AB INITIO NANOSTRUCTURE DETERMINATION

By

Saurabh Gujarathi

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Physics - Doctor of Philosophy

2014

ABSTRACT

AB INITIO NANOSTRUCTURE DETERMINATION

By

Saurabh Gujarathi

Reconstruction of complex structures is an inverse problem arising in virtually all areas of science and technology, from protein structure determination to bulk heterostructure solar cells and the structure of nanoparticles. This problem is cast as a complex network problem where the edges in a network have weights equal to the Euclidean distance between their endpoints. A method, called Tribond, for the reconstruction of the locations of the nodes of the network given only the edge weights of the Euclidean network is presented. The timing results indicate that the algorithm is a low order polynomial in the number of nodes in the network in two dimensions. Reconstruction of Euclidean networks in two dimensions of about one thousand nodes in approximately twenty four hours on a desktop computer using this implementation is done. In three dimensions, the computational cost for the reconstruction is a higher order polynomial in the number of nodes and reconstruction of small Euclidean networks in three dimensions is shown. If a starting network of size five is assumed to be given, then for a network of size 100, the remaining reconstruction can be done in about two hours on a desktop computer. In situations when we have less precise data, modifications of the method may be necessary and are discussed.

A related problem in one dimension known as the Optimal Golomb ruler (OGR) is also studied. A statistical physics Hamiltonian to describe the OGR problem is introduced and the first order phase transition from a symmetric low constraint phase to a complex symmetry broken phase at high constraint is studied. Despite the fact that the Hamiltonian is not

disordered, the asymmetric phase is highly irregular with geometric frustration. The phase diagram is obtained and it is seen that even at a very low temperature T there is a phase transition at finite and non-zero value of the constraint parameter γ/μ. Analytic calculations for the scaling of the density and free energy of the ruler are done and they are compared with those from the mean field approach. A scaling law is also derived for the length of OGR, which is consistent with Erdös conjecture and with numerical results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Reconstruction of heterogeneous and complex systems using pair correlation functions or pair distance information, is a problem that arises in many branches of materials physics [1, 2], in biology [3, 4] and also in a variety of engineering applications [5]. We distinguish between two problems (i) where the objective is to find a statistical characterization of a heterogeneous system that is consistent with experimental information. In these cases the reconstruction is not unique, but instead generates an ensemble of structures that are on average consistent with the data. Reverse Monte Carlo methods [6] for the atomic structure of glasses and simulated annealing methods for a range of heterogeneous materials are in this class. Large samples are often used and the system is highly underconstrained as there are many more degrees of freedom in the model for the atom locations than there is information in the data. (ii) A related but significantly different problem is where we seek to reconstruct a specific, unique, network or structure. The amount of information in the data must suffice to constrain the degrees of freedom in the structure. This problem can be hard for structures with only ten to hundreds of atoms or components. Uniqueness is lost when the model has too many degrees of freedom as compared to the available data. This unique structure problem is the focus of our study. Surprisingly, we find that it is possible to efficiently reconstruct large complex structures in two dimensions, given only Euclidean distance information.

Crystallography represents the gold standard for structure determination and provided methods to overcome the phase problem are implemented and if there are no homometric

variants [7], provides a unique crystal structure. When crystals are not available, but a unique structure is still the objective, new methods are required. One successful approach is the determination of protein structure in solution that may be found by using pair distance information extracted from NOESY NMR data [3, 4, 8, 9, 10]. Two other approaches are emerging. The first is determination of the structure of individual nanoparticles using lensless imaging algorithms [11, 12, 13, 14]. The second approach is to extract a list of interatomic distances from scattering data and to solve a new inverse problem to find the atom locations. Here we present a highly efficient method to solve the latter inverse problem for the case of complex or random point sets in two dimensions.

