<vector<T> > vec);
          int nump;
          molecules* m;
          vector<keyrange> AlphaHelices;
          molecules* backup;
          forceField* f;
          elec* e;
          occl* oc;
          bool USE_MM;
          minimizer* M;
          vector<coord> CoordsConfig;
          vector<int> RotamerConfig;
          vector<PositionRotamerPair> Modifications;
          vector<float> ResidueEnergies;

//        vector<pair<int,vector<pair<int,int> > > > RamaCodeReverse;

          vector<vector<int> > RamaCode2;
          //Newly designed code that is based on numbers here, 300 to 70
          vector<vector<char> > RamaCode;//RamaCode[i][j], i:row, j:col, i,j in [0,35]
          map<int,vector<pair<int,int> > > RamaCode2ToIndexes;
          map<char,vector<pair<int,int> > > RamaCodeToIndexes;

          map<string,vector<vector<int> > > ResidueRamaDistribution;
          map<string,int> ResidueNameToIndex;
 private:
};

extern map<string,Top7*> Top7Objects;
extern int NumOfTop7Objs;

struct Top7Error_BadPosition:public cpdsError
{Top7Error_BadPosition(string funcName, int pos);};

#ifdef BDA_PYTHON
PyObject* MakeTop7Object(PyObject* self, PyObject* args);
PyObject* GreedyRotamerOptimization(PyObject* self, PyObject* args);
PyObject* FlexBkBn(PyObject* self, PyObject* args);
PyObject* RossetaMove(PyObject* self, PyObject* args);
PyObject* FlexBkBnAnnealing(PyObject* self, PyObject* args);
PyObject* FlexBackrubMonteCarlo(PyObject* self, PyObject* args);
PyObject* FlexOstrichMonteCarlo(PyObject* self, PyObject* args);
PyObject* DisruptStructure(PyObject* self, PyObject* args);
PyObject* ShuffleBkBn(PyObject* self, PyObject* args);
PyObject* TinyGrid(PyObject* self, PyObject* args);
PyObject* TinyGrid3_Move_MPI(PyObject* self, PyObject* args);
PyObject* GradientBestPhiPsiMPI(PyObject* self, PyObject* args);
PyObject* GradientBestPhiPsi(PyObject* self, PyObject* args);
PyObject* GradientBestPhiPsiMPI_HighPerformance(PyObject* self, PyObject* args);
PyObject* GradientBestPhiPsiBackrubMPI_HighPerformance(PyObject* self, PyObject* args);
PyObject* AllAtOnceGradientMPI_HighPerformance(PyObject* self, PyObject* args);
PyObject* AllAtOnceGradientMPI_56Nodes(PyObject* self, PyObject* args);
PyObject* AllAtOnceGradientMPI_xNodes(PyObject* self, PyObject* args);
PyObject* SomeAtOnceGradientMPI_xNodes(PyObject* self, PyObject* args);
PyObject* BestResidueGradientMPI_HighPerformance(PyObject* self, PyObject* args);
PyObject* SomeAtOnceGradientMPI_HighPerformance(PyObject* self, PyObject* args);
PyObject* StochasticGradientBestPhiPsiMPI(PyObject* self, PyObject* args);
PyObject* MonteCarloGradientBestPhiPsiMPI(PyObject* self, PyObject* args);
PyObject* StochasticPerturbationGridRefinement(PyObject* self, PyObject* args);
PyObject* StochasticPerturbationBackrubRefinement(PyObject* self, PyObject* args);
PyObject* StochasticRamaPerturbationDecoyGeneration(PyObject* self, PyObject* args);
PyObject* ReplicaExchangeMonteCarloGRID(PyObject* self, PyObject* args);
PyObject* SimulatedAnnealingReplicaExchangeMonteCarloGRID(PyObject* self, PyObject* args);
PyObject* RandomStructurePerturbation(PyObject* self, PyObject* args);
PyObject* PhiPsiRepresentation(PyObject* self, PyObject* args);
PyObject* StochasticPDBFragmentRefinement(PyObject* self, PyObject* args);
PyObject* RamaStructurePerturbation(PyObject* self, PyObject* args);
PyObject* GreedyMinimization(PyObject* self, PyObject* args);
PyObject* AlignTwoMoleculesRMSD(PyObject* self, PyObject* args);
#endif

#endif
```

Appendix B

top7.cc

```
#include"./top7.hh"

map<string,Top7*> Top7Objects;
int NumOfTop7Objs=0;

Top7::Top7(molecules* _m,forceField* _f)
{
         m = _m;
         nump = m->numPositions();
         AlphaHelices = m->FindAlphaHelices();
         for(int i=0;i<AlphaHelices.size();i++)
         {
          OUT << "\nAlphaHelices: [" << AlphaHelices[i].start << "," << AlphaHelices[i].stop << ")" <<endl;
         }
         f = _f;

         UpdateResidueEnergies();
         Modifications.clear();
         /*
         ffTerm* temp = f->getTerm(0);
         vector<string> str = temp->getParamNames();
         OUT << "\nffTerm DESCRIPTIONS......... " << endl;
         for(int i=0;i<str.size();i++)
         {
          OUT << str[i] << endl;
         }
         string s;
         s = "Elec";
         vector<string> Elec;
         Elec.push_back(s);
         s = "Occl";
         vector<string> Occl;
         Occl.push_back(s);

         e = dynamic_cast<elec*>(f->getTerm(Elec));
         oc = dynamic_cast<occl*>(f->getTerm(Occl));


         M = new minimizer(*m,e,-1,oc);
         //M = new minimizer(*m,e,m->numPositions());
         USE_MM = false;
*/

/* if we wanna use the minimizer we need to uncomment.... */

         LoadRamaCodeDataStructures();
         LoadRamaCode2DataStructures();
}

Top7::~Top7()
{
         delete M;
}

vector<int> Top7::BackBonePerturb(int NumOfPosToPerturb, float MaxPerturb, float Precision)
{
         int RandomPosition, RandomAngle;
         float Alpha;
         vector<int> Positions;
//       OUT << "\n NumOfPosToPerturb = " << NumOfPosToPerturb << endl;
         for(int i=0;i<NumOfPosToPerturb;i++)
         {
         // do{
          RandomPosition = Rand.randomInt(nump);
          Positions.push_back(RandomPosition);
         // }while(IsInAlphaHelix(RandomPosition));
//        OUT << "\n 2.1. " << endl;
          RandomAngle = Rand.randomInt(2*int(MaxPerturb/Precision)+1);
//        OUT << "\n 2.2. " << endl;
          RandomAngle -= int(MaxPerturb/Precision);
//        OUT << "\n 2.3. " << endl;
          Alpha = RandomAngle * Precision;
          if(MPI_MASTER)
          {
            for(int _Rank=1;_Rank<MPI_SIZE;_Rank++)
            {
              mpiSend(_Rank,RandomPosition);
              mpiSend(_Rank,Alpha);
            }
          }
          else
          {
            mpiRecv(MPI_MASTER,RandomPosition);
            mpiRecv(MPI_MASTER,Alpha);
          }
//        OUT << "\n 2.4. " << endl;
          m->perturbPhiAngle(RandomPosition,Alpha);
//        OUT << "\n 2.5. " << endl;
          if(RandomPosition>0) m->perturbPsiAngle(RandomPosition-1,-Alpha);

//OUT << "\n 2.6. " << endl;
          //Compensating preceding psi angle perturbation but opposite direction
         }
         return Positions;
}

int Top7::GreedyRotamerOptimization()
```

```
{
         int NumOfChanges = 0;
         for(int i=0;i<nump;i++)
         {
          bool ChangeFlag = false;
          float BestEnergy = energyFlex(*m,i,*f); //= ResidueEnergies[i];
          int InitialConf = m->getActive(i);
          int BestConf = m->getActive(i);
          for(int j=0;j<m->numRots(i);j++)
          {
            m->setActive(i,j);
            float TempEnergy = energyFlex(*m,i,*f);
            if(TempEnergy<BestEnergy)
            {
              BestEnergy = TempEnergy;
              BestConf = j;
              ChangeFlag = true;
            }
          }
          if(ChangeFlag)
          {
            m->setActive(i,BestConf);
            NumOfChanges+=1;
          }
          else
          {
            m->setActive(i,InitialConf);
          }
         }
}

float Top7::EnergyCurrentConfig()
{
         float Result;
         if(USE_MM)
         {
          M->updateCoords();
          Result = M->getEnergy();
         }
         else
         {
          Result = energyFlex(*m,*f);
         }
         return Result;
}

void Top7::UpdateResidueEnergies()
{
         ResidueEnergies.clear();
         for(int i=0;i<nump;i++)
         {
          OUT << "\n" <<  i << endl;
          ResidueEnergies.push_back(energyFlex(*m,i,*f));
         }
}

void Top7::WindowOptimization(int StartPos,
                              int StopPos,
                              float AcceptProbability,
                              int MinNumOfSimultanousPerturb,
                              int MinNumOfIterations,
                              int NumOfRepeatedEngToStop)
{
//       OUT << "\n>>>Entered Top7::WindowOptimization() " << endl;
         int length = StopPos - StartPos;
         int NumOfRepeatedBestEng = 0;
         int counter = 0;
         set<int> Perturbations;
         int NumOfRotsToPerturb = length;
         float BestEnergy = energyFlexWindow(*m,StartPos,StopPos,*f);
         do{
          Perturbations.clear();
          int RandNumOfPerturbs = Rand.randomInt(NumOfRotsToPerturb);
          for(int i=0;i<RandNumOfPerturbs;i++)
          {
            int RandPos = Rand.randomInt(length);
            Perturbations.insert(RandPos);
          }
          set<int>::iterator It;
//        OUT <<"\n\n---- Perturbations.size() = " << Perturbations.size() << endl;
          ClearRotamerModification();
          for(It=Perturbations.begin();It!=Perturbations.end();It++)
          {
//          OUT << "\n"<<*It<<" Is being purturbed";
            AddRotamerModification(*It);
            RotamerPerturb(*It);
          }
          float Energy = energyFlexWindow(*m,StartPos,StopPos,*f);
//        OUT << "\nEnergy FlexWindow is : "<<Energy<<endl;
          if(Energy<BestEnergy)
          {
            ClearRotamerModification();
            BestEnergy = Energy;
            NumOfRepeatedBestEng = 0;
          }
          else if(Rand.randomInt(1000)/1000.0>AcceptProbability)
          {
            RestoreRotamerModification();
            NumOfRepeatedBestEng++;
          }
          else
          {
            ClearRotamerModification();
            BestEnergy = Energy;
            NumOfRepeatedBestEng = 0;
          }
          if(NumOfRotsToPerturb>MinNumOfSimultanousPerturb)
```

```
            NumOfRotsToPerturb--;
          counter++;
         }while((NumOfRepeatedBestEng<=NumOfRepeatedEngToStop)||(counter<MinNumOfIterations));
//       OUT << "\n<<<Left Top7::WindowOptimization() " << endl;
}

float Top7::FlexBkBn(int MaxNumOfBkBnPosToPerturb,
                     int NumOfIterations,
                     float MaxDegree,
                     float DegreePrecision,
                     float ThreshHold,
                     float AcceptProbability)
{
// Typical Values...
//       float ThreshHold = 0.5;
//       float MaxDegree = 3;
//       float DegreePrecision = 0.1;
//       int MaxNumOfBkBnPosToPerturb = 10;
// float AcceptProbability = 0.25

//   int Steps = 40;
//       int Min = 5;
//       int Counter = 0;
         vector<keyrange> Alpha = m->FindAlphaHelices();
         for(int i=0;i<Alpha.size();i++)
         {
          OUT << "\n"<<i<<". ["<<Alpha[i].start<<", "<<Alpha[i].stop<<"]"<<endl;
         }
         float InitialEng = EnergyCurrentConfig();
         OUT << "\nInitialEnergy : " << InitialEng << endl;
         SaveState();
         for(int i=0;i<NumOfIterations;i++)
         {
          int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
          BackBonePerturb(RandomNumOfPerturb,MaxDegree,DegreePrecision);
          float CurrentEng = EnergyCurrentConfig();
          if(CurrentEng<InitialEng)
          {
            SaveState();
            InitialEng = CurrentEng;
          }
          else if(CurrentEng-InitialEng<ThreshHold)
          {
            int Random = Rand.randomInt(1000);
            if(Random>1000*AcceptProbability)
            {
              SaveState();
              InitialEng = CurrentEng;
            }
            else
            {
              RestoreState();
            }
          }
          else
          {
            RestoreState();
          }
          OUT << i << "\t" << EnergyCurrentConfig() << endl;
         }
         return EnergyCurrentConfig();
}

void Top7::RossetaMove(int NumPos,
                       int Steps,
                       float MaxDegree,
                       float PrecisionDegree,
                       float KTi,
                       float KTf)
{
         bool InPlace = true;
         float DKT = (KTi-KTf)/Steps;
         for(float KT = KTi;KT>KTf;KT-=DKT)
         {
          m->SaveAll(InPlace);
          float Ei = energyFlex(*m,*f);
          for(int i=0;i<NumPos;i++)
          {
            int RandPos = Rand.randomInt(NumPos);
            float Angle = Rand.randomFloat(2*MaxDegree)-MaxDegree;
            float Phi = m->GetPhi(RandPos);
            float Psi = m->GetPsi(RandPos);
            m->SetPhi(RandPos,Phi+Angle);
            m->SetPsi(RandPos,Psi-Angle);
          }
          float Ef = energyFlex(*m,*f);
          float dE = Ef - Ei;
          if(dE>0)
          {
            float Coin = Rand.randomFloat(1);
            float P = exp(-dE/KT);
            if(Coin>P)
            {
              m->RewindAll(InPlace);
            }
          }
          OUT << KT << ": " << Ef << endl;
         }
}

float Top7::FlexBkBnAnnealing(int MaxNumOfBkBnPosToPerturb,
                              int NumOfSteps,
                              float MaxDegree,
                              float DegreePrecision,
                              float KTi,
                              float KTf)
{// Typical Values...
```

```
//       float ThreshHold = 0.5;
//       float MaxDegree = 3;
//       float DegreePrecision = 0.1;
//       int MaxNumOfBkBnPosToPerturb = 10;
// float AcceptProbability = 0.25

//   int Steps = 40;
//       int Min = 5;
//       int Counter = 0;
         vector<keyrange> Alpha = m->FindAlphaHelices();
         for(int i=0;i<Alpha.size();i++)
         {
          OUT << "\n"<<i<<". ["<<Alpha[i].start<<", "<<Alpha[i].stop<<"]"<<endl;
         }
         float InitialEng = EnergyCurrentConfig();
         OUT << "\nInitialEnergy : " << InitialEng << endl;
         SaveState();
         float DelKT= (KTf-KTi)/NumOfSteps;
         for(float KT=KTi;KT>KTf;KT+=DelKT)
         {
          int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
          BackBonePerturb(RandomNumOfPerturb,MaxDegree,DegreePrecision);
          float CurrentEng = EnergyCurrentConfig();
          if(CurrentEng<InitialEng)
          {
            SaveState();
            InitialEng = CurrentEng;
          }
          else
          {
            float Random = Rand.randomFloat(1);
            float TransitionProbability = exp(-(CurrentEng-InitialEng)/KT);
            if(Random<TransitionProbability)
            {
              SaveState();
              InitialEng = CurrentEng;
            }
            else
            {
              RestoreState();
            }
          }
          OUT << KT << ":\t" << EnergyCurrentConfig() << endl;
         }
         return EnergyCurrentConfig();
}

float Top7::DisruptStructure(int MaxNumOfBkBnPosToPerturb,
                             int NumOfIterations,
                             float MaxDegree,
                             float DegreePrecision,
                             float EnergyStepIncrease,
                             float ThreshHold,
                             float AcceptProbability)
{
// Typical Values...
//       float ThreshHold = 0.5;
//       float MaxDegree = 3;
//       float DegreePrecision = 0.1;
//       int MaxNumOfBkBnPosToPerturb = 10;
// float AcceptProbability = 0.25

//   int Steps = 40;
//       int Min = 5;
//       int Counter = 0;
         vector<keyrange> Alpha = m->FindAlphaHelices();
         for(int i=0;i<Alpha.size();i++)
         {
          OUT << "\n"<<i<<". ["<<Alpha[i].start<<", "<<Alpha[i].stop<<"]"<<endl;
         }
         float InitialEng = EnergyCurrentConfig();
         OUT << "\nInitialEnergy : " << InitialEng << endl;
         SaveState();
         for(int i=0;i<NumOfIterations;i++)
         {
          int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
          BackBonePerturb(RandomNumOfPerturb,MaxDegree,DegreePrecision);
          float CurrentEng = EnergyCurrentConfig();
          if(InitialEng-ThreshHold<CurrentEng && CurrentEng<InitialEng+EnergyStepIncrease)
          {
            SaveState();
            InitialEng = CurrentEng;
          }
          else
          {
            int Random = Rand.randomInt(1000);
            if(Random>1000*AcceptProbability)
            {
              SaveState();
              InitialEng = CurrentEng;
            }
            else
            {
              RestoreState();
            }
          }
          OUT << i << "\t" << EnergyCurrentConfig() << endl;
         }
         OUT << "\nEnergy = " << EnergyCurrentConfig() << endl;
         return EnergyCurrentConfig();
}

void Top7::ShuffleBkBn(int MaxNumOfBkBnPosToPerturb,
                       int NumOfIterations,
                       float MaxDegree,
                       float DegreePrecision)
{
```

```
          vector<keyrange> Alpha = m->FindAlphaHelices();
          for(int i=0;i<NumOfIterations;i++)
          {
           //int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
           int RandomNumOfPerturb = MaxNumOfBkBnPosToPerturb;
           if(MPI_MASTER)
           {
             for(int _Rank=1;_Rank<MPI_SIZE;_Rank++)
             {
               mpiSend(_Rank,RandomNumOfPerturb);
             }
           }
           else
           {
             mpiRecv(MPI_MASTER,RandomNumOfPerturb);
           }
           BackBonePerturb(RandomNumOfPerturb,MaxDegree,DegreePrecision);
          }
}

pair<int,float> Top7::NormalizeAngle(float alpha)
{
          float alphaPrime;
          int Case;
          if(0<=alpha && alpha<=180)
          {
           alphaPrime = alpha;
           Case = 1;
          }
          else if(180<alpha && alpha<=360)
          {
           alphaPrime = 360-alpha;
           Case = 2;
          }
          else if(-180<=alpha && alpha<0)
          {
           alphaPrime = -alpha;
           Case = 3;
          }
          else if(-360<alpha && alpha<-180)
          {
           alphaPrime = 360+alpha;
           Case = 4;
          }
          return pair<int,float>(Case,alphaPrime);
}

float Top7::DeNormalizeAngle(pair<int,float> CaseAlpha)
{
          float alpha = CaseAlpha.second;
          int Case = CaseAlpha.first;
          switch(Case)
          {
           case 1:
             return alpha;
             break;
           case 2:
             return 360-alpha;
             break;
           case 3:
             return -alpha;
             break;
           case 4:
             return -360+alpha;
             break;
          }
}

float Top7::CorrectedN_Ca_Ha(int p, int c)
{
          float x;
          float Angle = m->getAngleN_Ca_C(p,c);
          pair<int,float> Normal = NormalizeAngle(Angle);
          Angle = Normal.second;
          if(70<=Angle && Angle<=140)
          {
           x = toRadians(Angle);
           if((m->wtRestype(p)).compare("GLY")!=0)
             Angle = toDegrees(-0.0876*x*x + 0.1357*x + 1.9709);
           else
             Angle = toDegrees(-0.1008*x*x + 0.2001*x + 1.8962);
          }
          return DeNormalizeAngle(pair<int,float>(Normal.first,Angle));
}


float Top7::CorrectedN_Ca_Ha(int p)
{
          CorrectedN_Ca_Ha(p,m->getActive(p));
}

float Top7::CorrectedN_Ca_Chain(int p, int c)
{
          float x;
          float Angle = m->getAngleN_Ca_C(p,c);
          pair<int,float> Normal = NormalizeAngle(Angle);
          Angle = Normal.second;
          if(70<=Angle && Angle<=140)
          {
           x = toRadians(Angle);
           if((m->wtRestype(p)).compare("GLY")!=0)
             Angle = toDegrees(-0.0877*x*x + 0.1360*x + 1.9707);
           else
             Angle = toDegrees(-0.1008*x*x + 0.2001*x+ 1.8962);
          }
          return DeNormalizeAngle(pair<int,float>(Normal.first,Angle));
}
```