As discussed recently in [15, 16, 17] by Torquato and collaborators, reconstruction of heterogeneous systems in general requires multipoint correlation functions. However pair correlations are by far the most readily available structural data for heterogeneous materials as they are found by a Fourier transform of elastic electron, x-ray or neutron scattering data collected, for example, at national facilities. There is thus a strong motivation to find methods to determine the extent to which we can reconstruct heterogeneous systems using pair information only. The most fundamental pair information is the list of distances between points or atoms in a structure, reducing the problem to an inverse problem, namely: Given a set of interatomic distances find the location of the atoms, up to global rotations and translations of the structure. This pair distance inverse problem (PD-IP) may be interpreted as a complex network reconstruction problem where the edge weights are equal to the Euclidean distances between nodes in the network. Moreover, it has been recently shown that a list of pair distances may be extracted from scattering data using the pair distribution function (PDF) method [18].

The PD-IP is central to determining protein structure from NMR data, however there

are vital differences between the problem we study and the NMR PD-IP problem. The most important difference is that the list of residues or sequence of a protein is known enabling mutation and other experiments to be carried out to specify the points between which each distance lies. This leads to the *assigned* pair distance inverse problem (APD). In contrast, the problems concerning materials and most heterogeneous media, the pair distances are not assigned making the inverse problem significantly harder as there less information in the data. This is the unassigned pair distance inverse problem (UPD). In fact, APD algorithms for reconstruction of atom locations from precise distances is known to be easy, being of order the number of atoms in the structure (N). However the NMR problem is plagued by uncertainties in the experimentally determined interatomic distances with experimental imprecisions typically of order 25% or higher [19]. The problem of finding protein structure from NMR data is then best treated using loose restraints rather than hard distance constraints. The energy landscape of the APD with loose constraints has many of the features of spin glass problems leading to the belief that NMR structure determination using loose assigned distances (loose APD) is computationally hard [20, 1, 21].

In almost all other Euclidean network reconstruction problems the distances are not assigned, as we do not know which nodes lie at the end of each distance. For example, the pair distribution function method is used for the analysis of the local structure of nanoparticles and complex materials. In many complex materials, such as high performance thermoelectric materials [22], high temperature superconductors [23] and manganites [24], crystalline order and heterogeneous local distortions co-exist so that crystallographic and PDF methods are complementary. Crystallography finds the average structure and the PDF the local structure [25, 26]. The pair distribution function gives a direct measure of the list of interatomic distances arising in the local structure, however the endpoints of the distances are not known

so we face a computationally challenging UPD problem [27].

Recently, in collaboration with Professor Billinge's group, we developed efficient algorithms for the UPD problem for cases where there is significant symmetry in the structure, including C_{60} and a range of crystal structures. In those cases we found two types of algorithm worked well, genetic algorithms and a novel algorithm called Liga [28, 29, 30].

Liga works well while reconstructing structures having high symmetry. But for solving structures with hundreds of points, Liga fails miserably for low symmetry problems such as random point sets, due to the fact that there are a large number of unique pair distances in random structures. They thus fail for the general problem of complex Euclidean networks. Here we present an algorithm that is specifically designed for reconstructing complex Euclidean networks where there are a large number of unique distances.

A formal statement of the UPD problem is as follows. We are given a list of distances $\{d_l\}$, $l = 1...M$, between points in a D-dimensional Euclidean space. Our task is to find the co-ordinates of the points $\{\vec{r}_i\}$, $i = 1, ..., N$ such that the distance between every pair of points $|\vec{r}_i - \vec{r}_j| = r_{ij}$ is a member of the distance list $\{d_l\}$. Moreover we require that every distance in the list $\{d_l\}$ occurs for some pair of points (i, j) in the structure.