```
float Top7::CorrectedN_Ca_Chain(int p)
{
         CorrectedN_Ca_Chain(p,m->getActive(p));
}

float Top7::CorrectedC_N_Ca_Ha(int p, int c)
{
         float x;
         float Angle = m->getAngleN_Ca_C(p,c);
         pair<int,float> Normal = NormalizeAngle(Angle);
         Angle = Normal.second;
         if(70<=Angle && Angle<=140)
         {
          x = toRadians(Angle);
          if((m->wtRestype(p)).compare("GLY")!=0)
            Angle = toDegrees(-0.0719*x*x + 0.5080*x + 1.3853);
          else
            Angle = toDegrees(-0.0790*x*x + 0.5629*x + 1.3072);
         }
         return DeNormalizeAngle(pair<int,float>(Normal.first,Angle));
}

float Top7::CorrectedC_N_Ca_Ha(int p)
{
         CorrectedC_N_Ca_Ha(p,m->getActive(p));
}

float Top7::CorrectedC_N_Ca_Chain(int p, int c)
{
         float x;
         float Angle = m->getAngleN_Ca_C(p,c);
         pair<int,float> Normal = NormalizeAngle(Angle);
         Angle = Normal.second;
         if(70<=Angle && Angle<=140)
         {
          x = toRadians(Angle);
          if((m->wtRestype(p)).compare("GLY")!=0)
            Angle = toDegrees(0.0719*x*x - 0.5081*x - 1.3853);
          else
            Angle = toDegrees(0.0790*x*x - 0.5629*x - 1.3072);
         }
         return -DeNormalizeAngle(pair<int,float>(Normal.first,Angle));
}

float Top7::CorrectedC_N_Ca_Chain(int p)
{
         CorrectedC_N_Ca_Chain(p,m->getActive(p));
}

void Top7::FixHa(int p, int c)
{
//       OUT << "\n--------------------" << endl;
//       OUT << "\nAngleHa was: " << m->getAngleHa(p,c) << endl;
//       OUT << "\nCorrectedHa_Angle: " << CorrectedN_Ca_Ha(p,c) << endl;
         m->setAngleHaCaN(p,c,CorrectedN_Ca_Ha(p,c));
//       OUT << "\nAngleHa is: " << m->getAngleHa(p,c) << endl;

//       OUT << "\nDihedralHa was: " << m->getDihedralHa(p,c) << endl;
//       OUT << "\nCorrectedDihedralHa_Angle: " << CorrectedC_N_Ca_Ha(p,c) << endl;
         m->setDihedralHa_Ca_N_C(p,c,CorrectedC_N_Ca_Ha(p,c));
//       OUT << "\nDihedralHa is: " << m->getDihedralHa(p,c) << endl;
}

void Top7::FixHa(int p)
{
         FixHa(p,m->getActive(p));
}

void Top7::FixChain(int p, int c)
{
//       OUT << "\nAlpha: " << m->getAngleN_Ca_C(p,c) << endl;
//       OUT << "\n--------------------" << endl;
//       OUT << "\nAngleChain was: " << m->getAngleChain(p,c) << endl;
//       OUT << "\nCorrectedChain_Angle: " << CorrectedN_Ca_Chain(p,c) << endl;
         m->setAngleChainCaN(p,c,CorrectedN_Ca_Chain(p,c));
//       OUT << "\nAngleChain is: " << m->getAngleChain(p,c) << endl;

//       OUT << "\nDihedralChain was: " << m->getDihedralChain(p,c) << endl;
//       OUT << "\nCorrectedDihedralChain_Angle: " << CorrectedC_N_Ca_Chain(p,c) << endl;
         m->setDihedralChain_Ca_N_C(p,c,CorrectedC_N_Ca_Chain(p,c));
//       OUT << "\nDihedralChain is: " << m->getDihedralChain(p,c) << endl;
}

void Top7::FixChain(int p)
{
         FixChain(p,m->getActive(p));
}

void Top7::BackrubMove(int p1, int p2, float theta,int FixAtoms)
{
         if(p1<0 || p1>=m->numPositions())
          throw Top7Error_BadPosition("BackrubMove",p1);
         else if(p2<0 || p2>=m->numPositions())
          throw Top7Error_BadPosition("BackrubMove",p2);
         if((m->wtRestype(p1)).compare("GLY")==0||(m->wtRestype(p2)).compare("GLY")==0)
          return;
         if((m->wtRestype(p1)).compare("PRO")==0||(m->wtRestype(p2)).compare("PRO")==0)
          return;

         const keyrange& key1 = m->getConf(p1);
         const keyrange& key2 = m->getConf(p2);
         int Ca1 = getAtom(m->atoms,key1,padstringCA);
         int Ca2 = getAtom(m->atoms,key2,padstringCA);
         int H1  = getAtom(m->atoms,key1,padstringH);
         int H2  = getAtom(m->atoms,key2,padstringH);
```

```
          set<keyrange,keyrangeCmp> atomset;
          if(Ca1>Ca2)
          {
           int tempCa=Ca1;
           Ca1 = Ca2;
           Ca2 = tempCa;
          }
          m->getAtomsBkBn(Ca1,p1,Ca2,p2,atomset);
          m->perturbTorsion2(Ca1,Ca2,atomset,theta);
          m->perturbTorsion2(Ca1,Ca2,H1,-theta);
          m->perturbTorsion2(Ca1,Ca2,H2,theta);

          switch(FixAtoms)
          {
           case 0: // interpolate
           {
             PreMinimization(p1);
             PreMinimization(p2);
           }
           break;
           case -1:
           {
           //do nothing....
           }
           break;
           case 1://minimize
           {
             Minimization(p1,m->getActive(p1));
             Minimization(p2,m->getActive(p2));
           }
           break;
          }
}

void Top7::Minimization(int pos, int rot)
{
//        int CurrentRot = m->getActive(pos);
          m->setActive(pos,rot);
          vector<int> plist;
          plist.push_back(pos);
          minimizer myM(*m,plist,e);
          set<int> atoms;
          keyrange k = m->getConf(pos,rot);
          for(int i=k.start;i<k.stop;i++)
          {
           atoms.insert(i);
          }
          myM.SetActiveAtoms(atoms);
          myM.minimize();
//        m->setActive(pos,CurrentRot);
}

void Top7::PreMinimization(int p, int c)
{
          FixHa(p,c);
          FixChain(p,c);
}

void Top7::PreMinimization(int p)
{
          PreMinimization(p,m->getActive(p));
}

void Top7::FlexBackrubMonteCarlo(float MaxAngle,
                                 float Precision,
                                 float KTi,
                                 float KTf,
                                 int Steps,
                                 int Length,
                                 float BiasProbability,
                                 bool domin)
{
//Some other configurations are done here in place, but we can make it more
//general to accomodate all possible configuratons
          int FIXMIN = 1;
          int FIXINTERPOLATE=0;
          int DONTFIX=-1;
          bool SChainPerturbation, BBonePerturbation;
          float DelKT = (KTf-KTi)/Steps;
          float CurrentEnergy;
          float BestEnergy;
          m->SaveAll();
          for(float KT=KTi; KT>KTf; KT+=DelKT)
          {
           SChainPerturbation = Rand.randomFloat(1)>=BiasProbability;
           BBonePerturbation = !SChainPerturbation;
           if(SChainPerturbation)
           {
             int RandomPos=Rand.randomInt(m->numPositions());
             int r1 = m->getActive(RandomPos);
             float E1 = energyFlex(*m,RandomPos,*f);
             int r2 = Rand.randomInt(m->numRots(RandomPos));
             m->setActive(RandomPos,r2);
             float E2 = energyFlex(*m,RandomPos,*f);
             bool accept;
             float DelE = E2-E1;
             if(DelE<=0)
               accept = true;
             else
               accept=(exp((-DelE)/KT) >= Rand.randomFloat(1));
             if(!accept)
             {
               m->setActive(RandomPos,r1);
             }
             m->SaveInPlace(RandomPos);
           }
           else
```

```
          {
            int HeadPos = Rand.randomInt(m->numPositions()-Length);
            int TailPos = HeadPos+Length;
            m->SaveInPlace(HeadPos);
            m->SaveInPlace(TailPos);
            int NumOfGrids = ceil(MaxAngle/Precision);
            float Angle = Precision*(Rand.randomInt(2*NumOfGrids)-NumOfGrids);
            float E1 = energyFlex(*m,*f);//energyFlexWindow(*m,HeadPos,TailPos+1,*f);
            if(domin)
              BackrubMove(HeadPos,TailPos,Angle,FIXMIN);
            else
              BackrubMove(HeadPos,TailPos,Angle,FIXINTERPOLATE);
            float E2 = energyFlex(*m,*f);//energyFlexWindow(*m,HeadPos,TailPos+1,*f);
            float DelE = E2-E1;
            bool accept;
            if(DelE<=0)
              accept = true;
            else
              accept=(exp(-DelE/KT) >= Rand.randomFloat(1));
            if(!accept)
            {
              //m->RewindAll();
              //m->SaveAll();
              BackrubMove(HeadPos,TailPos,-Angle,DONTFIX);
              m->RewindInPlace(HeadPos);
              m->RewindInPlace(TailPos);
            }
            else
            {
              vector<int> Positions;
              for(int Pos=HeadPos;Pos<=TailPos;Pos++)
                Positions.push_back(Pos);
              m->ReAlignRotamers(Positions);
              for(int rot=1;rot<m->numRots(HeadPos);rot++)
              {
                if(!domin)
                  PreMinimization(HeadPos,rot);
                else
                  Minimization(HeadPos,rot);
              }
              for(int rot=1;rot<m->numRots(TailPos);rot++)
              {
                if(!domin)
                  PreMinimization(TailPos,rot);
                else
                  Minimization(TailPos,rot);
              }
              //m->SaveAll(true);
            }
          }
          CurrentEnergy = energyFlex(*m,*f);
          OUT << KT << "\t: " << CurrentEnergy << endl;
          if(CurrentEnergy<BestEnergy)
          {
            BestEnergy= CurrentEnergy;
            m->SaveAll(true);
          }
         }
         OUT << "\n\nFlexBackrub: " << BestEnergy << endl;
}


void Top7::FlexOstrichMonteCarlo(float MaxAngle,
                            float Precision,
                            float KTi,
                            float KTf,
                            int Steps,
                            int Length,
                            int MaxNumOfBkBnPosToPerturb,
                            int Top7Steps,
                            float SCProbability,
                            float SCMinProbability,
                            float BackrubProbability,
                            float Top7Probability)
{
//        float SCProbability=0.45;
//        float BackrubProbability=0.45;
//        float Top7Probability=0.1;
          bool SChainPerturbation=false;
          bool SCMinPerturbation=false;
          bool BackrubPerturbation=false;
          bool Top7Perturbation=false;
//        int Top7Steps = 25;
          float WhichMethod;
          float DelKT = (KTf-KTi)/Steps;
          float CurrentEnergy;
          float BestEnergy;
          m->SaveAll();
          for(float KT=KTi; KT>KTf; KT+=DelKT)
          {
           WhichMethod = Rand.randomFloat(1);
           if(WhichMethod<=SCProbability)
           {
             SChainPerturbation=true;
             SCMinPerturbation=false;
             BackrubPerturbation=false;
             Top7Perturbation=false;
           }
           else if(WhichMethod<=SCProbability+SCMinProbability)
           {
             SChainPerturbation=false;
             SCMinPerturbation=true;
             BackrubPerturbation=false;
             Top7Perturbation=false;
           }
           else if(WhichMethod<=(SCProbability+SCMinProbability+BackrubProbability))
           {
```

```
  SChainPerturbation=false;
  SCMinPerturbation=false;
  BackrubPerturbation=true;
  Top7Perturbation=false;
}
else
{
  SChainPerturbation=false;
  SCMinPerturbation=false;
  BackrubPerturbation=false;
  Top7Perturbation=true;
}
if(SChainPerturbation)
{
  //OUT << "\n SC..... " << endl;
  int RandomPos=Rand.randomInt(m->numPositions());
  int r1 = m->getActive(RandomPos);
  float E1 = energyFlex(*m,RandomPos,*f);
  int r2 = Rand.randomInt(m->numRots(RandomPos));
  m->setActive(RandomPos,r2);
  float E2 = energyFlex(*m,RandomPos,*f);
  bool accept;
  float DelE = E2-E1;
  if(DelE<=0)
    accept = true;
  else
    accept=(exp((-DelE)/KT) >= Rand.randomFloat(1));
  if(!accept)
  {
    m->setActive(RandomPos,r1);
  }
  m->SaveInPlace(RandomPos);
}
else if(SCMinPerturbation)
{
  int RandomPos=Rand.randomInt(m->numPositions());
  m->SaveInPlace(RandomPos);
  int r1 = m->getActive(RandomPos);
  float E1 = energyFlex(*m,RandomPos,*f);
  Minimization(RandomPos,r1);
  float E2 = energyFlex(*m,RandomPos,*f);
  bool accept;
  float DelE = E2-E1;
  if(DelE<=0)
    accept = true;
  else
    accept=(exp((-DelE)/KT) >= Rand.randomFloat(1));
  if(!accept)
  {
    m->RewindInPlace(RandomPos);
  }
  m->SaveInPlace(RandomPos);
}
else if(BackrubPerturbation)
{
  //OUT << "\n Backrub..... " << endl;
  int HeadPos = Rand.randomInt(m->numPositions()-Length);
  int TailPos = HeadPos+Length;
  m->SaveInPlace(HeadPos);
  m->SaveInPlace(TailPos);
  int NumOfGrids = ceil(MaxAngle/Precision);
  float Angle = Precision*(Rand.randomInt(2*NumOfGrids)-NumOfGrids);
  float E1 = energyFlex(*m,*f);//energyFlexWindow(*m,HeadPos,TailPos+1,*f);
  BackrubMove(HeadPos,TailPos,Angle);
  float E2 = energyFlex(*m,*f);//energyFlexWindow(*m,HeadPos,TailPos+1,*f);
  float DelE = E2-E1;
  bool accept;
  if(DelE<=0)
    accept = true;
  else
    accept=(exp(-DelE/KT) >= Rand.randomFloat(1));
  if(!accept)
  {
    //m->RewindAll();
    //m->SaveAll();
    BackrubMove(HeadPos,TailPos,-Angle);
    m->RewindInPlace(HeadPos);
    m->RewindInPlace(TailPos);
  }
  else
  {
    vector<int> Positions;
    for(int Pos=HeadPos;Pos<=TailPos;Pos++)
      Positions.push_back(Pos);
    m->ReAlignRotamers(Positions);
    for(int rot=1;rot<m->numRots(HeadPos);rot++)
    {
      PreMinimization(HeadPos,rot);
      //Minimization(HeadPos,rot);
    }
    for(int rot=1;rot<m->numRots(TailPos);rot++)
    {
      PreMinimization(TailPos,rot);
      //Minimization(TailPos,rot);
    }
    //m->SaveAll(true);
  }
}
else if(Top7Perturbation)
{
  //OUT << "\n Top7..... " << endl;
  for(int i=0;i<Top7Steps;i++)
  {
    //OUT << "\n" << i << endl;
    int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
    m->SaveAll(true);
    float E1 = energyFlex(*m,*f);
```

```
            BackBonePerturb(RandomNumOfPerturb,MaxAngle,Precision);
            float E2 = EnergyCurrentConfig();
            float DelE = E2-E1;
            bool accept;
            if(DelE<0)
            {
              accept =true;
            }
            else
            {
              float Random = Rand.randomFloat(1);
              float TransitionProbability = exp(-(DelE)/KT);
              if(Random<TransitionProbability)
              {
                accept =true;
              }
              else
              {
                accept = false;
              }
            }
            if(accept)
            {
              m->SaveAll(true);
            }
            else
            {
              m->RewindAll();
              m->SaveAll();
            }
          }
          m->ReAlignRotamers();
        }
        CurrentEnergy = energyFlex(*m,*f);
        OUT << KT << "\t: " << CurrentEnergy << endl;
        if(CurrentEnergy<BestEnergy)
        {
          BestEnergy= CurrentEnergy;
          m->SaveAll(true);
        }
       }
       OUT << "\n\nFlexOstrichMonteCarlo: " << BestEnergy << endl;
}


//e.g. Degree=2, Precision=0.1,===>  -2, -1.9, ... , 1.9, 2
// i->Phi, j->Psi
vector<vector<float> > Top7::PhiPsiTable(int Pos,
                                         int Rot,
                                         float Degree,
                                         float Precision)
{
        m->SaveAll();
        vector<vector<float> > results;
        if(Rot>=m->numRots(Pos)) throw moleculesError_badRotNum(posID(Pos),Rot);
        int NumAngles = ceil(Degree/Precision);
        float Phi, Psi;
        float Energy;
        float CurrentPhi = m->GetPhi(Pos);
        float CurrentPsi = m->GetPsi(Pos);
        int CurrentRot = m->getActive(Pos);
        float DeltaPhi = Precision;
        float DeltaPsi = Precision;
        OUT << "\nCurrentPhi: " << CurrentPhi;
        OUT << "\nCurrentPsi: " << CurrentPsi;
        OUT << endl;

        m->setActive(Pos,Rot);
        OUT << "\nRot: " << Rot << endl;
        for(int i=0; i<2*NumAngles+1;i++)
        {
         Phi = CurrentPhi+(i-NumAngles)*DeltaPhi;
         vector<float> Temp;
         OUT << int(float(i)/float(2*NumAngles+1)*100) <<"% -> ";
         OUT.flush();
         for(int j=0;j<2*NumAngles+1;j++)
         {
           Psi = CurrentPsi+(j-NumAngles)*DeltaPsi;
           m->SetPhi(Pos,Phi);
           m->SetPsi(Pos,Psi);
           Energy = energyFlex(*m,*f);
           Temp.push_back(Energy);
         }
         results.push_back(Temp);
        }
        m->RewindAll();
        return results;
}

vector<vector<float> > Top7::PhiPsiTableMPI(int Pos,
                                        int Rot,
                                        float Degree,
                                          float Precision)
{
        m->SaveAll();
        vector<vector<float> > results;
        vector<float> Temp;
        if(Rot>=m->numRots(Pos)) throw moleculesError_badRotNum(posID(Pos),Rot);
        int NumAngles = ceil(Degree/Precision);
        float Phi, Psi;
        float Energy;
        float CurrentPhi = m->GetPhi(Pos);
        float CurrentPsi = m->GetPsi(Pos);
        int CurrentRot = m->getActive(Pos);
        float DeltaPhi = Precision;
        float DeltaPsi = Precision;
        int Rank = MPI_ME;
```

```
          int i = floor(Rank/(2*NumAngles+1));
          int j = Rank % (2*NumAngles+1);

          if(MPI_MASTER)
          {
           OUT << "\nCurrentPhi: " << CurrentPhi;
           OUT << "\nCurrentPsi: " << CurrentPsi;
           OUT << endl;
           OUT << "\nRot: " << Rot << endl;
          }

          m->setActive(Pos,Rot);
          Phi = CurrentPhi+(i-NumAngles)*DeltaPhi;
          Psi = CurrentPsi+(j-NumAngles)*DeltaPsi;
          m->SetPhi(Pos,Phi);
          m->SetPsi(Pos,Psi);
          Energy = energyFlex(*m,*f);
          if(!MPI_MASTER)
          {
           ostringstream s;
           s << Energy;
           mpiSend(MPI_MASTER,s.str());
          }
          else
          {

           Temp.push_back(Energy);
           int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
           for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
           {
             string data;
             mpiRecv(_Rank,data);
             float EnergyGuest=atof(data.c_str());
             if(_Rank%(2*NumAngles+1)==0)
             {
               results.push_back(Temp);
               Temp.clear();
             }
             Temp.push_back(EnergyGuest);
           }
           results.push_back(Temp);
           Temp.clear();
          }

          m->RewindAll();
          return results;
}

/*
map<string,float> Top7::BestPhiPsiPos(int Pos, float Degree, float Precision)
{
          map<string,float> result;
          float Min=INFTY;
          float CurrentPhi = m->GetPhi(Pos);
          float CurrentPsi = m->GetPsi(Pos);
          float DeltaPhi;
          float DeltaPsi;
          int Rot=-1;
          int NumAngles = ceil(Degree/Precision);
          for(int rot=0;rot<m->numRots(Pos);rot++)
          {
           OUT << "\n-----"<<int(float(rot)/float(m->numRots(Pos))*100)<<"%";
           OUT << " completed.... --------"<<endl;
           vector<vector<float> > results = PhiPsiTable(Pos,rot,Degree,Precision);
           for(int i=0;i<results.size();i++)
           {
             for(int j=0;j<results.size();j++)
             {
               if(results[i][j]<Min)
               {
                 Min = results[i][j];
                 DeltaPhi=(i-NumAngles)*Precision;
                 DeltaPsi=(j-NumAngles)*Precision;
                 Rot = rot;
               }
             }
           }
           OUT <<"\nMin: " << Min;
           OUT.flush();
          }
          result["Min"]=Min;
          result["CurrentPhi"]=CurrentPhi;
          result["CurrentPsi"]=CurrentPsi;
          result["DeltaPhi"]=DeltaPhi;
          result["DeltaPsi"]=DeltaPsi;
          result["Rot"]=Rot;
          return result;
}*/
//We are rewriting it to have a high performance one....


map<string,float> Top7::BestPhiPsiPos(int Pos,
                                      float Degree,
                                      float Precision,
                                      bool RotamerOptimization)
{
          map<string,float> result;
          float Min=INFTY;
          float CurrentPhi = m->GetPhi(Pos);
          float CurrentPsi = m->GetPsi(Pos);
          int CurrentRot = m->getActive(Pos);
          int NumAngles = ceil(Degree/Precision);
          int NumRots = m->numRots(Pos);
          if(!RotamerOptimization)
           NumRots=1;
          float Phi, Psi;
          float Energy;
```

```
        float DPhi = Precision;
        float DPsi = Precision;
        float DeltaPhi;
        float DeltaPsi;
        int Rot = 0;
        vector<vector<vector<float> > > table3D;

        m->SaveAll();
//      OUT << "\nCurrentPhi: " << CurrentPhi;
//      OUT << "\nCurrentPsi: " << CurrentPsi;
//      OUT << endl;