The only inputs to the Euclidean network reconstruction algorithm described below are the number of points in the network N and a list of $N(N-1)/2$ Euclidean distances. Physically, it is useful to think of the Euclidean distances as natural lengths of Hookian springs, l^0_{ij} so that we may define an energy function,

$$E(\{l^0_{ij}\}) = \sum_{ij} k_{ij}(l_{ij} - l^0_{ij})^2 \tag{1.1}$$

In the ideal UPD problem the distance list is known precisely, but we don't know the

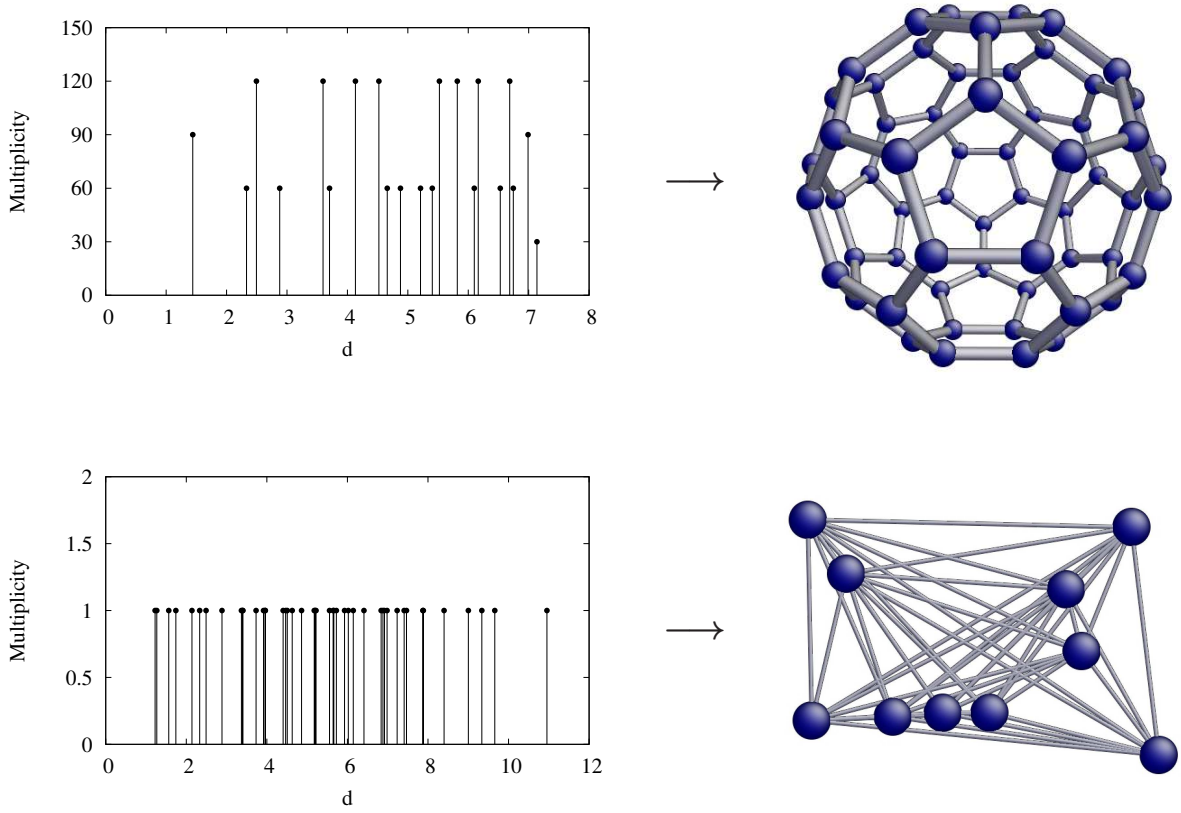

Figure 1.1: (color online) Simple examples of structures found from Euclidean distance lists. The figures on the left are plots of the distance lists for: a) (top) a C_{60} fullerene that has a degenerate distance list, and b) (bottom) a random set of 10 points in the plane that has a non-degenerate distance list. The fullerene has a total of 1770 interatomic distances, but only 21 unique distances. The random point set has, with high probability, 45 unique distances. The multiplicity is on the vertical axis while the distance is on the horizontal axis (in arbitrary units). The figures on the right hand side are solutions to the inverse problem found using the Liga algorithm (fullerene) and Tribond (random point set) to find the structure from the given distance lists, without the use of any other information. For the random point set all interatomic distances are drawn in the figure. For clarity only the nearest neighbor bonds are drawn in the fullerene case. In this study, the distance lists are taken from the known structure and then we try to solve the inverse problem using only the distance list. In the real world, the structure is unknown and the distance lists are derived from experiments, particularly x-ray and neutron scattering data.

mapping or assignment $d_l \rightarrow l_{ij}^0$. This is the precise UPD. In the precise UPD the key computational difficulty is to find this mapping or assignment of d_l to l_{ij}. If the correct assignment is found the energy (1) is zero, while wrong assignments lead to stretched or compressed springs and a finite energy. Strategies to treat the loose UPD problem are discussed in Section IV.

In the assigned pair distance inverse problem (APD), when the inter point distances are known precisely, the problem can be solved in polynomial time. A problem can be solved in polynomial time (P) if the computational cost for an input of size N is $\mathcal{O}(N^k)$, where k is a non negative integer. When there are uncertainties in the experimentally determined inter atomic distances, the problem is computationally hard and is NP [31, 32]. NP stands for non-deterministic polynomial. For problems in NP, the solution can be verified in polynomial time. Please note that problems in P are also in NP.

For the APD problem, the method for solving the precise case was the foundation for solving the problem with imprecise distances. Here, we present an algorithm that solves the unassigned problem (UPD) in the precise case and we hope that it will offer insights that lead to techniques for solving the imprecise case.

Two examples of this problem are presented in Fig. 1.1. Fig. 1.1a (top) presents an example of a degenerate distance list, typical of structures which have high symmetry, while Fig. 1.1b (bottom) is an example of a random point set where all distances are, with high probability, unique. Since the number of Euclidean distances is $M = N(N-1)/2$, a search over all permutations of the distances to find the correct assignment of d_l to l_{ij} requires a computational time proportional to the factorial of M, so that $\tau \sim M!$. This is worse than exponential time complexity and is also a very poor way to proceed. The Tribond algorithm is presented in this work and is shown to have a polynomial complexity.

A related problem in one dimension is the optimal Golomb ruler. Common rulers have

marks which are equally spaced so that you can measure any distance between 1 and the length of the ruler by placing an object between any two marks with the desired distance. With a common ruler (Fig. 1.2) one can measure the distance 4 in multiple ways, say by placing the object between the marks 0 and 4 or between marks 1 and 5. Golomb rulers can be thought of as a special kind of rulers in which every distance between two marks is different from all others. For example if there is a mark at position 1 and 5, then no other pair of marks must be separated by a distance of 4. From this definition, we can see that a common ruler with more than 2 marks is not Golomb. Using a ruler with the following marks 0, 1, 4, 9, 11 (Fig. 1.3) we can measure the distances $\{1, 2, 3, 4, 5, 7, 8, 9, 10, 11\}$ by using only one pair of marks, therefore the Golomb property is satisfied. Rulers which have the smallest possible length for a given number of marks are called optimal Golomb rulers (OGR).

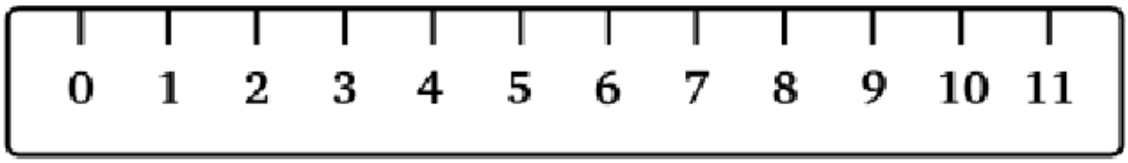

Figure 1.2: A common ruler

Maximizing irregularity and constructing optimal Golomb rulers are closely related [33]. Because of this property, Golomb rulers have applications in a wide variety of fields. Some of the real world applications include x-ray crystallography [34, 35, 36, 7], radio systems [37], radio astronomy [38, 39, 40, 41, 42, 43, 44, 45] and missile guidance [46].