//      m->setActive(Pos,0);
        for(int i=0; i<2*NumAngles+1;i++)
        {
         Phi = CurrentPhi+(i-NumAngles)*DPhi;
//       cout << "\ni: " << i << endl;
//       cout << "\nNumAngles: " << NumAngles << endl;
//       cout << "\nDeltaPhi: " << DPhi << endl;
//       cout << "\nDph: " << Phi - CurrentPhi << endl;
         vector<vector<float> > RowRot;
         for(int j=0;j<2*NumAngles+1;j++)
         {
           Psi = CurrentPsi+(j-NumAngles)*DPsi;
           m->SetPhi(Pos,Phi);
           m->SetPsi(Pos,Psi);
           if(RotamerOptimization)
             //m->ReAlignRotamers(Pos);
             m->ReAlignRotamers(); //June,29,2010 - I changed to only realign pos
          vector<float> RotTemp;
          for(int rot=0;rot<NumRots;rot++)
          {
            m->setActive(Pos,rot);
            Energy = energyFlex(*m,*f);
//          OUT << "DPh: " << Phi-CurrentPhi;
//          OUT <<", DPs: " << Psi-CurrentPsi;
//          OUT << ", rot: "<<rot<<", Eng: "<<Energy<<endl;
            RotTemp.push_back(Energy);
          }
          RowRot.push_back(RotTemp);
         }
         table3D.push_back(RowRot);

         for(int i=0;i<table3D.size();i++)
         {
           for(int j=0;j<table3D.size();j++)
           {
             for(int rot=0;rot<NumRots;rot++)
             {
               if(table3D[i][j][rot]<Min)
               {
                 Min = table3D[i][j][rot];
                 DeltaPhi=(i-NumAngles)*Precision;
                 DeltaPsi=(j-NumAngles)*Precision;
                 Rot = rot;
               }
             }
           }
         }
//       OUT <<"\nMin: " << Min << endl;
        }
        result["Min"]=Min;
        result["CurrentPhi"]=CurrentPhi;
        result["CurrentPsi"]=CurrentPsi;
        result["DeltaPhi"]=DeltaPhi;
        result["DeltaPsi"]=DeltaPsi;
        result["Rot"]=Rot;
        m->RewindAll();
        return result;
}

map<string,float> Top7::BestPhiPsiPosMPI(int Pos, float Degree, float Precision)
{
        map<string,float> result;
        float Min=INFTY;
        float CurrentPhi = m->GetPhi(Pos);
        float CurrentPsi = m->GetPsi(Pos);
        float DeltaPhi;
        float DeltaPsi;
        int Rot=-1;
        int NumAngles = ceil(Degree/Precision);
        for(int rot=0;rot<m->numRots(Pos);rot++)
        {
         OUT << "\n-----"<<int(float(rot)/float(m->numRots(Pos))*100)<<"%";
         OUT << " completed.... --------"<<endl;
         vector<vector<float> > results = PhiPsiTableMPI(Pos,rot,Degree,Precision);
         if(MPI_MASTER)
         {
           for(int i=0;i<results.size();i++)
           {
             for(int j=0;j<results.size();j++)
             {
               if(results[i][j]<Min)
               {
                 Min = results[i][j];
                 DeltaPhi=(i-NumAngles)*Precision;
                 DeltaPsi=(j-NumAngles)*Precision;
                 Rot = rot;
               }
             }
           }
           OUT <<"\nMin: " << Min;
           OUT.flush();
         }
        }
        if(MPI_MASTER)
        {
```

```
          result["Min"]=Min;
          result["CurrentPhi"]=CurrentPhi;
          result["CurrentPsi"]=CurrentPsi;
          result["DeltaPhi"]=DeltaPhi;
          result["DeltaPsi"]=DeltaPsi;
          result["Rot"]=Rot;
         }
         return result;
}

vector<vector<pair<float,int> > > Top7::BestPhiPsiPosMPI_HighPerformance(int Pos,
                                                          int StartPos,
                                                         int StopPos,
                                                          float Degree,
                                                           float Precision,
                                                         bool RotamerOptimization)
{
//NOTE: there was sth  wrong with energyWindow, consider fixing it....
         m->SaveAll();
         vector<vector<pair<float,int> > > results;
         vector<pair<float,int> > Temp;
         int NumAngles = ceil(Degree/Precision);
         int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
         float Phi, Psi;
         float BestPartialEnergy;
         float TempPartialEnergy;
         float Energy;
         int Rot = 0;
         int BestRot=0;
         float CurrentPhi = m->GetPhi(Pos);
         float CurrentPsi = m->GetPsi(Pos);
         int CurrentRot = m->getActive(Pos);
         float DeltaPhi = Precision;
         float DeltaPsi = Precision;
         int Rank = MPI_ME;
         int i = floor(Rank/(2*NumAngles+1));
         int j = Rank % (2*NumAngles+1);

         if(MPI_MASTER)
         {
          OUT << "\nCurrentPhi: " << CurrentPhi;
          OUT << "\nCurrentPsi: " << CurrentPsi;
          OUT << endl;
          OUT << "\nRot: " << Rot << endl;
         }

         Phi = CurrentPhi+(i-NumAngles)*DeltaPhi;
         Psi = CurrentPsi+(j-NumAngles)*DeltaPsi;
         m->SetPhi(Pos,Phi);
         m->SetPsi(Pos,Psi);
//Realign.......
//       m->ReAlignRotamers(Pos);
         m->ReAlignRotamers();
         BestPartialEnergy = INFTY;
         if(RotamerOptimization)
         {
          for(Rot=0;Rot<m->numRots(Pos);Rot++)
          {
            m->setActive(Pos,Rot);
            TempPartialEnergy = energyFlex(*m,*f);//----ADED, FEB05

            //TempPartialEnergy = energyFlexWindow(*m,Pos,StartPos,StopPos,*f);
            if(TempPartialEnergy<BestPartialEnergy)
            {
              BestRot = Rot;
              BestPartialEnergy = TempPartialEnergy;
            }
          }
          m->setActive(Pos,BestRot);
         }
         Energy = energyFlex(*m,*f);///--ADED FEB05
         //Energy = energyFlexWindow(*m,StartPos,StopPos,*f);

         if(!MPI_MASTER && MPI_ME<NumOfThreads)
         {
          ostringstream EnergyS;
          EnergyS << Energy;
          mpiSend(MPI_MASTER,EnergyS.str());
         // OUT << "\nSubmitted to MASTER: " << EnergyS.str() << endl;

          ostringstream BestRotS;
          BestRotS << BestRot;
          mpiSend(MPI_MASTER,BestRotS.str());
         // OUT << "\nSubmitted to MASTER: " << BestRotS.str() << endl;
         }
         else if(MPI_MASTER)
         {
          Temp.push_back(pair<float,int>(Energy,BestRot));
          for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
          {
            string EnergyData;
            mpiRecv(_Rank,EnergyData);
            //OUT << "\nSTRNG: " << EnergyData << endl;
            float EnergyGuest=atof(EnergyData.c_str());
            //OUT << "\nMASTER: Received energy from " << _Rank;
            //OUT << ": " << EnergyGuest << endl;
            string BestRotData;
            mpiRecv(_Rank,BestRotData);
            //OUT << "\nSTRNG: " << BestRotData << endl;
            int BestRotGuest=atoi(BestRotData.c_str());
            //OUT << "\nMASTER: Received BestRotGuest from " << _Rank;
            //OUT << ": " << BestRotGuest << endl;
            if(_Rank%(2*NumAngles+1)==0)
            {
              results.push_back(Temp);
              Temp.clear();
            }
```

```
            Temp.push_back(pair<float,int>(EnergyGuest,BestRotGuest));
           }
           results.push_back(Temp);
           Temp.clear();
          }
          m->RewindAll();
          return results;
}

void Top7::TinyGrid(float Degree)
{
          for(int pos=1;pos<m->numPositions()-1;pos++)
          {
           float CurrentPhi = m->GetPhi(pos);
           float CurrentPsi = m->GetPsi(pos);
           int CurrentRot = m->getActive(pos);
           float DPhi, DPsi;
           vector<pair<float,int> > EngRot;
           for(int i=-1;i<=1;i++)
           {
             DPhi = i*Degree;
             for(int j=-1;j<=1;j++)
             {
               DPsi = j*Degree;
               m->SetPhi(pos,CurrentPhi+DPhi);
               m->SetPsi(pos,CurrentPsi+DPsi);
               m->ReAlignRotamers(pos);
               //m->ReAlignRotamers();
               float Min = INFTY;
               int rotMin = 0;
               for(int rot=0;rot<m->numRots(pos);rot++)
               {
                 m->setActive(pos,rot);
                 float Energy = energyFlex(*m,*f);
                 //OUT << "Eng: " << Energy << endl;
                 //YOU CAN IMPROVE this by using EnergyWIndow
                 if(Energy<Min)
                 {
                   Min = Energy;
                   rotMin = rot;
                 }
               }
               m->setActive(pos,CurrentRot);
               EngRot.push_back(pair<float,int>(Min,rotMin));
             }
           }
           float Min = INFTY;
           int indexMin;
           for(int g=0;g<EngRot.size();g++)
           {
             float Eng = EngRot[g].first;
             if(Eng<Min)
             {
               Min = Eng;
               indexMin = g;
             }
           }
           int i=floor(indexMin/3)-1;
           int j=indexMin%3-1;
           DPhi = i*Degree;
           DPsi = j*Degree;
           m->SetPhi(pos,CurrentPhi+DPhi);
           m->SetPsi(pos,CurrentPsi+DPsi);
           m->setActive(pos,EngRot[indexMin].second);
           m->ReAlignRotamers();
           OUT << "Pos: " << pos << endl;
           OUT << "DPhi: " << DPhi << ", DPsi: " << DPsi << endl;
           OUT << "Rot: " << EngRot[indexMin].second << endl;
           OUT << "Eng:  " << Min << endl;
          }
}

void Top7::TinyGrid3_Move_MPI(int pos1, float deg1,
                              int pos2, float deg2,
                              int pos3, float deg3)
{
//In this implementation we use 27 nodes, breaks down 81 nodes into three for
//each node
/*
p3==0 will take care of p3==0 and p3==4, therefore we have a if statement that
checks if p3>=4 and increments that
p3==0 produces a vector of float that has 6 elements than 3.
*/
          pair<float, float> temp;
          int Rank = MPI_ME;
          int Node = floor(Rank/8);
          int p1 = int(Node/3);
          int p2 = Node % 3;
          int p3 = Rank % 8;
          if(p3>=4) p3++;
          float DPhi1, DPsi1, DPhi2, DPsi2, DPhi3, DPsi3;
          temp = PhiPsiPair(deg1, p1);
          DPhi1 = temp.first;
          DPhi1 = temp.second;
          temp = PhiPsiPair(deg3, p3);
          DPhi3 = temp.first;
          DPsi3 = temp.second;

          float InitPhi1 = m->GetPhi(pos1);
          float InitPsi1 = m->GetPsi(pos1);
          float InitPhi2 = m->GetPhi(pos2);
          float InitPsi2 = m->GetPsi(pos2);
          float InitPhi3 = m->GetPhi(pos3);
          float InitPsi3 = m->GetPsi(pos3);

          m->SetPhi(pos1,InitPhi1+DPhi1);
          m->SetPsi(pos1,InitPsi1+DPsi1);
```

```
m->SetPhi(pos3,InitPhi3+DPhi3);
m->SetPsi(pos3,InitPsi3+DPsi3);

vector<float> Energies;
for(int i=3*p2;i<3*(p2+1);i++)
{
 temp = PhiPsiPair(deg2, p2);
 DPhi2 = temp.first;
 DPhi2 = temp.second;
 m->SetPhi(pos2,InitPhi2+DPhi2);
 m->SetPsi(pos2,InitPsi2+DPsi2);
 float Eng = energyFlex(*m,*f);
 Energies.push_back(Eng);
}

if(p3==0)
{
 temp = PhiPsiPair(deg3, 4);
 DPhi3 = temp.first;
 DPsi3 = temp.second;
 m->SetPhi(pos3,InitPhi3+DPhi3);
 m->SetPsi(pos3,InitPsi3+DPsi3);
 for(int i=3*p2;i<3*(p2+1);i++)
 {
   temp = PhiPsiPair(deg2, p2);
   DPhi2 = temp.first;
   DPhi2 = temp.second;
   m->SetPhi(pos2,InitPhi2+DPhi2);
   m->SetPsi(pos2,InitPhi2+DPsi2);
   float Eng = energyFlex(*m,*f);
   Energies.push_back(Eng);
 }
}
if(MPI_MASTER)
{
 vector<float> TempEng;
 vector<vector<float> > AllEng;
 for(int _Rank=1;_Rank<MPI_SIZE;_Rank++)
 {
   mpiRecv(_Rank,TempEng);
   AllEng.push_back(TempEng);
 }
 float MinEng = INFTY;
 int mpirank;
 int Index;
 for(int i=0;i<AllEng.size();i++)
 {
   vector<float> temp = AllEng[i];
   for(int j=0;j<temp.size();j++)
   {
     if(temp[j]<MinEng)
     {
       MinEng = temp[j];
       mpirank = i;
       Index = j;
     }
   }
 }
 int _Node = floor(mpirank/8);
 int _p1 = int(_Node/3);
 int _p2 = _Node % 3;
 int _p3 = mpirank % 8;
 if(_p3>=4) _p3++;

 if(_p3==0 && Index>2)
 {
   _p3=4;
   Index = Index%3;
 }

 temp = PhiPsiPair(deg1,_p1);
 float _DPhi1 = temp.first;
 float _DPsi1 = temp.second;

 temp = PhiPsiPair(deg2,3*_p2+Index);
 float _DPhi2 = temp.first;
 float _DPsi2 = temp.second;

 temp = PhiPsiPair(deg3,_p3);
 float _DPhi3 = temp.first;
 float _DPsi3 = temp.second;

 vector<float> Angles;
 Angles.push_back(InitPhi1+_DPhi1);
 Angles.push_back(InitPsi1+_DPsi1);
 Angles.push_back(InitPhi2+_DPhi2);
 Angles.push_back(InitPsi2+_DPsi2);
 Angles.push_back(InitPhi3+_DPhi3);
 Angles.push_back(InitPsi3+_DPsi3);

 m->SetPhi(pos1,Angles[0]);
 m->SetPsi(pos1,Angles[1]);
 m->SetPhi(pos2,Angles[2]);
 m->SetPsi(pos2,Angles[3]);
 m->SetPhi(pos3,Angles[4]);
 m->SetPsi(pos3,Angles[5]);

 for(int _Rank=1;_Rank<MPI_SIZE;_Rank++)
 {
   mpiSend(_Rank,Angles);
 }
}
else
{
 mpiSend(MPI_MASTER,Energies);
 vector<float> Angles;
 mpiRecv(MPI_MASTER,Angles);
```

```
          m->SetPhi(pos1,Angles[0]);
          m->SetPsi(pos1,Angles[1]);
          m->SetPhi(pos2,Angles[2]);
          m->SetPsi(pos2,Angles[3]);
          m->SetPhi(pos3,Angles[4]);
          m->SetPsi(pos3,Angles[5]);
          }
          m->ReAlignRotamers();
}

pair<float, float> Top7::PhiPsiPair(float deg, int offset)
{
          float DPhi, DPsi;
          switch(offset)
          {
           case 0:
             DPhi = -deg;
             DPsi = deg;
             break;
           case 1:
             DPhi = 0;
             DPsi = deg;
             break;
           case 2:
             DPhi = deg;
             DPsi = deg;
             break;
           case 3:
             DPhi = -deg;
             DPsi = 0;
             break;
           case 4:
             DPhi = 0;
             DPsi = 0;
             break;
           case 5:
             DPhi = deg;
             DPsi = 0;
             break;
           case 6:
             DPhi = -deg;
             DPsi = -deg;
             break;
           case 7:
             DPhi = 0;
             DPsi = -deg;
             break;
           case 8:
             DPhi = deg;
             DPsi = -deg;
             break;
          }
          return pair<float,float>(DPhi,DPsi);
}

void Top7::GradientBestPhiPsi(float Degree,
                              float Precision,
                              bool RotamerOptimization,
                              int RecordSnapShots)
{
          for(int Pos=1;Pos<m->numPositions()-1;Pos++)
          {
           map<string,float> result;
           result = BestPhiPsiPos(Pos,Degree,Precision,RotamerOptimization);
           OUT << "\nPos: " << Pos << endl;
           OUT << "\nDeg: " << Degree << endl;
           OUT << "\nPrecision: " << Precision << endl;
           OUT << "\nMin: " << result["Min"] << endl;
           OUT << "\nCurrentPhi: " << result["CurrentPhi"] << endl;
           OUT << "\nCurrentPsi: " << result["CurrentPsi"] << endl;
           OUT << "\nDeltaPhi: " << result["DeltaPhi"] << endl;
           OUT << "\nDeltaPsi: " << result["DeltaPsi"] << endl;
           OUT << "\nRot: " << int(result["Rot"]) << endl;
           float CurrentPhi = m->GetPhi(Pos);
           float CurrentPsi = m->GetPsi(Pos);
           m->SetPhi(Pos,CurrentPhi+result["DeltaPhi"]);
           m->SetPsi(Pos,CurrentPsi+result["DeltaPsi"]);
           OUT << "\nNow while we measure current Phi and Psi: ";
           OUT << "CrrtPhi: " << CurrentPhi << "\tCrrtPsi: " << CurrentPsi << endl;
           if(RotamerOptimization)
             m->ReAlignRotamers();
           m->setActive(Pos,int(result["Rot"]));
           OUT << "\nPhi and Psi after apply Deltas: ";
           OUT << "NewPhi: " << m->GetPhi(Pos) << "\tNewPsi: " << m->GetPsi(Pos) << endl;
           OUT << "\nEnergy= " << energyFlex(*m,*f) << endl;
           if(RecordSnapShots)
           {
             SnapShotRecorder(RecordSnapShots,Pos);
           }
           bool InPlace = true;
           m->SaveAll(InPlace);
          }
}

void Top7::GradientBestPhiPsiPosMPI(int Pos,
                                    float Degree,
                                    float Precision)

{
          int NumAngles = ceil(Degree/Precision);
          int Threads = (2*NumAngles+1)*(2*NumAngles+1);
          map<string,float> result;
          result = BestPhiPsiPosMPI(Pos,Degree,Precision);
          OUT << "\nMin: " << result["Min"] << endl;
          OUT << "\nCurrentPhi: " << result["CurrentPhi"] << endl;
          OUT << "\nCurrentPsi: " << result["CurrentPsi"] << endl;
```

```
          OUT << "\nDeltaPhi: " << result["DeltaPhi"] << endl;
          OUT << "\nDeltaPsi: " << result["DeltaPsi"] << endl;
          OUT << "\nRot: " << int(result["Rot"]) << endl;
          float CurrentPhi = m->GetPhi(Pos);
          float CurrentPsi = m->GetPsi(Pos);
          OUT << "\nNow while we measure current Phi and Psi: ";
          OUT << "CrrtPhi: " << CurrentPhi << "\tCrrtPsi: " << CurrentPsi << endl;
          if(MPI_MASTER)
          {
           vector<float> RotPhiPsiResults;
           RotPhiPsiResults.push_back(result["Rot"]);
           RotPhiPsiResults.push_back(CurrentPhi+result["DeltaPhi"]);
           RotPhiPsiResults.push_back(CurrentPsi+result["DeltaPsi"]);

           for(int _Rank=1;_Rank<Threads;_Rank++)
           {
             mpiSend(_Rank,RotPhiPsiResults);
           }
           //m->setActive(Pos,int(result["Rot"]));
           m->SetPhi(Pos,CurrentPhi+result["DeltaPhi"]);
           m->SetPsi(Pos,CurrentPsi+result["DeltaPsi"]);
           m->setActive(Pos,int(result["Rot"]));
          }
          else
          {
           vector<float> RotPhiPsiResults;
           mpiRecv(MPI_MASTER,RotPhiPsiResults);
           OUT << "\n"<<MPI_ME<<": ";
           OUT.flush();
           int Rot = int(RotPhiPsiResults[0]);
           float Phi = RotPhiPsiResults[1];
           float Psi = RotPhiPsiResults[2];
           OUT << "Rot=" << Rot <<", Phi= "<<Phi<<", Psi= "<<Psi<<endl;
           //m->setActive(Pos,Rot);
           m->SetPhi(Pos,Phi);
           m->SetPsi(Pos,Psi);
           m->setActive(Pos,Rot);
          }
          m->ReAlignRotamers();
          OUT << "\nPhi and Psi after apply Deltas: ";
          OUT << "NewPhi: " << m->GetPhi(Pos) << "\tNewPsi: " << m->GetPsi(Pos) << endl;
          OUT << "\nEnergy= " << energyFlex(*m,*f) << endl;
}

void Top7::GradientBestPhiPsiMPI(float Degree,
                           float Precision,
                           bool ExcludeAlphaHelices)
{
          int NumAngles = ceil(Degree/Precision);
          int Threads = (2*NumAngles+1)*(2*NumAngles+1);
          for(int Pos=1;Pos<m->numPositions()-1;Pos++)
          {
           if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
           GradientBestPhiPsiPosMPI(Pos,Degree,Precision);
          }
}

void Top7::GradientBestPhiPsiMPI_HighPerformance(int StartPos,
                                      int StopPos,
                                      float Degree,
                                      float Precision,
                                      bool ExcludeAlphaHelices,
                                      int RecordSnapShots)
{
          int NumAngles = ceil(Degree/Precision);
          int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
          vector<vector<pair<float,int> > > results;
          for(int Pos=StartPos+1;Pos<StopPos-1;Pos++)
          {
           map<string,float> result;
           float Min=INFTY;
           float CurrentPhi = m->GetPhi(Pos);
           float CurrentPsi = m->GetPsi(Pos);
           float DeltaPhi;
           float DeltaPsi;
           int Rot=-1;
           vector<float> PhiPsiRot;
           PhiPsiRot.reserve(3);

           if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
           results =
           BestPhiPsiPosMPI_HighPerformance(Pos,StartPos,StopPos,Degree,Precision);
           if(MPI_MASTER)
           {
             for(int i=0;i<results.size();i++)
               for(int j=0;j<results.size();j++)
               {
                 if(results[i][j].first<Min)
                 {
                   Min = results[i][j].first;
                   DeltaPhi=(i-NumAngles)*Precision;
                   DeltaPsi=(j-NumAngles)*Precision;
                   Rot = results[i][j].second;
                 }
               }