Stated mathematically, a Golomb ruler is a set of non-negative integers with the property

Figure 1.3: A Golomb ruler

that the differences between the integers are all distinct. To be concrete, consider a set of n integer markers $m_1, m_2, ..., m_n$, and their associated distances $d_{ij} = |m_i - m_j|$. By convention, the first mark is set at position zero ($m_1 = 0$) so the length of ruler is equal to m_n. A Golomb ruler is a set $\{m_i\}$ satisfying the constraint that all distances $d_{i<j}$ are distinct. An optimal Golomb ruler is the shortest marker set (i.e. smallest m_n) satisfying the Golomb ruler constraints.

Dimitromanolakis [47] showed that Golomb rulers are equivalent to an old problem of Sidon sets [48, 49, 50]. A Sidon set is a subset of the set $A = 1, ..., N$ of positive integers such that for every two elements a_i, a_j $(a_i \leq a_j)$ of the set, the sum $a_i + a_j$ is different from all other sums. Sidon sets and Golomb rulers are equivalent as a set having distinct differences between any two elements will also have distinct sums and vice versa. Finding an optimal Golomb ruler is equivalent to finding a Sidon set with the maximum number of elements. This also gives us a tight upper bound on the length of a optimal Golomb ruler m_n at large n, $m_n \leq n(n+1)$. Golomb rulers can undergo simple similarity transformations, translation and multiplication that lead to new rulers which also have the Golomb property. Golomb rulers have a two fold degeneracy, with the two degenerate states transforming into each other by spatial reflection, i.e. $m_i \rightarrow m_n - m_i$. However, a Golomb ruler cannot have

reflection symmetry as it would lead to repeat distances.

There does not exist a closed form formula to generate optimal Golomb rulers. Algebraic constructions like the Bose-Chowla construction [51] and the Ruzsa construction [52, 53] are used to generate rulers, most of these rulers turned out to be optimal. Hence the rulers generated by these methods are called near-optimal rulers. The optimality proofs of such constructions can only be made with exhaustive search methods.

Finding an optimal Golomb ruler consists of two parts. First we must verify that the proposed ruler is Golomb (polynomial complexity) and then check that it has the shortest possible length. Although it has not been proved to be NP-hard, it is believed that no polynomial time algorithm exists for this problem [54]. Problems are said to be NP-Hard when the solution cannot even be verified in polynomial time, let alone coming up with a solution in polynomial time. The largest OGR found as of July 2014, has $n = 26$ with $m_n = 492$. Small rulers with marks upto $n = 18$ were relatively easy to obtain [55, 56]. After that exhaustive search is being used in addition to some heuristics and some clever ways of eliminating bad candidates [57, 58] to solve this problem. The 19 mark ruler was obtained using a computer search using 36,000 CPU hours [57]. The search for larger rulers is now being carried out using a distributed computer network consisting of thousands of computers and takes years to complete [59].

In the past decade statistical physics approaches to combinatorial optimization problems have provided new theoretical insights and improved algorithms, particularly near the phase transitions [60, 61, 62, 63]. Examples include random K-SAT [64], coloring and maximum independent set on random graphs or vertex cover [65, 66]. All NP-complete problems [67] can be mapped using lattice gas approach. Problems are said to be NP-Complete when the solution can be verified in polynomial time, but it is not possible to come up with a solution