             OUT << "\nPos: " << Pos << endl;
             OUT << "\nMin Energy: " << Min << endl;
             OUT << "\nDeltaPhi: " << DeltaPhi << endl;
             OUT << "\nDeltaPsi: " << DeltaPsi << endl;
             OUT << "\nRot: " << Rot << endl;
           /*
             result["Min"]=Min;
             result["CurrentPhi"]=CurrentPhi;
             result["CurrentPsi"]=CurrentPsi;
             result["DeltaPhi"]=DeltaPhi;
             result["DeltaPsi"]=DeltaPsi;
```

```
            result["Rot"]=Rot;*/
            //OUT << "\n We reached to the point to send back the stuff...." << endl;
            for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
            {
              PhiPsiRot.clear();
              PhiPsiRot.push_back(CurrentPhi+DeltaPhi);
              PhiPsiRot.push_back(CurrentPsi+DeltaPsi);
              PhiPsiRot.push_back(float(Rot)+0.001);
              mpiSend(_Rank,PhiPsiRot);
              //OUT << "\nSent to: " << _Rank << endl;
            }
//          if(!(DeltaPhi==0 && DeltaPsi==0))
//          {
            m->SetPhi(Pos,CurrentPhi+DeltaPhi);
            m->SetPsi(Pos,CurrentPsi+DeltaPsi);
            m->ReAlignRotamers();
            m->setActive(Pos,Rot);
//          }
          }
          else
          {
            mpiRecv(MPI_MASTER,PhiPsiRot);
            //OUT << "\nRecvd from MASTER" << endl;
//          if(!(CurrentPhi==PhiPsiRot[0] && CurrentPsi==PhiPsiRot[1]))
//          {
            m->SetPhi(Pos,PhiPsiRot[0]);
            m->SetPsi(Pos,PhiPsiRot[1]);
            m->ReAlignRotamers();
            m->setActive(Pos,int(PhiPsiRot[2]));
//          }
          }
          if(RecordSnapShots)
          {
            SnapShotRecorder(RecordSnapShots,Pos);
          }
         }
}

void Top7::GradientBestPhiPsiBackrubMPI_HighPerformance(int StartPos,
                                          int StopPos,
                                          float Degree,
                                          float Precision,
                                          bool ExcludeAlphaHelices,
                                          int BackrubLength,
                                          float BackrubDegree,
                                          float BackrubPrecision)
{
//        OUT << "\nEntered PsiBackrub..... " << endl;
          int NumAngles = ceil(Degree/Precision);
//        OUT << "\nNumAngles:" << NumAngles << endl;
          int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
          OUT << "\nNumThreads:" << NumOfThreads << endl;
          vector<vector<pair<float,int> > > results;
          int Rank = MPI_ME;
//        OUT << "\nRank:" << Rank << endl;
          int NumOfBackrubProcesses=2*int(floor(BackrubDegree/BackrubPrecision+0.5))+1;

          OUT << "\nNumOfBackrubProcesses:" << NumOfBackrubProcesses << endl;
          float Theta = BackrubPrecision*Rank-BackrubDegree;
//        OUT << "\nTheta:" << Theta << endl;
          float OptimalTheta;
//        OUT << "\nStart: " << StartPos << endl;
//        OUT << "\nStop: " << StopPos << endl;
          m->SaveAll();
          for(int Pos=StartPos+1;Pos<=StopPos-1;Pos++)
          {
//         OUT << "\nPos: " << Pos << endl;
           float BackrubEnergy;
           if(Pos+3>=StopPos) continue;
           vector<int> Positions;
           Positions.push_back(Pos);
           Positions.push_back(Pos+1);
           Positions.push_back(Pos+2);
           Positions.push_back(Pos+3);

           m->SaveInPlace(Positions);

//         if(Rank>NumOfBackrubProcesses)
//           synchronize();
//         else
//         OUT << "\nPos: " << Pos << endl;
           if(Rank<NumOfBackrubProcesses)
           {
             BackrubMove(Pos,Pos+3,Theta);
             BackrubEnergy=energyFlex(*m,*f);
//           OUT << "\nBackrubEnergy: " << BackrubEnergy << endl;
             if(MPI_MASTER)
             {
               float MinEng=BackrubEnergy;
               int MinRankIndex=0;
               float TempEng;

               for(int _Rank=1;_Rank<NumOfBackrubProcesses;_Rank++)
               {
                 mpiRecv(_Rank,TempEng);
//               OUT << "\nFrom" << _Rank << " Recvd: " << TempEng << endl;
                 if(TempEng<MinEng)
                 {
                   MinEng = TempEng;
                   MinRankIndex=_Rank;
                 }
               }
               OptimalTheta = BackrubPrecision*MinRankIndex-BackrubDegree;
               OUT << "\nMinEng: " << MinEng << ", MinRank: " << MinRankIndex<<endl;
               for(int _Rank=1;_Rank<NumOfBackrubProcesses;_Rank++)
               {
                 mpiSend(_Rank,OptimalTheta);
```

```
            }
          }
          else
          {
            mpiSend(MPI_MASTER,BackrubEnergy);
            mpiRecv(MPI_MASTER,OptimalTheta);
//          OUT << "\nOptimalTheta recvd: " << OptimalTheta << endl;
          }
          m->RewindInPlace(Positions);
          BackrubMove(Pos,Pos+3,OptimalTheta);
          m->ReAlignRotamers(Positions);
        }
        //synchronize();

//      OUT << "\nWe got out of that backrub thing.... " << endl;
        map<string,float> result;
        float Min=INFTY;
        float CurrentPhi = m->GetPhi(Pos);
        float CurrentPsi = m->GetPsi(Pos);
        float DeltaPhi;
        float DeltaPsi;
        int Rot=-1;
        vector<float> PhiPsiRot;
        PhiPsiRot.reserve(3);

        if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
//      OUT << "\nBefore BestPhiPsi.... " << endl;
        results = BestPhiPsiPosMPI_HighPerformance(Pos,StartPos,StopPos,Degree,Precision);
//      OUT << "\nAfter BestPhiPsi.... " << endl;
        if(MPI_MASTER)
        {
          for(int i=0;i<results.size();i++)
            for(int j=0;j<results.size();j++)
            {
              if(results[i][j].first<Min)
              {
                Min = results[i][j].first;
                DeltaPhi=(i-NumAngles)*Precision;
                DeltaPsi=(j-NumAngles)*Precision;
                Rot = results[i][j].second;
              }
            }

          OUT << "\nPos: " << Pos << endl;
          OUT << "\nMin Energy: " << Min << endl;
          OUT << "\nDeltaPhi: " << DeltaPhi << endl;
          OUT << "\nDeltaPsi: " << DeltaPsi << endl;
          OUT << "\nRot: " << Rot << endl;
        /*
          result["Min"]=Min;
          result["CurrentPhi"]=CurrentPhi;
          result["CurrentPsi"]=CurrentPsi;
          result["DeltaPhi"]=DeltaPhi;
          result["DeltaPsi"]=DeltaPsi;
          result["Rot"]=Rot;*/
          //OUT << "\n We reached to the point to send back the stuff...." << endl;
          for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
          {
            PhiPsiRot.clear();
            PhiPsiRot.push_back(CurrentPhi+DeltaPhi);
            PhiPsiRot.push_back(CurrentPsi+DeltaPsi);
            PhiPsiRot.push_back(float(Rot)+0.001);
            mpiSend(_Rank,PhiPsiRot);
            //OUT << "\nSent to: " << _Rank << endl;
          }
//        if(!(DeltaPhi==0 && DeltaPsi==0))
//        {
          m->SetPhi(Pos,CurrentPhi+DeltaPhi);
          m->SetPsi(Pos,CurrentPsi+DeltaPsi);
          m->ReAlignRotamers();
          m->setActive(Pos,Rot);
//        }
        }
        else
        {
          mpiRecv(MPI_MASTER,PhiPsiRot);
          //OUT << "\nRecvd from MASTER" << endl;
//        if(!(CurrentPhi==PhiPsiRot[0] && CurrentPsi==PhiPsiRot[1]))
//        {
          m->SetPhi(Pos,PhiPsiRot[0]);
          m->SetPsi(Pos,PhiPsiRot[1]);
          m->ReAlignRotamers();
          m->setActive(Pos,int(PhiPsiRot[2]));
//        }
        }
       }
}


void Top7::AllAtOnceGradientMPI_HighPerformance(int StartPos,
                                    int StopPos,
                                    float Degree,
                                    float Precision,
                                    bool ExcludeAlphaHelices,
                                    bool RotamerOptimization)
{
        int NumAngles = ceil(Degree/Precision);
        int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
        vector<vector<pair<float,int> > > results;
        vector<vector<float> > PhiPsiRotList;
        for(int Pos=StartPos;Pos<StopPos;Pos++)
        {
         map<string,float> result;
         float Min=INFTY;
         float CurrentPhi = m->GetPhi(Pos);
         float CurrentPsi = m->GetPsi(Pos);
         float DeltaPhi;
```

```
          float DeltaPsi;
          int Rot=-1;
          vector<float> PhiPsiRot;
          PhiPsiRot.reserve(3);

          if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
          results = BestPhiPsiPosMPI_HighPerformance(Pos,StartPos,StopPos,Degree,Precision);
          if(MPI_MASTER)
          {
            for(int i=0;i<results.size();i++)
              for(int j=0;j<results.size();j++)
              {
                if(results[i][j].first<Min)
                {
                  Min = results[i][j].first;
                  DeltaPhi=(i-NumAngles)*Precision;
                  DeltaPsi=(j-NumAngles)*Precision;
                  Rot = results[i][j].second;
                }
              }

            OUT << "\nPos: " << Pos << endl;
            OUT << "\nMin Energy: " << Min << endl;
            OUT << "\nDeltaPhi: " << DeltaPhi << endl;
            OUT << "\nDeltaPsi: " << DeltaPsi << endl;
            OUT << "\nRot: " << Rot << endl;

            PhiPsiRot.clear();
            PhiPsiRot.push_back(CurrentPhi+DeltaPhi);
            PhiPsiRot.push_back(CurrentPsi+DeltaPsi);
            PhiPsiRot.push_back(float(Rot)+0.001);

            PhiPsiRotList.push_back(PhiPsiRot);
            for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
            {
              mpiSend(_Rank,PhiPsiRot);
            }
          }
          else
          {
            mpiRecv(MPI_MASTER,PhiPsiRot);
            PhiPsiRotList.push_back(PhiPsiRot);
          }
          }
          OUT << "\nWe finished... the calculation......."<<endl;
          for(int Pos=StartPos;Pos<StopPos;Pos++)
          {
          m->SetPhi(Pos,PhiPsiRotList[Pos-StartPos][0]);
          m->SetPsi(Pos,PhiPsiRotList[Pos-StartPos][1]);
          m->ReAlignRotamers();
          m->setActive(Pos,int(PhiPsiRotList[Pos-StartPos][2]));
          }
}

void Top7::AllAtOnceGradientMPI_56Nodes(float Degree,
                                        int Steps,
                                        bool ExcludeAlphaHelices)
{
//This function is supposed to be working for proteinG. If it turns that it is
//useful we can generalize it to suire for other protein structures as well....
//It isd meant to be working simultanously making a grid of 3x3 and all th
//eresidus get calculated all at once....
          OUT << "\n56Node Started..." << endl;
          for(int step=0; step < Steps; step++)
          {
          float InitialEnergy = energyFlex(*m,*f);
          int Rank = MPI_ME;
          int Pos = floor(Rank/8);
          int Order = Rank-Pos*8;
          float DPhi, DPsi;
          switch(Order)
          {
            case 0:
              DPhi = -Degree;
              DPsi = Degree;
            break;
            case 1:
              DPhi = 0;
              DPsi = Degree;
            break;
            case 2:
              DPhi = Degree;
              DPsi = Degree;
            break;
            case 3:
              DPhi = -Degree;
              DPsi = 0;
            break;
            case 4:
              DPhi = Degree;
              DPsi = 0;
            break;
            case 5:
              DPhi = -Degree;
              DPsi = -Degree;
            break;
            case 6:
              DPhi = 0;
              DPsi = -Degree;
            break;
            case 7:
              DPhi = Degree;
              DPsi = -Degree;
            break;
            default:
              OUT <<"\nError: Order out of range! (AllAtOnceGradientMPI_56Nodes) ";
              OUT << Order << endl;
```

```
  break;
}
if(!MPI_MASTER)
{
  m->perturbPhiAngle(Pos,DPhi,false);
  m->perturbPsiAngle(Pos,DPsi,false);
  m->ReAlignRotamers(Pos);
}
int optRot=-1;
float minEng = INFTY;
for(int rot=0;rot<m->numRots(Pos);rot++)
{
  float Eng = energyFlex(*m,*f);
  if(Eng<minEng)
  {
    optRot = rot;
    minEng = Eng;
  }
}
vector<float> eng;
vector<int> rot;
vector<pair<float,float> > PhiPsiPerturbations;
vector<int> RotPerturbations;
if(MPI_MASTER)
{
  vector<float> optEnergies;
  vector<int> optRotamers;
  for(int Pos=0;Pos<56;Pos++)
  {
    int optOrder=-1;
    float minOrderEng = InitialEnergy;
    int optRotamer=-1;
    float dPhi, dPsi;
    if(!(ExcludeAlphaHelices && IsInAlphaHelix(Pos)))
    {
      for(int Order=0;Order<8;Order++)
      {
        int Rank = Pos*8+Order;
        if(Rank==0) continue;
        mpiRecv(Rank,eng);
        mpiRecv(Rank,rot);
        float orderEng = eng[0];
        int orderRot = rot[0];
        if(orderEng<minOrderEng)
        {
          minOrderEng = orderEng;
          optRotamer = orderRot;
          optOrder = Order;
        }
      }
      switch(optOrder)
      {
        case 0:
          dPhi = -Degree;
          dPsi = Degree;
        break;
        case 1:
          dPhi = 0;
          dPsi = Degree;
        break;
        case 2:
          dPhi = Degree;
          dPsi = Degree;
        break;
        case 3:
          dPhi = -Degree;
          dPsi = 0;
        break;
        case 4:
          dPhi = Degree;
          dPsi = 0;
        break;
        case 5:
          dPhi = -Degree;
          dPsi = -Degree;
        break;
        case 6:
          dPhi = 0;
          dPsi = -Degree;
        break;
        case 7:
          dPhi = Degree;
          dPsi = -Degree;
        break;
        default://This means that dphi=0, dpsi=0 is the best
          dPhi = 0;
          dPsi = 0;
        break;
      }
    }
    else
    {
      dPhi = 0;
      dPsi = 0;
    }
    PhiPsiPerturbations.push_back(pair<float,float>(dPhi,dPsi));
    RotPerturbations.push_back(optRotamer);
  }
  for(int rank=1;rank<56*8;rank++)
  {
    mpiSend(rank,PhiPsiPerturbations);
    mpiSend(rank,RotPerturbations);
  }
}
else
{
  eng.push_back(minEng);
```

```
            rot.push_back(optRot);
            mpiSend(MPI_MASTER,eng);
            mpiSend(MPI_MASTER,rot);
            mpiRecv(MPI_MASTER,PhiPsiPerturbations);
            mpiRecv(MPI_MASTER,RotPerturbations);
          }

          for(int Pos=1;Pos<56;Pos++)
          {
            float dPhi = PhiPsiPerturbations[Pos].first;
            float dPsi = PhiPsiPerturbations[Pos].second;
            if(!(dPhi==0 && dPsi==0))
            {
              m->perturbPhiAngle(Pos,dPhi,false);
              m->perturbPhiAngle(Pos,dPsi,false);
            }
          }
          m->ReAlignRotamers();
          for(int Pos=1;Pos<56;Pos++)
          {
            float dPhi = PhiPsiPerturbations[Pos].first;
            float dPsi = PhiPsiPerturbations[Pos].second;
            if(!(dPhi==0 && dPsi==0))
            {
              m->setActive(Pos,RotPerturbations[Pos]);
            }
          }
         }
         OUT << "\n56Node Ended..." << endl;
}

void Top7::AllAtOnceGradientMPI_xNodes(int numberOfNodes,
                                       vector<float> Degrees,
                                       bool ExcludeAlphaHelices)
{
/*
The assumption here in this functon is that we want to have an efficient method
of AllAtOnce with 9 grid points. Each residue will be take care of by each node,
and then based on the number of nodes available, they would be calculated and
sent back to the head node.
*/
         OUT << "Entered AllAtOnceGradintMPI_xNodes" << endl;
         if(m->numPositions()!=Degrees.size())
         {
          OUT << "\nError: Number of configurations doesn't match number of residues...";
          OUT << endl;
         }
         else
         {
          vector<pair<float,float> > CollectedResults;
          int rounds = ceil(m->numPositions()*1.0/numberOfNodes);
          for(int i=0;i<rounds;i++)
          {
            int remainder = m->numPositions()-(i*numberOfNodes);
            int Offset = i*numberOfNodes;
            OUT << "\nOffset: " << Offset << endl;
            int localRank = MPI_ME % 8;
            OUT << "\nlocalRank: " << localRank << endl;
            int nodeNumber = MPI_ME/8;
            OUT << "\nnodeNumber: " << nodeNumber << endl;
            int residueNumber = nodeNumber + Offset; //[0,n-1]
            OUT << "\nresidueNumber: " << residueNumber << endl;

            OUT << "\nremainder: " << remainder << endl;
            if(nodeNumber>remainder)
              break;

            float Phi, Psi;
            float dPhi, dPsi;
            float theta = Degrees[residueNumber];
            OUT << "\ntheta: " << theta << endl;
            Phi = m->GetPhi(residueNumber);
            Psi = m->GetPsi(residueNumber);
            float Energy;

            if(localRank==0)
            {
              vector<float> Energies;
              float EnergyInit = energyFlex(*m,*f);
              dPhi = -theta;
              dPsi = theta;
              m->SetPhi(residueNumber,Phi+dPhi);
              m->SetPsi(residueNumber,Psi+dPsi);
              Energy = energyFlex(*m,*f);
              Energies.push_back(Energy);
              //measure the energy and then immediately return it bacl
              m->SetPhi(residueNumber,Phi);
              m->SetPsi(residueNumber,Psi);
              for(int j=1;j<8;j++)
              {
                float tempEnergy;
                mpiRecv(nodeNumber*8+j,tempEnergy);
                Energies.push_back(tempEnergy);
                if(j==3)
                {
                  Energies.push_back(EnergyInit);
                }
              }
              float MinEnergy = INFTY;
              int Index=-1;
              for(int j=0;j<Energies.size();j++)
              {
                if(MinEnergy>Energies[j])
                {
                  MinEnergy = Energies[j];
                  Index = j;
                }
```

```
    }
    pair<float,float> results;
    results.first = (float(Index)+0.1);
    results.second = MinEnergy;
    if(MPI_MASTER)
    {
      CollectedResults.push_back(results);

      int minNodesRemainder = numberOfNodes;
      if(minNodesRemainder>remainder)
        minNodesRemainder = remainder;
      //this is for the last round...MPI_MASTER doesn't wait for the
      //idle nodes
      for(int j=1;j<minNodesRemainder;j++)
      {
        pair<float,float> tempResults;
        mpiRecv(j*8,tempResults);
        CollectedResults.push_back(tempResults);
      }
    }
    else
    {
      mpiSend(MPI_MASTER,results);
    }
  }
  else
  {
    switch(localRank)
    {
      case 1:
        dPhi = 0;
        dPsi = theta;
        break;
      case 2:
        dPhi = theta;
        dPsi = theta;
        break;
      case 3:
        dPhi = -theta;
        dPsi = 0;
        break;
      case 4:
        dPhi = theta;
        dPsi = 0;
        break;
      case 5:
        dPhi = -theta;
        dPsi = -theta;
        break;
      case 6:
        dPhi = 0;
        dPsi = -theta;
        break;
      case 7:
        dPhi = theta;
        dPsi = -theta;
        break;
    }
    m->SetPhi(residueNumber,Phi+dPhi);
    m->SetPsi(residueNumber,Psi+dPsi);
    Energy = energyFlex(*m,*f);
    //Measure the energy and the rewind back into its original form
    m->SetPhi(residueNumber,Phi);
    m->SetPsi(residueNumber,Psi);
    mpiSend(nodeNumber*8,Energy);
  }
}
/*
for(int pos=0;pos<m->numPositions();pos++)
{
  if(ExcludeAlphaHelices&&IsInAlphaHelix(pos))
  {
    CollectedResults[pos].first = 4; //dPhi=0; dPsi=0;
  }
}*/
if(MPI_MASTER)
{
  for(int rank=1;rank<MPI_SIZE;rank++)
  {
    mpiSend(rank,CollectedResults);
  }
}
else
{
  mpiRecv(MPI_MASTER,CollectedResults);
}
if(CollectedResults.size()!=m->numPositions())
{
  OUT << "\nNumber of CollectedResults does not match number of positions" << endl;
}
for(int pos=0;pos<m->numPositions();pos++)
{
  float Phi, Psi;
  float dPhi, dPsi;
  float theta = Degrees[pos];
  Phi = m->GetPhi(pos);
  Psi = m->GetPsi(pos);
  pair<float, float> temp;
  temp = CollectedResults[pos];
  int tempIndex = int(temp.first);

  switch(tempIndex)
  {
    case 0:
      dPhi = -theta;
      dPsi = theta;
      break;
```

```
                case 1:
                  dPhi = 0;
                  dPsi = theta;
                  break;
                case 2:
                  dPhi = theta;
                  dPsi = theta;
                  break;
                case 3:
                  dPhi = -theta;
                  dPsi = 0;
                  break;
                case 4:
                  dPhi = 0;
                  dPsi = 0;
                  break;
                case 5:
                  dPhi = theta;
                  dPsi = 0;
                  break;
                case 6:
                  dPhi = -theta;
                  dPsi = -theta;
                  break;
                case 7:
                  dPhi = 0;
                  dPsi = -theta;
                  break;
                case 8:
                  dPhi = theta;
                  dPsi = -theta;
                  break;
              }
              m->SetPhi(pos,Phi+dPhi);
              m->SetPsi(pos,Psi+dPsi);
            }
           }
}

void Top7::SomeAtOnceGradientMPI_xNodes(vector<int> Positions,
                                  vector<float> Degrees)
{
/*
The assumption here in this functon is that we want to have an efficient method
of AllAtOnce with 9 grid points. Each residue will be take care of by each node,
and then based on the number of nodes available, they would be calculated and
sent back to the head node.
*/
/*
Number of nodes used for this function should be equal to the number of
Positions
*/
          OUT << "Entered SomeAtOnceGradintMPI_xNodes" << endl;
          if(Positions.size()!=Degrees.size())
           OUT << "Error: number of Positions are not equal the number of Degrees\n"<< endl;
          vector<pair<float,float> > CollectedResults;
          int rounds = 1;
          for(int i=0;i<rounds;i++)
          {
           int localRank = MPI_ME % 8;
           OUT << "\nlocalRank: " << localRank << endl;
           int nodeNumber = MPI_ME/8;
           OUT << "\nnodeNumber: " << nodeNumber << endl;
           int residueNumber = Positions[nodeNumber];
           OUT << "\nresidueNumber: " << residueNumber << endl;

           float Phi, Psi;
           float dPhi, dPsi;
           float theta = Degrees[nodeNumber];
           OUT << "\ntheta: " << theta << endl;
           Phi = m->GetPhi(residueNumber);
           Psi = m->GetPsi(residueNumber);
           float Energy;

           if(localRank==0)
           {
             vector<float> Energies;
             float EnergyInit = energyFlex(*m,*f);
             dPhi = -theta;
             dPsi = theta;
             m->SetPhi(residueNumber,Phi+dPhi);
             m->SetPsi(residueNumber,Psi+dPsi);
             Energy = energyFlex(*m,*f);
             Energies.push_back(Energy);
             //measure the energy and then immediately return it bacl
             m->SetPhi(residueNumber,Phi);
             m->SetPsi(residueNumber,Psi);
             for(int j=1;j<8;j++)
             {
               float tempEnergy;
               mpiRecv(nodeNumber*8+j,tempEnergy);
               Energies.push_back(tempEnergy);
               if(j==3)
               {
                 Energies.push_back(EnergyInit);
               }
             }
             float MinEnergy = INFTY;
             int Index=-1;
             for(int j=0;j<Energies.size();j++)
             {
               if(MinEnergy>Energies[j])
               {
                 MinEnergy = Energies[j];
                 Index = j;
               }
             }
```

```
  pair<float,float> results;
  results.first = (float(Index)+0.1);
  results.second = MinEnergy;
  if(MPI_MASTER)
  {
    CollectedResults.push_back(results);

    /*
    int minNodesRemainder = numberOfNodes;
    if(minNodesRemainder>remainder)
      minNodesRemainder = remainder;*/
    //this is for the last round...MPI_MASTER doesn't wait for the
    //idle nodes
    //for(int j=1;j<minNodesRemainder;j++)
    for(int j=1;j<Positions.size();j++)
    {
      pair<float,float> tempResults;
      mpiRecv(j*8,tempResults);
      CollectedResults.push_back(tempResults);
    }
  }
  else
  {
    mpiSend(MPI_MASTER,results);
  }
}
else
{
  switch(localRank)
  {
    case 1:
      dPhi = 0;
      dPsi = theta;
      break;
    case 2:
      dPhi = theta;
      dPsi = theta;
      break;
    case 3:
      dPhi = -theta;
      dPsi = 0;
      break;
    case 4:
      dPhi = theta;
      dPsi = 0;
      break;
    case 5:
      dPhi = -theta;
      dPsi = -theta;
      break;
    case 6:
      dPhi = 0;
      dPsi = -theta;
      break;
    case 7:
      dPhi = theta;
      dPsi = -theta;
      break;
  }
  m->SetPhi(residueNumber,Phi+dPhi);
  m->SetPsi(residueNumber,Psi+dPsi);
  Energy = energyFlex(*m,*f);
  //Measure the energy and the rewind back into its original form
  m->SetPhi(residueNumber,Phi);
  m->SetPsi(residueNumber,Psi);
  mpiSend(nodeNumber*8,Energy);
}

/*
for(int pos=0;pos<m->numPositions();pos++)
{
  if(ExcludeAlphaHelices&&IsInAlphaHelix(pos))
  {
    CollectedResults[pos].first = 4; //dPhi=0; dPsi=0;
  }
}*/
if(MPI_MASTER)
{
  for(int rank=1;rank<MPI_SIZE;rank++)
  {
    mpiSend(rank,CollectedResults);
  }
}
else
{
  mpiRecv(MPI_MASTER,CollectedResults);
}
for(int j=0;j<Positions.size();j++)
{
  int pos = Positions[j];
  float Phi, Psi;
  float dPhi, dPsi;
  float theta = Degrees[pos];
  Phi = m->GetPhi(pos);
  Psi = m->GetPsi(pos);
  pair<float, float> temp;
  temp = CollectedResults[pos];
  int tempIndex = int(temp.first);

  switch(tempIndex)
  {
    case 0:
      dPhi = -theta;
      dPsi = theta;
      break;
    case 1:
      dPhi = 0;
```

```
                dPsi = theta;
                break;
              case 2:
                dPhi = theta;
                dPsi = theta;
                break;
              case 3:
                dPhi = -theta;
                dPsi = 0;
                break;
              case 4:
                dPhi = 0;
                dPsi = 0;
                break;
              case 5:
                dPhi = theta;
                dPsi = 0;
                break;
              case 6:
                dPhi = -theta;
                dPsi = -theta;
                break;
              case 7:
                dPhi = 0;
                dPsi = -theta;
                break;
              case 8:
                dPhi = theta;
                dPsi = -theta;
                break;
            }
            m->SetPhi(pos,Phi+dPhi);
            m->SetPsi(pos,Psi+dPsi);
          }
         }
}


void Top7::BestResidueGradientMPI_HighPerformance(int StartPos,
                                        int StopPos,
                                        float Degree,
                                        float Precision,
                                        bool ExcludeAlphaHelices)
{
         int NumAngles = ceil(Degree/Precision);
         int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
         vector<vector<pair<float,int> > > results;
         vector<vector<float> > PhiPsiRotList;
         for(int Pos=StartPos;Pos<StopPos;Pos++)
         {
          map<string,float> result;
          float Min=INFTY;
          float CurrentPhi = m->GetPhi(Pos);
          float CurrentPsi = m->GetPsi(Pos);
          float DeltaPhi;
          float DeltaPsi;
          int Rot=-1;
          vector<float> PhiPsiRot;
          PhiPsiRot.reserve(4);

          if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
          results = BestPhiPsiPosMPI_HighPerformance(Pos,StartPos,StopPos,Degree,Precision);
          if(MPI_MASTER)
          {
            for(int i=0;i<results.size();i++)
              for(int j=0;j<results.size();j++)
              {
                if(results[i][j].first<Min)
                {
                  Min = results[i][j].first;
                  DeltaPhi=(i-NumAngles)*Precision;
                  DeltaPsi=(j-NumAngles)*Precision;
                  Rot = results[i][j].second;
                }
              }

            OUT << "\nPos: " << Pos << endl;
            OUT << "\nMin Energy: " << Min << endl;
            OUT << "\nDeltaPhi: " << DeltaPhi << endl;
            OUT << "\nDeltaPsi: " << DeltaPsi << endl;
            OUT << "\nRot: " << Rot << endl;

            PhiPsiRot.clear();
            PhiPsiRot.push_back(CurrentPhi+DeltaPhi);
            PhiPsiRot.push_back(CurrentPsi+DeltaPsi);
            PhiPsiRot.push_back(float(Rot)+0.001);
            PhiPsiRot.push_back(Min);

            PhiPsiRotList.push_back(PhiPsiRot);
            for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
            {
              mpiSend(_Rank,PhiPsiRot);
            }
          }
          else
          {
            mpiRecv(MPI_MASTER,PhiPsiRot);
            PhiPsiRotList.push_back(PhiPsiRot);
          }
         }
         OUT << "\nWe finished... the calculation......."<<endl;
         float MinEnergyResidue=INFTY;
         int PosMinResidue=-1;
         for(int Pos=StartPos;Pos<StopPos;Pos++)
         {
          if(MinEnergyResidue>PhiPsiRotList[Pos-StartPos][3])
          {
```

```
            MinEnergyResidue = PhiPsiRotList[Pos-StartPos][3];
            PosMinResidue = Pos;
          }
         }

         m->SetPhi(PosMinResidue,PhiPsiRotList[PosMinResidue-StartPos][0]);
         m->SetPsi(PosMinResidue,PhiPsiRotList[PosMinResidue-StartPos][1]);
         m->ReAlignRotamers();
         m->setActive(PosMinResidue,int(PhiPsiRotList[PosMinResidue-StartPos][2]));
}


void Top7::SomeAtOnceGradientMPI_HighPerformance(int StartPos,
                                      int StopPos,
                                      float Degree,
                                      float Precision,
                                      bool ExcludeAlphaHelices,
                                      vector<int> Positions)
{
         int NumAngles = ceil(Degree/Precision);
         int NumOfThreads = (2*NumAngles+1)*(2*NumAngles+1);
         vector<vector<pair<float,int> > > results;
         vector<vector<float> > PhiPsiRotList;
         int Pos;
         for(int PosIndex=0;PosIndex<Positions.size();PosIndex++)
         {
          Pos = Positions[PosIndex];
          if(Pos>=StopPos || Pos<StartPos)
          {
            OUT << "\nSomeAtOnce: Out of range positions given..." << endl;
            return;
          }
          map<string,float> result;
          float Min=INFTY;
          float CurrentPhi = m->GetPhi(Pos);
          float CurrentPsi = m->GetPsi(Pos);
          float DeltaPhi;
          float DeltaPsi;
          int Rot=-1;
          vector<float> PhiPsiRot;
          PhiPsiRot.reserve(3);

          if(ExcludeAlphaHelices && IsInAlphaHelix(Pos)) continue;
          results = BestPhiPsiPosMPI_HighPerformance(Pos,StartPos,StopPos,Degree,Precision);
          if(MPI_MASTER)
          {
            for(int i=0;i<results.size();i++)
              for(int j=0;j<results.size();j++)
              {
                if(results[i][j].first<Min)
                {
                  Min = results[i][j].first;
                  DeltaPhi=(i-NumAngles)*Precision;
                  DeltaPsi=(j-NumAngles)*Precision;
                  Rot = results[i][j].second;
                }
              }

            OUT << "\nPos: " << Pos << endl;
            OUT << "\nMin Energy: " << Min << endl;
            OUT << "\nDeltaPhi: " << DeltaPhi << endl;
            OUT << "\nDeltaPsi: " << DeltaPsi << endl;
            OUT << "\nRot: " << Rot << endl;

            PhiPsiRot.clear();
            PhiPsiRot.push_back(CurrentPhi+DeltaPhi);
            PhiPsiRot.push_back(CurrentPsi+DeltaPsi);
            PhiPsiRot.push_back(float(Rot)+0.001);

            PhiPsiRotList.push_back(PhiPsiRot);
            for(int _Rank=1;_Rank<NumOfThreads;_Rank++)
            {
              mpiSend(_Rank,PhiPsiRot);
            }
          }
          else
          {
            mpiRecv(MPI_MASTER,PhiPsiRot);
            PhiPsiRotList.push_back(PhiPsiRot);
          }
         }
         OUT << "\nWe finished... the calculation......."<<endl;
         for(int PosIndex=0;PosIndex<Positions.size();PosIndex++)
         {
          Pos = Positions[PosIndex];
          if(Pos>=StopPos || Pos<StartPos)
          {
            OUT << "\nSomeAtOnce: Out of range positions given..." << endl;
            return;
          }
          m->SetPhi(Pos,PhiPsiRotList[PosIndex][0]);
          m->SetPsi(Pos,PhiPsiRotList[PosIndex][1]);
          m->ReAlignRotamers();
          m->setActive(Pos,int(PhiPsiRotList[PosIndex][2]));
         }
}


void Top7::StochasticGradientBestPhiPsiMPI(int NumOfIterations,
                                int MaxNumOfPositions,
                                float Degree,
                                float Precision,
                                bool ExcludeAlphaHelices)
{
         int NumAngles = ceil(Degree/Precision);
         int Threads = (2*NumAngles+1)*(2*NumAngles+1);
         map<string,float> result;
```

```
          int ShuffleCycles = 50;
          for(int i=0;i<NumOfIterations;i++)
          {
           vector<int> Positions;
           if(MPI_MASTER)
           {
             int NumOfPositions = Rand.randomInt(MaxNumOfPositions);
             for(int pos=1;pos<m->numPositions()-1;pos++)
             {
               if(ExcludeAlphaHelices && IsInAlphaHelix(pos)) continue;
               Positions.push_back(pos);
             }
             int r1, r2, Temp;
             for(int j=0;j<ShuffleCycles;j++)
             {
               r1 = Rand.randomInt(Positions.size());
               r2 = Rand.randomInt(Positions.size());
               Temp = Positions[r1];
               Positions[r1] = Positions[r2];
               Positions[r2] = Temp;
             }
             for(int j=0;j<Positions.size()-NumOfPositions;j++)
             {
               Positions.pop_back();//throw away these elements
             }
             for(int _Rank=1;_Rank<Threads;_Rank++)
             {
               mpiSend(_Rank,Positions);
             }
           }
           else
           {
             mpiRecv(MPI_MASTER,Positions);
           }
           for(int Index=0;Index<Positions.size();Index++)
           {
             GradientBestPhiPsiPosMPI(Positions[Index],Degree,Precision);
           }
          }
}

void Top7::MonteCarloGradientBestPhiPsiMPI(float Degree,
                                   float KTi,
                                   float KTf,
                                   int Steps,
                                   bool fixedRange,
                                   bool recordSnapshots,
                                   char* Filename)

{
/*
0: -dphi, +dpsi
1:     0, +dpsi
2:       +dphi, +dpsi
3: -dphi, 0
4: +dphi, 0
5: -dphi, -dpsi
6:     0, -dpsi
7:       +dphi, -dpsi
*/
          float dKT = (KTf - KTi)/Steps;
          float KT = KTi;
          float dDegree = 0.01; //with this precision those degrees would be generated
          for(int step=0; step<Steps; step++)
          {
           int RndPos;
           float range;
           vector<float> PosRangeData;
           if(MPI_MASTER)
           {
             RndPos = Rand.randomInt(m->numPositions()-2)+1;
             if(fixedRange)
               range = Degree;
             else
               range = Rand.randomInt(int(Degree/dDegree))*dDegree;
             PosRangeData.push_back(RndPos+0.05);
             PosRangeData.push_back(range);
             for(int rank=1;rank<8;rank++)
             {
               mpiSend(rank,PosRangeData);
             }
           }
           else
           {
             mpiRecv(MPI_MASTER,PosRangeData);
             RndPos = int(PosRangeData[0]);
             range = PosRangeData[1];
           }
           OUT << "\nRndPos "<< RndPos << endl;
           OUT << "\nrange " << range << endl;
           float E0 = energyFlex(*m,*f);
           OUT << step << ": " << E0 << endl;
           float Phi0 = m->GetPhi(RndPos);
           float Psi0 = m->GetPsi(RndPos);
           float dPhi, dPsi;
           vector<float> E0_Rotamers;
           if(MPI_MASTER)
           {
             int CurrentRot = m->getActive(RndPos);
             for(int rot=0;rot<m->numRots(RndPos);rot++)
             {
               m->setActive(RndPos,rot);
               float Eng = energyFlex(*m,*f);
               E0_Rotamers.push_back(Eng);
             }
             m->setActive(RndPos,CurrentRot);
             dPhi = -range;
```

```
  dPsi = range;
}
else
{
  switch(MPI_ME)
  {
    case 1:
      dPhi = 0;
      dPsi = range;
    break;
    case 2:
      dPhi = range;
      dPsi = range;
    break;
    case 3:
      dPhi = -range;
      dPsi = 0;
    break;
    case 4:
      dPhi = range;
      dPsi = 0;
    break;
    case 5:
      dPhi = -range;
      dPsi = -range;
    break;
    case 6:
      dPhi = 0;
      dPsi = -range;
    break;
    case 7:
      dPhi = range;
      dPsi = -range;
  }
}
vector<float> energies;
m->perturbPhiAngle(RndPos,dPhi,false);
m->perturbPsiAngle(RndPos,dPsi,false);
int CurrentRotamer = m->getActive(RndPos);
//m->ReAlignRotamers(RndPos);
m->ReAlignRotamers();
for(int rot=0;rot<m->numRots(RndPos);rot++)
{
  m->setActive(RndPos,rot);
  float energy = energyFlex(*m,*f);
  energies.push_back(energy);
}
m->setActive(RndPos,CurrentRotamer);
m->perturbPhiAngle(RndPos,-dPhi,false);
m->perturbPsiAngle(RndPos,-dPsi,false);
//m->ReAlignRotamers(RndPos);
m->ReAlignRotamers();

float PhiFinal, PsiFinal;
int RotFinal;

if(MPI_MASTER)
{
  vector<vector<float> > VecEnergies;
  VecEnergies.push_back(E0_Rotamers);
  VecEnergies.push_back(energies);
  for(int rank=1;rank<8;rank++)
  {
    vector<float> temp;
    mpiRecv(rank,temp);
    VecEnergies.push_back(temp);
  }

  vector<vector<float> > probabilities;
  float pAcc=0;
  for(int i=0;i<VecEnergies.size();i++)
  {
    vector<float> probabilityTemp;
    for(int j=0;j<VecEnergies[i].size();j++)
    {
      float p = exp(-(VecEnergies[i][j]-E0)/KT);
      pAcc += p;
      probabilityTemp.push_back(p);
    }
    probabilities.push_back(probabilityTemp);
  }
  float RandomFloat = Rand.randomFloat(pAcc);
  pAcc = 0;
  int iSelected, jSelected;
  bool BreakLoop = false;
  for(int i=0;i<probabilities.size();i++)
  {
    if(BreakLoop) break;
    for(int j=0;j<probabilities[i].size();j++)
    {
      if(BreakLoop) break;
      pAcc += probabilities[i][j];
      if(RandomFloat<pAcc)
      {
        iSelected = i;
        jSelected = j;
        BreakLoop = true;
      }
    }
  }
  float dPhiSelected, dPsiSelected;
  int rotSelected;
  switch(iSelected)
  {
    case 0:
      dPhiSelected = 0;
      dPsiSelected = 0;
```

```
              rotSelected = jSelected;
            break;
            case 1:
              dPhiSelected = -range;
              dPsiSelected = range;
              rotSelected = jSelected;
            break;
            case 2:
              dPhiSelected = 0;
              dPsiSelected = range;
              rotSelected = jSelected;
            break;
            case 3:
              dPhiSelected = range;
              dPsiSelected = range;
              rotSelected = jSelected;
            break;
            case 4:
              dPhiSelected = -range;
              dPsiSelected = 0;
              rotSelected = jSelected;
            break;
            case 5:
              dPhiSelected = range;
              dPsiSelected = 0;
              rotSelected = jSelected;
            break;
            case 6:
              dPhiSelected = -range;
              dPsiSelected = -range;
              rotSelected = jSelected;
            break;
            case 7:
              dPhiSelected = 0;
              dPsiSelected = -range;
              rotSelected = jSelected;
            break;
            case 8:
              dPhiSelected = range;
              dPsiSelected = -range;
              rotSelected = jSelected;
            break;
          }
          PhiFinal = Phi0+dPhiSelected;
          PsiFinal = Psi0+dPsiSelected;
          RotFinal = jSelected;
          for(int Rank=1;Rank<8;Rank++)
          {
            mpiSend(Rank,PhiFinal);
            mpiSend(Rank,PsiFinal);
            mpiSend(Rank,RotFinal);
          }
        }
        else
        {
          mpiSend(MPI_MASTER,energies);
          mpiRecv(MPI_MASTER,PhiFinal);
          mpiRecv(MPI_MASTER,PsiFinal);
          mpiRecv(MPI_MASTER,RotFinal);
        }

        float CurrentPhi = m->GetPhi(RndPos);
        float CurrentPsi = m->GetPsi(RndPos);

        m->SetPhi(RndPos,PhiFinal);
        m->SetPsi(RndPos,PsiFinal);
        m->ReAlignRotamers();
        m->setActive(RndPos,RotFinal);