in polynomial time. The problem of finding the optimal Golomb ruler (OGR) is to find the shortest ruler (with length L) which has a given number of marks (m). Also, all the interpoint distances in the ruler should be unique. In this study a lattice gas model is introduced for the OGR problem and its phase behaviour is studied. The OGR problem is different than most models studied so far as it's associated statistical physics model is not disordered, being defined on regular lattices. Nevertheless the OGR solutions are highly irregular with frustration of distance geometry constraints being the physical origin of complex aperiodic ground states and glass like behavior, as is familiar in geometrically frustrated problems [68, 69] arising in other contexts. The OGR problem is also an example of geometrical frustration. If we are trying to solve for a density which is higher than the OGR density then there is geometrical frustration. There is no solution for which there exists a zero energy ground state. (L,m) pairs for the optimal Golomb ruler have the lowest energy and the solution is degenerate due to reflection symmetry. From numerical calculations it is seen that the Golomb lattice gas exhibits spontaneous symmetry breaking from a low constraint reflection symmetric phase to a high constraint phase that does not have reflection symmetry. Asymptotic calculations are also done and the results for the scaling behavior and phase boundary are compared with those from numerical calculations.

The layout of this thesis is as follows. Chapter 2 is about the Liga algorithm and the extension of the algorithm in order to solve crystal structures. The work on the Liga algorithm is prior work by my colleague Pavol Juhas. The work on the extended algorithm was published in the Journal of Applied Crystallography and I was a co-author. Chapter 3 describes the Tribond algorithm in two dimensions and the corresponding paper is ready for submission to Physics Review E. Chapter 4 deals with the algorithm in three dimensions and the corresponding paper is in preparation for Physics Review E. In chapter 5, the work on

the optimal Golomb ruler (OGR) is presented, which will be resubmitted to Physics Review Letters. In chapter 6, the concluding remarks are made. I will also be writing a review paper, which will cover Liga and Tribond algorithms, that will be submitted to Discrete Applied Math journal. In the appendix, all the code for the Tribond 2D and 3D algorithm is given.

Chapter 2

The Liga algorithm

2.1 Algorithm details

Inspired by the Spanish soccer league (La Liga), Liga uses a combination of ideas from dynamic programming with backtracking and tournaments. Reconstruction is done by building up small divisions of feasible substructures. Each division or level has substructures with the same number of atoms and is labeled by the the number of atoms in the substructures. The algorithm has tournaments where substructures in a division compete with each other to gain atoms and move to a higher division (promotion). The structures that lose have some their atoms removed and they fall to a lower division (relegation). Fig. 2.1 gives a depiction of this algorithm. The cost function for the algorithm is based on the closeness of the target and model distance lists. Each atom has an individual cost that is based on its contribution to the total cost for the substructure.