        OUT << "\nPhi: " << CurrentPhi << endl;
        OUT << "\nPsi: " << CurrentPsi << endl;
        OUT << "\nRot: " << m->getActive(RndPos) << endl;

        KT += dKT;
        if(MPI_MASTER && recordSnapshots)
        {
          ostringstream ss;
          ss << Filename << "_" << step << ".pdb";
          m->write(ss.str(),true);
        }
       }
}

void Top7::StochasticPerturbationGridRefinement(vector<float> degrees,
                                    vector<float> precision,
                                    int NumOfStructures,
                                    float MaxPerturbationAngle,
                                    string path,
                                    string name,
                                    int Rank,
                                    bool UseRamaCode,
                                    bool UniformRama,
                                    bool RecordSnapShots,
                                    bool Code2,
                                    bool doGRID)
{
        float E;

        ostringstream StreamEnergyBefore;
        StreamEnergyBefore << path << "/" << name << "_" << Rank << "_Before.txt";
        ofstream FileEnergyBefore(StreamEnergyBefore.str().c_str());
        ostringstream StreamEnergyAfter;
        StreamEnergyAfter << path << "/" << name << "_" << Rank << ".txt";
        ofstream FileEnergyAfter(StreamEnergyAfter.str().c_str());

        ostringstream PhiPsiRepr;
        PhiPsiRepr << path << "/" << name << "_" << Rank << "_phipsi.txt";
```

```
        ofstream FilePhiPsiRepr(PhiPsiRepr.str().c_str());


        assert(degrees.size()==precision.size());
        m->SaveAll();
        for(int i=0;i<NumOfStructures;i++)
        {
         cout <<"\n---------- Structure "<<i<<" ----------" << endl;
         if(UseRamaCode)
         {
           RamaStructurePerturbation(UniformRama,Code2);
         }
         else
         {
           RandomStructurePerturbation(MaxPerturbationAngle);
         }
         m->ReAlignRotamers();
         ostringstream beforeRefine;
         beforeRefine << path << "/" << name << "_" << Rank << "_" << i << "_before.pdb";
         m->write(beforeRefine.str());
         E = energyFlex(*m,*f);
         FileEnergyBefore << E << endl;
         if(doGRID)
         {
           for(int j=0;j<degrees.size();j++)
           {
             float deg = degrees[j];
             float prec = precision[j];
             GradientBestPhiPsi(deg,prec,true,RecordSnapShots);
           }
           GreedyMinimization();
         }
         ostringstream afterRefine;
         afterRefine << path << "/" << name << "_" << Rank << "_" << i << ".pdb";
         m->write(afterRefine.str());
         E = energyFlex(*m,*f);
         FileEnergyAfter << E << endl;

         vector<float> phipsiRep = PhiPsiRepresentation();
         for(int j=0;j<phipsiRep.size();j++)
           FilePhiPsiRepr << phipsiRep[j] << ",";
         FilePhiPsiRepr << endl;

         m->CleanSavesKeepFirst();
         bool InPlace = true;
         m->RewindAll(InPlace);
        }
        FileEnergyBefore.close();
        FileEnergyAfter.close();
        FilePhiPsiRepr.close();
}

void Top7::StochasticPerturbationBackrubRefinement(float MaxAngle,
                                        float Precision,
                                       float MaxPerturbationAngle,
                                        float KTi,
                                        float KTf,
                                       int Steps,
                                       int Length,
                                       float BiasProbability,
                                       bool domin,
                                       int NumOfStructures,
                                       string path,
                                       string name,
                                       int Rank,
                                       bool RecordSnapShots)
{
        OUT << "We entered this module..." << endl;
        float E;
        ostringstream StreamEnergyBefore;
        StreamEnergyBefore << path << "/" << name << "_" << Rank << "_Before.txt";
        ofstream FileEnergyBefore(StreamEnergyBefore.str().c_str());
        ostringstream StreamEnergyAfter;
        StreamEnergyAfter << path << "/" << name << "_" << Rank << ".txt";
        ofstream FileEnergyAfter(StreamEnergyAfter.str().c_str());

        OUT << "ready before saveall.... " << endl;
        m->SaveAll();
        for(int i=0;i<NumOfStructures;i++)
        {
         cout <<"\n---------- Structure "<<i<<" ----------" << endl;
         RandomStructurePerturbation(MaxPerturbationAngle);
         OUT << "Random Structure Perturbation Executed...\n"<<endl;
         m->ReAlignRotamers();
         ostringstream beforeRefine;
         beforeRefine << path << "/" << name << "_" << Rank << "_" << i << "_before.pdb";
         m->write(beforeRefine.str());
           E = energyFlex(*m,*f);
         FileEnergyBefore << E << endl;
         FlexBackrubMonteCarlo(MaxAngle,Precision,KTi,KTf,Steps,Length,BiasProbability,domin);
         ostringstream afterRefineBeforeMin;
         afterRefineBeforeMin << path << "/" << name << "_" << Rank << "_" << i <<".pdb";
         m->write(afterRefineBeforeMin.str());
         GreedyMinimization();
         ostringstream afterRefine;
         afterRefine << path << "/" << name << "_" << Rank << "_" << i << "_Min.pdb";
         m->write(afterRefine.str());
         E = energyFlex(*m,*f);
         FileEnergyAfter << E << endl;

         m->CleanSavesKeepFirst();
         bool InPlace = true;
         m->RewindAll(InPlace);
        }
        FileEnergyBefore.close();
        FileEnergyAfter.close();
}
```

```
void Top7::StochasticRamaPerturbationDecoyGeneration(int NumOfStructures,
                                        string path,
                                        string name,
                                        float PerturbationRate)
{
         int Rank = MPI_ME;
         m->SaveAll();
         for(int i=0;i<NumOfStructures;i++)
         {
          cout <<"\n---------- Structure "<<i<<" ----------" << endl;

          bool UniformRama = false;
          bool Code2 = false;
          RamaStructurePerturbation(UniformRama,Code2,PerturbationRate);
          m->ReAlignRotamers();
          //float Energy = energyFlex(*m,*f);
          ostringstream beforeRefine;
          beforeRefine << path << "/" << name << "_" << Rank << "_" << i << "_before.pdb";
          m->write(beforeRefine.str());
          m->CleanSavesKeepFirst();
          bool InPlace = true;
          m->RewindAll(InPlace);
         }
}


void Top7::ReplicaGRID(double degree,
                       double precision,
                       size_t steps,
                       double kT)
{
/*
running "steps" steps of monte carlo and accept based on metropolis criterion
*/
         int numPos = m->numPositions();
         double E_prev;
         E_prev = INFTY;
         for(size_t step=0;step<steps;step++)
         {
          int rot_min_E = -1;
          double min_E = INFTY;
          int RndPos = Rand.randomInt(numPos);
          float dPhi = precision*(Rand.randomInt(2*int(degree/precision))-int(degree/precision));
          float dPsi = precision*(Rand.randomInt(2*int(degree/precision))-int(degree/precision));
          m->perturbPhiAngle(RndPos,dPhi,false);
          m->perturbPsiAngle(RndPos,dPsi,false);
          int CurrentRotamer = m->getActive(RndPos);
          m->ReAlignRotamers(RndPos);
          for(int rot=0;rot<m->numRots(RndPos);rot++)
          {
            m->setActive(RndPos,rot);
            float energy = energyFlex(*m,RndPos,*f);
            if(energy<min_E)
            {
              rot_min_E = rot;
              min_E = energy;
            }
          }
          m->setActive(RndPos,rot_min_E);
          min_E = energyFlex(*m,*f);
          float dE = min_E-E_prev;
          float tempRnd = Rand.randomFloat(1);
          if(tempRnd<exp(-dE/kT))
          {
            E_prev = min_E;
            /*
            OUT << "Energy: " << E_prev << endl;
            OUT << "dPhi: " << dPhi << endl;
            OUT << "dPsi: " << dPsi << endl;
            OUT << "Rot: " << rot_min_E << endl;*/
            //accept
            m->ReAlignRotamers();
          }
          else
          {//rewind
            m->setActive(RndPos,CurrentRotamer);
            m->perturbPhiAngle(RndPos,-dPhi,false);
            m->perturbPsiAngle(RndPos,-dPsi,false);
            m->ReAlignRotamers(RndPos);
          }
         }
}

void Top7::ReplicaExchangeMonteCarloGRID(vector<float> degrees,
                                  vector<float> precisions,
                                  int rounds,
                                  int steps_sync,
                                  vector<float> kTs,
                                  int round)
{
/*
The idea is that we have kT.size() different processes. Each process
is running a monte carlo similation at its own kT (temprature). Every
steps_sync, all of processes stop and based on their energies and their
tempratures they swap their configurations (or equally their tempratures and
their order in line)
T1, T2, T3, ..., Tn
c1, c2, c3, ..., cn
*/
         struct ReplicaConfig
         {
          int rank;
          double lastE;
         };

         struct ReplicaTempratureProfile
```

```
{
 double kT;
 double deg; //degree
 double prec; //precision
};

assert(degrees.size()==precisions.size());
assert(degrees.size()==kTs.size());
assert(degrees.size()==MPI_SIZE);

ReplicaTempratureProfile *profiles;
profiles = new ReplicaTempratureProfile[MPI_SIZE];

int Rank = MPI_ME;

ReplicaConfig *processes;
processes = new ReplicaConfig[MPI_SIZE];

for(int i=0;i<MPI_SIZE;i++)
{
 profiles[i].deg = degrees[i];
 profiles[i].prec = precisions[i];
 profiles[i].kT = kTs[i];

 processes[i].lastE = INFTY;
 processes[i].rank = Rank;
}

mapStruct processProfile; //process number, profile number

int SnapShotRecordingPdbEnergy = 5; // every five rounds

ostringstream StreamEnergies;
StreamEnergies << "Energies_round_" << round << "_rank_" << Rank << ".txt";
ofstream FileEnergy(StreamEnergies.str().c_str());

//Initialize the assignment of processes to temprature profiles
for(int i=0;i<MPI_SIZE;i++)
 processProfile.set(i,i);

//Production part
for(int i=0;i<rounds;i++)
{

 if(MPI_MASTER)
   for(int i=1;i<MPI_SIZE;i++)
     mpiSend(i,processProfile.m);
 else
   mpiRecv(MPI_MASTER,processProfile.m);

 int profileNum = processProfile.get(Rank);
 double kT = profiles[profileNum].kT;
 double deg = profiles[profileNum].deg;
 double prec = profiles[profileNum].prec;

 ReplicaGRID(deg,prec,steps_sync,kT);
 double eng = energyFlex(*m,*f);

 if(MPI_MASTER)
 {
   processes[0].lastE = eng;
   for(int j=1;j<MPI_SIZE;j++)
   {
     double engTemp;
     mpiRecv(j,engTemp);
     processes[j].lastE = engTemp;
   }
 }
 else
 {
   mpiSend(MPI_MASTER,eng);
 }
 if(MPI_MASTER)
 {
   OUT << "THIS IS MASTER: " << endl;
   processProfile.show();
   for(int j=0;j<MPI_SIZE;j++)
   {
     OUT << j << ":" << processes[j].lastE << endl;
   }

   for(int k=0;k<3;k++) //shuffle this many times
   for(int j=0;j<MPI_SIZE-1;j++)
   {
     double kT_j = profiles[j].kT;
     double kT_j_plus_1 = profiles[j+1].kT;
     int p_j = processProfile.get_inv(j);
     int p_j_plus_1 = processProfile.get_inv(j+1);
     double E_j = processes[p_j].lastE;
     double E_j_plus_1 = processes[p_j_plus_1].lastE;

     double temp1 = (E_j-E_j_plus_1);
     double temp2 = (1.0/kT_j-1.0/kT_j_plus_1);
     double prob = exp(temp1*temp2);
     double rndTemp = Rand.randomFloat(1);

     if(rndTemp<prob)
     {
       processProfile.set(p_j,j+1);
       processProfile.set(p_j_plus_1,j);
       processes[p_j].lastE = E_j_plus_1;
       processes[p_j_plus_1].lastE = E_j;
     }
   }
 }
 //record every 5 rounds
 if(i%5==0)
```

```
          {
            SnapShotRecorder(i+MPI_SIZE*(round-1),MPI_ME);
            FileEnergy << eng << endl;
          }

          // every twenty rounds, sync with the lowest energy structure
         }

         FileEnergy.close();

/*
          replicas[Rank].E = energyFlex(*m,*f);

          if(MPI_MASTER)
          {
            vector<double> temp;
            vector<pair<int,double> > tempIndex_kT;
            tempIndex_kT.push_back(pair<int,double>(0,replicas[0].kT));
            for(int r=1;r<MPI_SIZE;r++)
            {
              mpiRecv(r,temp);
              replicas[r].importData(temp);
              tempIndex_kT.push_back(pair<int,double>(r,replicas[r].kT));
            }

            tempIndex_kT = Sort(tempIndex_kT);
            for(int j=0;j<tempIndex_kT.size();j++)
            {
              OUT << j << ": (" << tempIndex_kT[j].first << "," << tempIndex_kT[j].second << ")" << endl;
            }

          }
          else
          {
            mpiSend(MPI_MASTER,replicas[Rank].exportData());
          }
         }
*/
         delete [] profiles;
         delete [] processes;
}

void Top7::SimulatedAnnealingReplicaExchangeMonteCarloGRID(float kT_high_i,
                                                           float kT_high_f,
                                                           float kT_low_i,
                                                           float kT_low_f,
                                                           float degree_scale,
                                                           int rounds,
                                                           int rounds_of_rounds,
                                                           int steps_sync)
{
         double dkT_high = (kT_high_i - kT_high_f)/rounds_of_rounds;
         double dkT_low = (kT_low_i - kT_low_f)/rounds_of_rounds;
         for(int i=0;i<rounds_of_rounds;i++)
         {
          vector<float> degrees;
          vector<float> precisions;
          vector<float> kTs;
          float temp_kT_high = kT_high_i-i*dkT_high;
          float temp_kT_low = kT_low_i-i*dkT_low;
          for(int j=0;j<MPI_SIZE;j++)
          {
            float delta_kT = (temp_kT_high-temp_kT_low)/MPI_SIZE;
            float my_kT = temp_kT_high-j*delta_kT;
            kTs.push_back(my_kT);
            degrees.push_back(my_kT*degree_scale);
            precisions.push_back(0.05);
          }
          ReplicaExchangeMonteCarloGRID(degrees,precisions,rounds,steps_sync,kTs,i+1);
         }
}



void Top7::SnapShotRecorder(int SnapShot, int Pos)
{
         ostringstream filename;
         filename << "SnapShot_" << SnapShot << "_" << MPI_ME << "_" << Pos << ".pdb";
         bool includeConnectivity = true;
         m->write(filename.str(),includeConnectivity);
}

vector<float> Top7::PhiPsiRepresentation()
{
         vector<float> result;
         for(int pos=0;pos<m->numPositions();pos++)
         {
          float phi = m->GetPhi(pos);
          float psi = m->GetPsi(pos);
          result.push_back(phi);
          result.push_back(psi);
         }
         return result;
}

void Top7::StochasticPDBFragmentRefinement()
{//for now we'll have many assumptions, then we make it generalized
         string Path = "/home/chitsaz/test/flex-test/2009/July_15/testDB_";
         string PDB = "1PGA";

         int NumOfFrags = 18;
         int FragLength = 3;
         vector<vector<vector<float> > > BkBnConf;
         vector<vector<float> > dummy;
         BkBnConf.push_back(dummy);
         for(int i=1;i<NumOfFrags;i++)
         {
```

```
 ostringstream oss;
 oss << Path << PDB << FragLength*i-1 <<".txt";
 ifstream file(oss.str().c_str(),ifstream::in);
 cout << "\n\ni: " << i << endl;
 vector<vector<float> > FileTuples;
 while(file.good())
 {
   string str;
   file >> str;
   if(str.compare("")==0) break;
   char * pch;
   vector<float> tuple;
   pch = strtok((char*)str.c_str(),",");
   while(pch != NULL)
   {
     tuple.push_back(atof(pch));
     pch = strtok(NULL, ",");
   }
   FileTuples.push_back(tuple);
 }
 file.close();
 BkBnConf.push_back(FileTuples);
}
//in BkBnConf we have all conformations of backbone....

for(int i=1;i<BkBnConf.size();i++)
{
 cout << "\n\ni: " << i<<endl;
 for(int j=0;j<BkBnConf[i].size();j++)
 {
   for(int k=0;k<BkBnConf[i][j].size();k++)
   {
     cout << BkBnConf[i][j][k] << ",";
   }
   cout << "\n";
 }
}
m->SaveAll();
m->SaveAll();
float KT = 1;
int Steps = 1;
vector<int> positions;
for(int i=0;i<56;i++)
 positions.push_back(i);
for(int i=0;i<Steps;i++)
{
 int RandomFrag, RandomBkBnConf;
// do
// {
   RandomFrag = Rand.randomInt(NumOfFrags-2)+1;
   cout << "RandomFrag: " << RandomFrag << endl;
   RandomBkBnConf = Rand.randomInt(BkBnConf[RandomFrag].size());
   cout << "RandomBkBnConf: " << RandomBkBnConf << endl;
   cout << "BkBnConf[RandomFrag].size(): " << BkBnConf[RandomFrag].size() << endl;
// }while(BkBnConf[RandomFrag].size()<=1);
 vector<float> tuple = BkBnConf[RandomFrag][RandomBkBnConf];
 float E1 = energyFlex(*m,*f);
 cout << "E1: " << E1 << endl;

 /*
 positions.push_back(RandomFrag*3-2);
 positions.push_back(RandomFrag*3-1);
 positions.push_back(RandomFrag*3);
 m->SaveInPlace(positions);*/
 m->SaveAll(true);


 /*
 for(int j=0;j<FragLength;j++)
 {
   cout << "\n"<<RandomFrag*3-2+j+1<<":"<<tuple[j*3+1]<<endl;
   m->SetPhi(RandomFrag*3-2+j,tuple[j*3+1]);
   cout << "\n"<<RandomFrag*3-2+j+1<<":"<<tuple[j*3+2]<<endl;
   m->SetPsi(RandomFrag*3-2+j,tuple[j*3+2]);
   cout << "\n"<<RandomFrag*3-2+j+1<<":"<<tuple[j*3+3]<<endl;
   m->SetOmega(RandomFrag*3-2+j,tuple[j*3+3]);
 }*/

 int StartPos = (RandomFrag-1)*FragLength+1;
 int StopPos = StartPos + FragLength;

 cout << "\nBefore Perturbation....." << endl;
 for(int i=StartPos-1;i<StopPos+1;i++)
 {
   cout << i+1 <<":"<< m->GetPhi(i) << ",";
   cout << i+1 <<":"<< m->GetPsi(i) << ",";
   cout << i+1 <<":"<< m->GetOmega(i) << endl;
 }

 cout <<"\nShould be..." << endl;
 for(int j=0;j<FragLength;j++)
 {
   cout << StartPos+j+1<<":"<<tuple[j*3+1]<<",";
   m->setPhiAngleFragment(StartPos+j,StopPos,tuple[j*3+1]);
   cout << StartPos+j+1<<":"<<tuple[j*3+2]<<",";
   m->setPsiAngleFragment(StartPos+j,StopPos,tuple[j*3+2]);
   cout << StartPos+j+1<<":"<<tuple[j*3+3]<<endl;
   m->setOmgAngleFragment(StartPos+j,StopPos,tuple[j*3+3]);
 }

 cout << "\nAfter Perturbation....." << endl;
 for(int i=StartPos-1;i<StopPos+1;i++)
 {
   cout << i+1 <<":"<< m->GetPhi(i) << ",";
   cout << i+1 <<":"<< m->GetPsi(i) << ",";
   cout << i+1 <<":"<< m->GetOmega(i) << endl;
 }
```

```
          float E2 = energyFlex(*m,*f);
          cout << "E2: " << E2 << endl;
          float dE = E2 - E1;
          float p = exp(-dE/KT);
          bool Accept=true;
          if(p<1)
          {
            float rnd = Rand.randomFloat(1);
            if(rnd>p)//reject the move
            {
//            Accept = false;
            }
          }
          if(Accept)
          {
            cout << "\nAccepted..."<<endl;
            m->ReAlignRotamers(positions);
            m->SaveAll(true);
          }
          else
          {
            cout << "\nRejected..."<<endl;
            m->RewindInPlace(positions);
            //m->RewindAll(true);
          }
          cout << "\n"<< i << ":" << energyFlex(*m,*f) << endl;
          cout << "\nLength is supposed to be: " << tuple[0] << endl;
         }
}

void Top7::GreedyMinimization()
{
         float E1, E2;
         bool accept;
         bool replaceLastElement = true;
         m->SaveAll(replaceLastElement);
         for(int i=0;i<m->numPositions();i++)
         {
          E1 = energyFlex(*m,*f);
//        m->setActive(i,0);
          int r1 = m->getActive(i);
          m->SaveInPlace(i);
          Minimization(i,r1);
          E2 = energyFlex(*m,*f);
          accept = (E2<=E1);
          if(accept)
          {
            bool ReplaceLastElement = true;
            m->SaveAll(ReplaceLastElement);
            /*for(int j=0;j<m->numRots(i);j++)
            {
              Minimization(i,j);
            }*/
          }
          else
          {
            m->RewindInPlace(i);
          }
          float E = energyFlex(*m,*f);
          OUT << "\nPos: " << i << ", Eng: " << E << endl;
         }
}

bool Top7::ByPassRank(float MaxDegree, float DegreePrecision)
{
         int NumAngles = ceil(MaxDegree/DegreePrecision);
         int Threads = (2*NumAngles+1)*(2*NumAngles+1);
         if(MPI_ME>=Threads)
         {
          OUT << "\nI got bypassed: " << MPI_ME << endl;
          return true;
         }
         return false;
}

map<string,float> Top7::CorrelationTwoPos(int Pos1, int Pos2)
{
         map<string,float> result;
         float SumEnergy=0;
         float DiffAccumulator=0;
         float AvgEnergy;
         float Variance;
         float StdDev;
         for(int rot1=0;rot1<m->numRots(Pos1);rot1++)
         {
          for(int rot2=0;rot2<m->numRots(Pos2);rot2++)
          {
            SumEnergy+=energyFlex(*m,Pos1,rot1,Pos2,rot2,*f);
          }
         }
         AvgEnergy = SumEnergy/(m->numRots(Pos1)*m->numRots(Pos2));
         for(int rot1=0;rot1<m->numRots(Pos1);rot1++)
         {
          for(int rot2=0;rot2<m->numRots(Pos2);rot2++)
          {
            DiffAccumulator+=pow((energyFlex(*m,Pos1,rot1,Pos2,rot2,*f)-AvgEnergy),2);
          }
         }
         Variance = DiffAccumulator/(m->numRots(Pos1)*m->numRots(Pos2));
         StdDev = sqrt(Variance);
         result["SumEnergy"]=SumEnergy;
         result["AvgEnergy"]=AvgEnergy;
         result["Variance"]=Variance;
         result["StdDev"]=StdDev;
         result["DiffAcc"] = DiffAccumulator;
         return result;
```

```
}

void Top7::RotamerPerturb(int Position)
{
         OUT << "\n>>>Entered Top7::Perturb() " << endl;
         OUT << "\n"<<m->confkeys[Position].size()<<endl;
         int RandomRot = Rand.randomInt(m->confkeys[Position].size());
         OUT << "\n"<<RandomRot << endl;
         m->setActive(Position,RandomRot);
         OUT << "\n<<<Left Top7::Perturb() " << endl;
}

void Top7::RandomStructurePerturbation(float MaxAngle,bool Sequential)
{
         vector<int> positions;
         for(int p=1;p<m->numPositions()-1;p++)
         {
          positions.push_back(p);
         }
         int Shuffles=0;

         if(!Sequential)
         {
          for(int i=0;i<Shuffles;i++)
          {
            int RandPos1 = Rand.randomInt(positions.size());
            int RandPos2 = Rand.randomInt(positions.size());
            int temp = positions[RandPos2];
            positions[RandPos2] = positions[RandPos1];
            positions[RandPos1] = temp;
          }
         }
         for(int i=0;i<positions.size();i++)
         {
          int p = positions[i];
          float randomPhiPerturb = Rand.randomFloat(2*MaxAngle)-MaxAngle;
          float randomPsiPerturb = Rand.randomFloat(2*MaxAngle)-MaxAngle;
          float CurrentPhi = m->GetPhi(p);
          float CurrentPsi = m->GetPsi(p);
          m->SetPhi(p,CurrentPhi+randomPhiPerturb);
          m->SetPsi(p,CurrentPsi+randomPsiPerturb);
         }
}

void Top7::RamaStructurePerturbation(bool Uniform, bool Code2, float PerturbationRate)
{
         vector<int> Code;
         vector<char> TempCode;
         if(!Code2)
         {
          TempCode = RamaCodeOfStructure();
          for(int i=0;i<TempCode.size();i++)
          {
            Code.push_back(int(TempCode[i]));
          }
         }
         else
         {
          Code = RamaCode2OfStructure();
         }

         for(int p=1;p<m->numPositions()-1;p++)
         {
          int RandomNum = Rand.randomInt(1000);
          if(RandomNum>1000*PerturbationRate)
            continue;
          OUT << "\n Pos: " << p << endl;
          pair<float,float> phipsi =  RamaRandomPhiPsi(Code[p-1],
                                        m->confRestype(p),
                                        Uniform,
                                        Code2);
          OUT << p << ":" << phipsi.first << "," << phipsi.second << endl;
          float phi = phipsi.first;
          float psi = phipsi.second;

          m->SetPhi(p,phi);
          m->SetPsi(p,psi);
         }
}

vector<vector<int> > Top7::GetIntMatrix(string filename)
{
         vector<vector<int> > result;
         vector<int> tuple;
         ifstream file(filename.c_str(),ifstream::in);
         if(!file.is_open())
          OUT << "\nGetIntMatrix couldn't open its file..." << endl;
         while(file.good())
         {
          string str;
          file >> str;
          if(str.compare("")==0) break;
          char * pch;
          vector<int> tuple;
          pch = strtok((char*)str.c_str(),",");
          while(pch != NULL)
          {
            tuple.push_back(atoi(pch));
            pch = strtok(NULL, ",");
          }
          result.push_back(tuple);
         }
         return result;
}

vector<vector<float> > Top7::GetFloatMatrix(string filename)
{
```

```
          vector<vector<float> > result;
          vector<float> tuple;
          ifstream file(filename.c_str(),ifstream::in);
          if(!file.is_open())
           OUT << "\nGetIntMatrix couldn't open its file..." << endl;
          while(file.good())
          {
           string str;
           file >> str;
           if(str.compare("")==0) break;
           char * pch;
           vector<float> tuple;
           pch = strtok((char*)str.c_str(),",");
           while(pch != NULL)
           {
             tuple.push_back(atof(pch));
             pch = strtok(NULL, ",");
           }
           result.push_back(tuple);
          }
          return result;
}

template <class T> pair<T,pair<int,int> > Top7::max(vector<vector<T> > vec)
{
          T maximum = -INFTY;
          int imax, jmax;
          for(int i=0;i<vec.size();i++)
          {
           for(int j=0;j<vec[i].size();j++)
           {
             if(maximum<vec[i][j])
             {
               maximum = vec[i][j];
               imax = i;
               jmax = j;
             }
           }
          }
          pair<int,int> Index = pair<int,int>(imax,jmax);
          return pair<T,pair<int,int> >(maximum,Index);
}

template <class T> pair<T,pair<int,int> > Top7::min(vector<vector<T> > vec)
{
          T minimum = INFTY;
          int imin, jmin;
          for(int i=0;i<vec.size();i++)
          {
           for(int j=0;j<vec[i].size();j++)
           {
             if(minimum>vec[i][j])
             {
               minimum = vec[i][j];
               imin = i;
               jmin = j;
             }
           }
          }
          pair<int,int> Index = pair<int,int>(imin,jmin);
          return pair<T,pair<int,int> >(minimum,Index);
}

void Top7::LoadRamaCode2DataStructures()
{
          OUT << "\n>>Ready to Load Code2..... " << endl;
          string Forward = "/home/chitsaz/phoenix/externaldata/RamaCode2Forward.csv";
          //string Forward = "/home/chitsaz/phoenix/externaldata/Forward106.csv";

          ifstream fileForward(Forward.c_str(),ifstream::in);
          if(!fileForward.is_open())
           OUT << "\nFile of Ramachandran Code2 Forward couldn't be opened..." << endl;

          while(fileForward.good())
          {
           string str;
           fileForward >> str;
           if(str.compare("")==0) break;
           char *pch;
           vector<int> tuple;
           pch = strtok((char*)str.c_str(),",");
           while(pch != NULL)
           {
             tuple.push_back(atoi(pch));
             pch = strtok(NULL, ",");
           }
           RamaCode2.push_back(tuple);
          }
          fileForward.close();

          string Reverse = "/home/chitsaz/phoenix/externaldata/RamaCode2Reverse.csv";
          //string Reverse = "/home/chitsaz/phoenix/externaldata/Reverse106.csv";

          ifstream fileReverse(Reverse.c_str(),ifstream::in);
          if(!fileReverse.is_open())
           OUT << "\nFile of Ramachandran Code2 Reverse couldn't be opened..." << endl;

          while(fileReverse.good())
          {
           string strLetter;
           string strListi;
           string strListj;
           fileReverse >> strLetter;
           fileReverse >> strListi;
           fileReverse >> strListj;
           if(strLetter.compare("")==0) break;
           if(strListi.compare("") ==0) break;
```

```
          if(strListj.compare("") ==0) break;
          char *pchListi;
          char *pchListj;

          int Letter;
          vector<int> tuplei;
          vector<int> tuplej;

          Letter = atoi(strLetter.c_str());

          pchListi = strtok((char*)strListi.c_str(),",");
          while(pchListi != NULL)
          {
            tuplei.push_back(atoi(pchListi));
            pchListi = strtok(NULL, ",");
          }

          pchListj = strtok((char*)strListj.c_str(),",");
          while(pchListj != NULL)
          {
            tuplej.push_back(atoi(pchListj));
            pchListj = strtok(NULL, ",");
          }

          vector<pair<int,int> > Tuplesij;
          for(int i=0;i<tuplei.size();i++)
          {
            Tuplesij.push_back(pair<int,int>(tuplei[i],tuplej[i]));
          }
          RamaCode2ToIndexes[Letter]=Tuplesij;
         }
         fileReverse.close();
         OUT << "\n<<Finished loading Code2..... " << endl;
}

void Top7::LoadRamaCodeDataStructures()
{
         OUT << "\n>>Load Rama Code Data Structures..." << endl;
         ifstream file("/home/chitsaz/phoenix/externaldata/RAMACODE.csv",ifstream::in);
         if(!file.is_open())
          OUT << "\nFile of Ramachandran Code couldn't be opened..." << endl;

         while(file.good())
         {
          string str;
          file >> str;
          if(str.compare("")==0) break;
          char * pch;
          vector<char> tuple;
          pch = strtok((char*)str.c_str(),",");
          while(pch != NULL)
          {
            tuple.push_back(*pch);
            pch = strtok(NULL, ",");
          }
          RamaCode.push_back(tuple);
         }
         file.close();


         ResidueNameToIndex.insert(pair<string,int>("ALA",0));
         ResidueNameToIndex.insert(pair<string,int>("ARG",1));
         ResidueNameToIndex.insert(pair<string,int>("ASN",2));
         ResidueNameToIndex.insert(pair<string,int>("ASP",3));
         ResidueNameToIndex.insert(pair<string,int>("CYS",4));
         ResidueNameToIndex.insert(pair<string,int>("GLU",5));
         ResidueNameToIndex.insert(pair<string,int>("GLN",6));
         ResidueNameToIndex.insert(pair<string,int>("GLY",7));
         ResidueNameToIndex.insert(pair<string,int>("HIS",8));
         ResidueNameToIndex.insert(pair<string,int>("ILE",9));
         ResidueNameToIndex.insert(pair<string,int>("LEU",10));
         ResidueNameToIndex.insert(pair<string,int>("LYS",11));
         ResidueNameToIndex.insert(pair<string,int>("MET",12));
         ResidueNameToIndex.insert(pair<string,int>("PHE",13));
         ResidueNameToIndex.insert(pair<string,int>("PRO",14));
         ResidueNameToIndex.insert(pair<string,int>("SER",15));
         ResidueNameToIndex.insert(pair<string,int>("THR",16));
         ResidueNameToIndex.insert(pair<string,int>("TRP",17));
         ResidueNameToIndex.insert(pair<string,int>("TYR",18));
         ResidueNameToIndex.insert(pair<string,int>("VAL",19));

         map<string,int>::iterator it = ResidueNameToIndex.begin();
//       map<string,int>::iterator it2 =
         for(int i=0;i<ResidueNameToIndex.size();i++)
         {
          stringstream ss;
          ss << "/home/chitsaz/phoenix/externaldata/";
          ss << (*it).first << ".csv";

          ifstream file(ss.str().c_str(),ifstream::in);
          if(!file.is_open())
            OUT << "\nFile of Ramachandran Code couldn't be opened..." << endl;
          while(file.good())
          {
            string str;
            file >> str;
            if(str.compare("")==0) break;
            char * pch;
            vector<int> tuple;
            pch = strtok((char*)str.c_str(),",");
            while(pch != NULL)
            {
              tuple.push_back(atoi(pch));
              pch = strtok(NULL, ",");
            }
            ResidueRamaDistribution[(*it).first].push_back(tuple);
          }
```

```
          file.close();
          it++;
          }


//ASCI CODE A:65, Z:90

          vector<vector<pair<int,int> > > temp;
          temp.resize(26);
          for(int i=0;i<36;i++)
          {
           for(int j=0;j<36;j++)
           {
             temp[int(RamaCode[i][j]-65)].push_back(pair<int,int>(i,j));
           }
          }

          for(char Code='A';Code<='Z';Code++)
          {
           if(Code=='J' || Code=='O' || Code=='U' || Code=='X')
             continue;
           RamaCodeToIndexes[Code] = temp[Code-65];
          }
          OUT << "\n<<Load Rama Code Data Structures..." << endl;
}

vector<int> Top7::RamaCode2OfStructure()
{
          vector<int> result;
          for(int p=1;p<m->numPositions()-1;p++)
          {
           float phi = m->GetPhi(p)+180;
           float psi = m->GetPsi(p)+180;
           int j = floor(phi/10);
           int i = 35-floor(psi/10);
           OUT << "\nPos: " << p;
           OUT << "\nPhi: " << phi << "\t" << j;
           OUT << "\nPsi: " << psi << "\t" << i;
           OUT << "\nCode: " <<  RamaCode2[i][j] << endl;
           result.push_back(RamaCode2[i][j]);
          }
          return result;
}


vector<char> Top7::RamaCodeOfStructure()
{
          vector<char> result;
          for(int p=1;p<m->numPositions()-1;p++)
          {
           float phi = m->GetPhi(p)+180;
           float psi = m->GetPsi(p)+180;
           int j = floor(phi/10);
           int i = 35-floor(psi/10);
           OUT << "\nPos: " << p;
           OUT << "\nPhi: " << phi << "\t" << j;
           OUT << "\nPsi: " << psi << "\t" << i;
           OUT << "\nCode: " <<  RamaCode[i][j] << endl;
           result.push_back(RamaCode[i][j]);
          }
          return result;
}

pair<float,float> Top7::RamaRandomPhiPsi(int Code,
                                  string Residue,
                                  bool Uniform,
                                  bool Code2)
{
          OUT << "\n";
          OUT << (char)Code << ", " << Residue << endl;
          vector<pair<int,int> > Indexes;
          if(Code2==false)
          {
           Indexes = RamaCodeToIndexes[char(Code)];
          }
          else
          {
           Indexes = RamaCode2ToIndexes[Code];
          }


          for(int i=0;i<Indexes.size();i++)
          {
           OUT << "--- " << Indexes[i].first << "," << Indexes[i].second << endl;
          }
          OUT << "\n" << endl;


          vector<vector<int> > Distribution;
          if(Residue.compare("HID")==0)
          {
           Distribution = ResidueRamaDistribution["HIS"];
          }
          else
          {
           Distribution = ResidueRamaDistribution[Residue];
          }
/*
          for(int i=0;i<Distribution.size();i++)
          {
           for(int j=0; j<Distribution[i].size(); j++)
           {
             OUT << Distribution[i][j] << ",";
           }
           OUT << endl;
          }
*/
```

```
          if(Uniform)
          {
           for(int i=0;i<Distribution.size();i++)
           {
             for(int j=0;j<Distribution.size();j++)
             {
               Distribution[i][j] = 100;//arbitrary but constant... make it uniform
             }
           }
          }
          vector<int> CodeDistribution;
          CodeDistribution.resize(Indexes.size());
          int AccSum=0;
          for(int k=0;k<Indexes.size();k++)
          {
           int i = Indexes[k].first;
           int j = Indexes[k].second;
           AccSum+=Distribution[i][j];
           CodeDistribution[k]=AccSum;
          }
          int RandomNum = Rand.randomInt(AccSum);
          for(int k=0;k<Indexes.size();k++)
          {
           if(RandomNum<=CodeDistribution[k])
           {
             int i=Indexes[k].first;
             int j=Indexes[k].second;
             float phi = 10*j-180+Rand.randomFloat(10);
             float psi = 170-10*i+Rand.randomFloat(10);
             return pair<float,float>(phi,psi);
           }
          }
          cout << "\nError in Top7: RandomPhiPsi, Code:" <<Code<<", Residue: "<<Residue<<endl;
          return pair<float,float>(999.9,999.9); // In case error happens to occur...
}

float Top7::Move()
{
//        OUT << "\n>>>Entered Top7::Move() " << endl;
          int WindowSize = 5;
          float AcceptanceProbability = 0.1;
          int MinNumOfSimultanousPerturb = 3;
          int MinNumOfIterations =20;//70 and 35 look fine
          int NumOfRepeatedEngToStop = 10;
          float MaxDegree = 3;
          float DegreePrecision = 0.1;
          int MaxNumOfBkBnPosToPerturb = 5;
          int Windows = int(ceil(nump/float(WindowSize)));
          int RandomNumOfPerturb = Rand.randomInt(MaxNumOfBkBnPosToPerturb);
          BackBonePerturb(RandomNumOfPerturb,MaxDegree,DegreePrecision);
          /*We've commented out this part for now
          for(int i=0;i<Windows;i++)
          {
           GreedyRotamerOptimization();
           if(i<Windows-1)
           {
             WindowOptimization(WindowSize*i,
                                WindowSize*(i+1),
                                AcceptanceProbability,
                                MinNumOfSimultanousPerturb,
                                MinNumOfIterations,
                                NumOfRepeatedEngToStop);
           }
           else if(i==Windows-1)
           {
             WindowOptimization(WindowSize*i,
                               nump,
                               AcceptanceProbability,
                               MinNumOfSimultanousPerturb,
                               MinNumOfIterations,
                               NumOfRepeatedEngToStop);
           }
          }*/
//        OUT << "\n<<<Left Top7::Move() " << endl;
}

bool Top7::IsInAlphaHelix(int Position)
{
//        OUT << "\n>>>Entered Top7::IsInAlphaHelix() " << endl;
          bool result = false;
          for(int i=0;i<AlphaHelices.size();i++)
          {
           result = result || AlphaHelices[i].has(Position);
          }
//        OUT << "\n<<<Left Top7::IsInAlphaHelix() " << endl;
          return result;
}

void Top7::AddRotamerModification(int Position)
{
          PositionRotamerPair PRP;
          PRP.Pos = Position;
          PRP.Rotamer = m->getActive(Position);
          Modifications.push_back(PRP);
}

void Top7::RestoreRotamerModification()
{
          for(int i=0;i<Modifications.size();i++)
          {
           m->setActive(Modifications[i].Pos,Modifications[i].Rotamer);
          }
}

void Top7::ClearRotamerModification()
{
          Modifications.clear();
```

```
}

void Top7::SaveState()
{
         RotamerConfig.clear();
         for(int i=0;i<nump;i++)
         {
          RotamerConfig.push_back(m->getActive(i));
         }
         CoordsConfig.clear();
         for(int i=0;i<m->totatoms;i++)
         {
          CoordsConfig.push_back(m->atoms[i].xyz);
         }
}

template <class T> void Top7::ShowTable(vector<vector<T> > vec)
{
         for(int i=0;i<vec.size();i++)
         {
          for(int j=0;j<vec[i].size()-1;j++)
          {
            cout << vec[i][j] << ",\t";
          }
          cout << vec[i][vec[i].size()-1] << endl;
          cout << "-----------------------------------" << endl;
         }
}

void Top7::RestoreState()
{
         for(int i=0;i<nump;i++)
         {
          m->setActive(i,RotamerConfig[i]);
         }
         for(int i=0;i<m->totatoms;i++)
         {
          m->atoms[i].xyz = CoordsConfig[i];
         }
}



Top7Error_BadPosition::Top7Error_BadPosition(string funcName, int Pos)
{
         stringstream ss;
         ss << "Top7:Func: "<<funcName<<" ,Pos: " << Pos << "." << endl;
         message() += ss.str();
}


#ifdef BDA_PYTHON

PyObject* MakeTop7Object(PyObject* self, PyObject* args)
{
         try
         {
          char* MoleculeName;
          char* ForceFieldName;

          if(!PyArg_ParseTuple(args,"ss",&MoleculeName,&ForceFieldName))
             return NULL;
          Top7* MyTop7 =
          new Top7(Molecules[MoleculeName],forcefields[ForceFieldName]);
          OUT << "\nWe got a new Top7  Object " << endl;
          stringstream name;
          name << "MyTop7_" << NumOfTop7Objs;
          OUT << "\nTop7Obj name: " << name.str() << endl;
          NumOfTop7Objs++;
          Top7Objects[name.str()]=MyTop7;
          return Py_BuildValue("s",name.str().c_str());
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* GreedyRotamerOptimization(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          if(!PyArg_ParseTuple(args,"s",&Top7ObjName))
             return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            ((*It).second)->GreedyRotamerOptimization();
            return Py_BuildValue("s","done");
          }
          else
          {
            OUT << "\nBad Top7 Object Name... " << endl;
          }
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}
```

```
PyObject* FlexBkBn(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int MaxNumOfPosToPerturb;
          int NumOfIterations;
          float MaxDegree;
          float DegreePrecision;
          float ThreshHold;
          float AcceptProbability;

          if
          (!
PyArg_ParseTuple(args,"siiffff",&Top7ObjName,&MaxNumOfPosToPerturb,&NumOfIterations,&MaxDegree,&DegreePrecision,&ThreshHold,&AcceptProbability))
              return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->FlexBkBn(MaxNumOfPosToPerturb,
                                   NumOfIterations,
                                   MaxDegree,
                                   DegreePrecision,
                                   ThreshHold,
                                   AcceptProbability);
          }
          else
          {
            OUT << "\nBad Top7 Object Name ..." << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* RossetaMove(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int NumPos;
          int Steps;
          float MaxDegree;
          float PrecisionDegree;
          float KTi;
          float KTf;

          if(!PyArg_ParseTuple(args,"siiffff",&Top7ObjName,
                               &NumPos,
                               &Steps,
                               &MaxDegree,
                               &PrecisionDegree,
                               &KTi,
                               &KTf)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->RossetaMove(NumPos,
                                      Steps,
                                      MaxDegree,
                                      PrecisionDegree,
                                      KTi,
                                      KTf);
          }
          else
          {
            OUT << "\nBad Top7Obj Name .... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}


PyObject* FlexBkBnAnnealing(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int MaxNumOfBkBnPosToPerturb;
          int NumOfSteps;
          float MaxDegree;
          float DegreePrecision;
          float KTi;
          float KTf;

          if(!PyArg_ParseTuple(args,"siiffff",&Top7ObjName,
                               &MaxNumOfBkBnPosToPerturb,
                               &NumOfSteps,
                               &MaxDegree,
                               &DegreePrecision,
```

```
                                &KTi,
                                &KTf)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->FlexBkBnAnnealing(MaxNumOfBkBnPosToPerturb,
                                  NumOfSteps,
                                MaxDegree,
                                DegreePrecision,
                                KTi,
                                KTf);
          }
          else
          {
            OUT << "\nBad Top7Obj Name .... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* FlexBackrubMonteCarlo(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float MaxAngle;
          float Precision;
          float KTi;
          float KTf;
          int Steps;
          int Length=3;
          float BiasProbability=0.5;
          bool domin=false;

          if(!PyArg_ParseTuple(args,"sffffi|ifi",&Top7ObjName,
                                &MaxAngle,
                                &Precision,
                                &KTi,
                                 &KTf,
                                &Steps,
                                &Length,
                                 &BiasProbability,
                               &domin)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->FlexBackrubMonteCarlo(MaxAngle,
                                Precision,
                                KTi,
                                KTf,
                                Steps,
                                Length,
                                BiasProbability,
                                domin);
          }
          else
          {
            OUT << "\nBad Top7Obj Name .... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* FlexOstrichMonteCarlo(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float MaxAngle;
          float Precision;
          float KTi;
          float KTf;
          int Steps;
          int Length=3;
          int MaxNumOfBkBnPosToPerturb=5;
          int Top7Steps=50;
          float SCProbability = 0.3;
          float SCMinProbability = 0.2;
          float BackrubProbability = 0.4;
          float Top7Probability = 0.1;

          if(!PyArg_ParseTuple(args,"sffffi|iiiffff", &Top7ObjName,
                                    &MaxAngle,
                                    &Precision,
                                    &KTi,
                                    &KTf,
                                   &Steps,
                                   &Length,
                                   &MaxNumOfBkBnPosToPerturb,
                                   &Top7Steps,
                                   &SCProbability,
```

```
                                    &SCMinProbability,
                                    &BackrubProbability,
                                    &Top7Probability)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            float SumProbability =
            SCProbability+SCMinProbability+BackrubProbability+Top7Probability;

            if(!(SumProbability<1.1 && SumProbability>0.9))
            {
              OUT << "\nSum of probabilities is not equal 1!!! " << endl;
              return Py_BuildValue("s","Bad input parameters");
            }
            else

              (*It).second->FlexOstrichMonteCarlo(MaxAngle,
                                  Precision,
                                  KTi,
                                  KTf,
                                  Steps,
                                  Length,
                                  MaxNumOfBkBnPosToPerturb,
                                  Top7Steps,
                                  SCProbability,
                                  SCMinProbability,
                                  BackrubProbability,
                                  Top7Probability);
          }
          else
          {
            OUT << "\nBad Top7Obj Name .... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* DisruptStructure(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int MaxNumOfPosToPerturb;
          int NumOfIterations;
          float MaxDegree;
          float DegreePrecision;
          float EnergyStepIncrease;
          float ThreshHold;
          float AcceptProbability;

          if
          (!PyArg_ParseTuple(args,"siifffff",&Top7ObjName,&MaxNumOfPosToPerturb,&NumOfIterations,&MaxDegree,&DegreePrecision,&EnergyStepIncrease,&Thresh-
Hold,&AcceptProbability))
              return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->DisruptStructure(MaxNumOfPosToPerturb,
                                NumOfIterations,
                                MaxDegree,
                                DegreePrecision,
                                EnergyStepIncrease,
                                ThreshHold,
                                AcceptProbability);
          }
          else
          {
            OUT << "\nBad Top7Obj Name.... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* ShuffleBkBn(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int MaxNumOfPosToPerturb;
          int NumOfIterations;
          float MaxDegree;
          float DegreePrecision;

          if(!PyArg_ParseTuple(args,"siiff",&Top7ObjName,&MaxNumOfPosToPerturb,&NumOfIterations,&MaxDegree,&DegreePrecision)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
```

```
            (*It).second->ShuffleBkBn(MaxNumOfPosToPerturb,
                                      NumOfIterations,
                                      MaxDegree,
                                      DegreePrecision);
          }
          else
          {
            OUT << "\nBad Top7Obj Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* TinyGrid(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float Degree;
          if(!PyArg_ParseTuple(args,"sf",&Top7ObjName,&Degree)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->TinyGrid(Degree);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* TinyGrid3_Move_MPI(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int pos1,pos2,pos3;
          float deg1, deg2, deg3;
          if(!PyArg_ParseTuple(args,"sifififf",&Top7ObjName,&pos1,&deg1,&pos2,&deg2,&pos3,&deg3)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->TinyGrid3_Move_MPI(pos1,deg1,pos2,deg2,pos3,deg3);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}


PyObject* GradientBestPhiPsiMPI(PyObject* self, PyObject* args)
{
/*-----------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=true;

          if(!PyArg_ParseTuple(args,"sff|i",&Top7ObjName,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->GradientBestPhiPsiMPI(MaxDegree,DegreePrecision,ExcludeAlphaHelices);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
```

```
         }
         return NULL;
}

PyObject* GradientBestPhiPsi(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          float MaxDegree;
          float DegreePrecision;
          bool RotamerOptimization=true;
          int RecordSnapShots=false;

          if(!PyArg_ParseTuple(args,"sff|ii",&Top7ObjName,&MaxDegree,&DegreePrecision,&RotamerOptimization,&RecordSnapShots)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
//          if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
//            return Py_BuildValue("s","PassedBy_done");
            (*It).second->GradientBestPhiPsi(MaxDegree,DegreePrecision,RotamerOptimization,RecordSnapShots);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* GradientBestPhiPsiMPI_HighPerformance(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int StartPos;
          int StopPos;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=false;
          bool RecordSnapShots=false;

          if(!PyArg_ParseTuple(args,"siiff|ii",&Top7ObjName,&StartPos,&StopPos,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices,&RecordSnapShots)) return
NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            StartPos--;
            StopPos--;
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->GradientBestPhiPsiMPI_HighPerformance(StartPos,StopPos,MaxDegree,DegreePrecision,ExcludeAlphaHelices,RecordSnapShots);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* GradientBestPhiPsiBackrubMPI_HighPerformance(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int StartPos;
          int StopPos;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=false;
          int BackrubLength=3;
          float BackrubDegree=4.0;
          float BackrubPrecision=0.02;

          if(!PyArg_ParseTuple(args,"siiff|iiff",&Top7ObjName,&StartPos,&StopPos,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices,&BackrubLength,&Backrub-
Degree,&BackrubPrecision)) return NULL;
```

```
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            StartPos--;
            StopPos--;
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->GradientBestPhiPsiBackrubMPI_HighPerformance(StartPos,StopPos,MaxDegree,DegreePrecision,ExcludeAlphaHelices,BackrubLength,Back-
rubDegree,BackrubPrecision);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* AllAtOnceGradientMPI_HighPerformance(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int StartPos;
          int StopPos;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=false;
          bool RotamerOptimization=true;

          if(!PyArg_ParseTuple(args,"siiff|ii",&Top7ObjName,&StartPos,&StopPos,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices,&RotamerOptimization))
return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            StartPos--;
            StopPos--;
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->AllAtOnceGradientMPI_HighPerformance(StartPos,StopPos,MaxDegree,DegreePrecision,ExcludeAlphaHelices,RotamerOptimization);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* AllAtOnceGradientMPI_56Nodes(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float Degree;
          int Steps=1;
          int ExcludeAlphaHelices=false;

          if(!PyArg_ParseTuple(args,"sf|ii",&Top7ObjName,&Degree,&Steps,&ExcludeAlphaHelices)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->AllAtOnceGradientMPI_56Nodes(Degree,Steps,ExcludeAlphaHelices);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* AllAtOnceGradientMPI_xNodes(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          PyObject* PyDegrees;
          vector<float> Degrees;
          int ExcludeAlphaHelices=false;

          if(!PyArg_ParseTuple(args,"sO|i",&Top7ObjName,&PyDegrees,&ExcludeAlphaHelices)) return NULL;
```

```
          Degrees = pyparse_floatlist(PyDegrees);
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->AllAtOnceGradientMPI_xNodes(MPI_SIZE/8,Degrees,ExcludeAlphaHelices);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* SomeAtOnceGradientMPI_xNodes(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          PyObject* PyPositions;
          PyObject* PyDegrees;
          vector<int> Positions;
          vector<float> Degrees;

          if(!PyArg_ParseTuple(args,"sOO",&Top7ObjName,&PyPositions,&PyDegrees)) return NULL;

          Positions = pyparse_intlist(PyPositions);
          Degrees = pyparse_floatlist(PyDegrees);
          for(int i=0;i<Positions.size();i++)
            OUT << Positions[i] << ",";

          OUT << endl;

          for(int i=0;i<Degrees.size();i++)
            OUT << Degrees[i] << ",";


          for(int i=0;i<Positions.size();i++)
          {
            Positions[i]--;
          }
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->SomeAtOnceGradientMPI_xNodes(Positions,Degrees);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* BestResidueGradientMPI_HighPerformance(PyObject* self, PyObject* args)
{
/*-----------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int StartPos;
          int StopPos;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=false;

          if(!PyArg_ParseTuple(args,"siiff|i",&Top7ObjName,&StartPos,&StopPos,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            StartPos--;
            StopPos--;
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->BestResidueGradientMPI_HighPerformance(StartPos,StopPos,MaxDegree,DegreePrecision,ExcludeAlphaHelices);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}
```

```
PyObject* SomeAtOnceGradientMPI_HighPerformance(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int StartPos;
          int StopPos;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=false;
          vector<int> Positions;
          PyObject* _Positions;
          if(!PyArg_ParseTuple(args,"siiffiO",&Top7ObjName,&StartPos,&StopPos,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices,&_Positions)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            StartPos--;
            StopPos--;
            Positions = pyparse_intlist(_Positions);
            for(int i=0;i<Positions.size();i++)
            {
              Positions[i]--;
            }
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->SomeAtOnceGradientMPI_HighPerformance(StartPos,StopPos,MaxDegree,DegreePrecision,ExcludeAlphaHelices,Positions);
          }
          else
          {
            OUT << "\nBad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}


PyObject* StochasticGradientBestPhiPsiMPI(PyObject* self, PyObject* args)
{
/*------------comment-----------
         number of ppn=8 nodes required which we call N is:
         NumAngles=ceil(MaxDegree/DegreePrecision)
         N = ceil(((2*NumAngles+1)^2)/8)
*/
         try
         {
          char* Top7ObjName;
          int NumOfIterations;
          int MaxNumOfPositions;
          float MaxDegree;
          float DegreePrecision;
          bool ExcludeAlphaHelices=true;

          if(!PyArg_ParseTuple(args,"siiff|i",&Top7ObjName,&NumOfIterations,&MaxNumOfPositions,&MaxDegree,&DegreePrecision,&ExcludeAlphaHelices)) return
NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            if((*It).second->ByPassRank(MaxDegree,DegreePrecision))
              return Py_BuildValue("s","PassedBy_done");
            (*It).second->StochasticGradientBestPhiPsiMPI(NumOfIterations,MaxNumOfPositions,MaxDegree,DegreePrecision,ExcludeAlphaHelices);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* MonteCarloGradientBestPhiPsiMPI(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float Degree;
          float KTi, KTf;
          int Steps;
          bool fixedRange = false;
          bool recordSnapshots=false;
          char* Filename=NULL;

          if(!PyArg_ParseTuple(args,"sfffi|iis",&Top7ObjName,&Degree,&KTi,&KTf,&Steps,&fixedRange,&recordSnapshots,&Filename)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->MonteCarloGradientBestPhiPsiMPI(Degree,KTi,KTf,Steps,fixedRange,recordSnapshots,Filename);
          }
          else
```

```
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* StochasticPerturbationGridRefinement(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          PyObject* PyDegrees;
          PyObject* PyPrecision;
          int NumOfStructs;
          float MaxPerturbationAngle;
          char* path;
          char* name;
          int Rank;
          int UseRamaCode=1;
          int UniformRama=0;
          int RecordSnapShots=false;
          bool Code2=false;
          bool doGRID = true;

          vector<float> degrees;
          vector<float> precision;

          if(!PyArg_ParseTuple(args,"sOOifssiiiii|i",
                               &Top7ObjName,
                               &PyDegrees,
                               &PyPrecision,
                               &NumOfStructs,
                               &MaxPerturbationAngle,
                               &path,
                               &name,
                               &Rank,
                               &UseRamaCode,
                               &UniformRama,
                               &RecordSnapShots,
                               &Code2,
                               &doGRID)) return NULL;

          degrees = pyparse_floatlist(PyDegrees);
          precision = pyparse_floatlist(PyPrecision);

          OUT << "\ndegrees: " << endl;
          for(int i=0;i<degrees.size();i++)
            OUT << degrees[i] << "," ;
          OUT << "\nprecision: " << endl;
          for(int i=0;i<precision.size();i++)
            OUT << precision[i] << "," ;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->StochasticPerturbationGridRefinement(degrees,
                                               precision,
                                               NumOfStructs,
                                               MaxPerturbationAngle,
                                               path,
                                               name,
                                               Rank,
                                               UseRamaCode,
                                               UniformRama,
                                               RecordSnapShots,
                                               Code2,
                                               doGRID);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* StochasticPerturbationBackrubRefinement(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float MaxAngle;
          float Precision;
          float MaxPerturbationAngle;
          float KTi;
          float KTf;
          int Steps;
          int Length;
          float BiasProbability;
          bool domin;
          int NumOfStructures;
          char* path;
          char* name;
          int Rank;
```

```
          bool RecordSnapShots;

          if(!PyArg_ParseTuple(args,"sfffffiifiissii",
                               &Top7ObjName,
                               &MaxAngle,
                               &Precision,
                               &MaxPerturbationAngle,
                               &KTi,
                               &KTf,
                               &Steps,
                               &Length,
                               &BiasProbability,
                               &domin,
                               &NumOfStructures,
                               &path,
                               &name,
                               &Rank,
                               &RecordSnapShots)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->StochasticPerturbationBackrubRefinement(MaxAngle,
                                                Precision,
                                                MaxPerturbationAngle,
                                                KTi,
                                                KTf,
                                                Steps,
                                                Length,
                                                BiasProbability,
                                                domin,
                                                NumOfStructures,
                                                path,
                                                name,
                                                Rank,
                                                RecordSnapShots);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* StochasticRamaPerturbationDecoyGeneration(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          int NumberOfStructures;
          char* path;
          char* name;
          float PerturbationRate = 0.1;

          if(!PyArg_ParseTuple(args,"sissf",
                               &Top7ObjName,
                               &NumberOfStructures,
                               &path,
                               &name,
                               &PerturbationRate)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->StochasticRamaPerturbationDecoyGeneration(NumberOfStructures,
                                                path,
                                                name,
                                                PerturbationRate);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* ReplicaExchangeMonteCarloGRID(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          PyObject* PyDegrees;
          vector<float> degrees;
          PyObject* PyPrecisions;
          vector<float> precisions;
          int rounds;
          int steps_sync;
          PyObject* PykTs;
          vector<float> kTs;

          if(!PyArg_ParseTuple(args,"sOOiiO",
                               &Top7ObjName,
                               &PyDegrees,
```

```
                            &PyPrecisions,
                            &rounds,
                            &steps_sync,
                            &PykTs)) return NULL;

          degrees = pyparse_floatlist(PyDegrees);
          precisions = pyparse_floatlist(PyPrecisions);
          kTs = pyparse_floatlist(PykTs);

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->ReplicaExchangeMonteCarloGRID(degrees,
                                        precisions,
                                        rounds,
                                        steps_sync,
                                        kTs);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}

PyObject* SimulatedAnnealingReplicaExchangeMonteCarloGRID(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float kT_high_i;
          float kT_high_f;
          float kT_low_i;
          float kT_low_f;
          float degree_scale;
          int rounds;
          int rounds_of_rounds;
          int steps_sync;

          if(!PyArg_ParseTuple(args,"sfffffiii",
                            &Top7ObjName,
                            &kT_high_i,
                            &kT_high_f,
                            &kT_low_i,
                            &kT_low_f,
                            &degree_scale,
                            &rounds,
                            &rounds_of_rounds,
                            &steps_sync)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->SimulatedAnnealingReplicaExchangeMonteCarloGRID(kT_high_i,
                                                  kT_high_f,
                                                  kT_low_i,
                                                  kT_low_f,
                                                  degree_scale,
                                                  rounds,
                                                  rounds_of_rounds,
                                                  steps_sync);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
          OUT << e.Message() << endl;
          cleanexit();
         }
         return NULL;
}


PyObject* RandomStructurePerturbation(PyObject* self, PyObject* args)
{
         try
         {
          char* Top7ObjName;
          float MaxAngle;
          int Sequential=true;

          if(!PyArg_ParseTuple(args,"sfi",&Top7ObjName,&MaxAngle,&Sequential)) return NULL;

          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->RandomStructurePerturbation(MaxAngle,Sequential);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
         }
         catch(cpdsError& e)
         {
```

```
          OUT << e.Message() << endl;
          cleanexit();
          }
          return NULL;
}

PyObject* PhiPsiRepresentation(PyObject* self, PyObject* args)
{
          try
          {
          char* Top7ObjName;
          if(!PyArg_ParseTuple(args,"s",&Top7ObjName)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          vector<float> PhiPsiList;
          if(It!=Top7Objects.end())
          {
            PhiPsiList = (*It).second->PhiPsiRepresentation();
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return pyformat_floatlist(PhiPsiList);
          }
          catch(cpdsError& e)
          {
          OUT << e.Message() << endl;
          cleanexit();
          }
          return NULL;
}


PyObject* StochasticPDBFragmentRefinement(PyObject* self, PyObject* args)
{
          try
          {
          char* Top7ObjName;
          if(!PyArg_ParseTuple(args,"s",&Top7ObjName)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->StochasticPDBFragmentRefinement();
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
          }
          catch(cpdsError& e)
          {
          OUT << e.Message() << endl;
          cleanexit();
          }
          return NULL;
}

PyObject* RamaStructurePerturbation(PyObject* self, PyObject* args)
{
          try
          {
          char* Top7ObjName;
          int Uniform = 0;
          if(!PyArg_ParseTuple(args,"s|i",&Top7ObjName,&Uniform)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->RamaStructurePerturbation(Uniform);
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
          }
          catch(cpdsError& e)
          {
          OUT << e.Message() << endl;
          cleanexit();
          }
          return NULL;
}

PyObject* GreedyMinimization(PyObject* self, PyObject* args)
{
          try
          {
          char* Top7ObjName;

          if(!PyArg_ParseTuple(args,"s",&Top7ObjName)) return NULL;
          map<string,Top7*>::const_iterator It = Top7Objects.find(Top7ObjName);
          if(It!=Top7Objects.end())
          {
            (*It).second->GreedyMinimization();
          }
          else
          {
            OUT << "\n Bad Top7Name ... " << endl;
          }
          return Py_BuildValue("s","done");
          }
          catch(cpdsError& e)
          {
          OUT << e.Message() << endl;
          cleanexit();
          }
```

```
          return NULL;
}

PyObject* AlignTwoMoleculesRMSD(PyObject* self, PyObject* args)
{
          //this manipulates coordinates of the second molecule...
          try
          {
           char* MoleculeName1;
           char* MoleculeName2;
           if(!PyArg_ParseTuple(args,"ss",&MoleculeName1,&MoleculeName2)) return NULL;
           molecules* M1 = get_molecules(MoleculeName1);
           molecules* M2 = get_molecules(MoleculeName2);
           vector<keyrange> k1 = M1->getActivesAll();
           vector<keyrange> k2 = M2->getActivesAll();
           float rmsdValue = alignAndRMSD(M1->atoms,M2->atoms,k1,k2);
           return Py_BuildValue("f",rmsdValue);
          }
          catch(cpdsError& e)
          {
           OUT << e.Message() << endl;
           cleanexit();
          }
          return NULL;
}


#endif
```