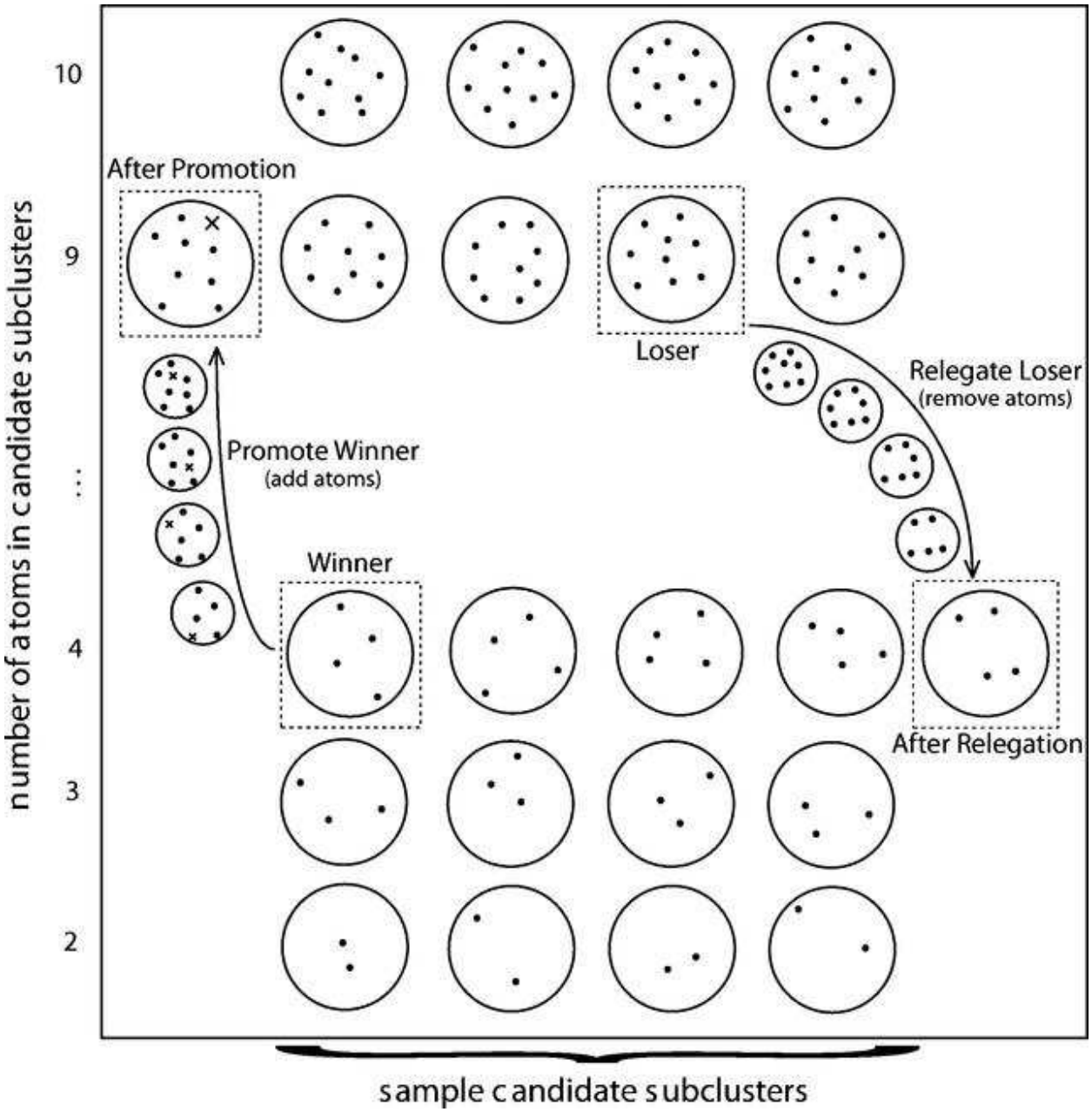

Figure 2.1: An example of promotion and relegation in Liga for N=10 and four structures at each level. The algorithm is at level 4, where a winner structure is randomly selected with probability equal to the reciprocal of its cost. The winner attempts to add as many candidate points (denoted by 'x') with a low cost as possible. In this example, the winner can add five points and gets promoted to the 9th level. A loser structure is randomly selected from the 9th level, it loses five of its atoms and is relegated to the 4th level. The choice of the losing structure and the points that are removed in relegation are done randomly with a probability equal to the cost.

In a tournament, a "winner" is randomly chosen from the candidates with probability

equal to the reciprocal of its cost. A "loser" is randomly chosen with probability equal to its cost. This operation is carried out for both points and substructures. Whenever atoms are added or removed from a structure, the target distance list is updated so that the new distances are no longer available.

In order to add an atom to an existing substructure, a pool of candidate points is generated. Three methods are used to find candidate points: line trials, triangle trials and pyramid trials. In line trials, two winner atoms are chosen along with a distance from the target list. The first atom is chosen as the anchor, while the second one is used for the direction and this type of trial leads to two candidate points. In planar trials, three winner atoms are chosen. The first two form the base of the triangle, while the third defines the plane of the triangle. Two distances are selected from the target list and they are used to construct a triangle vertex. After reflecting this vertex along the X and the Y axis, four candidate points are generated. In pyramid trials, three winner atoms are chosen and they form the base for the pyramid. The apex point for the pyramid is generated by using 3 randomly chosen lengths from the target list. There are 12 candidate points generated at the end of this trial, as there are 3! ways of assigning 3 distances to 3 atoms and also the 2 possible choices (positive and negative) for the Z coordinate. The above trials are carried out for a specified number of times to generate a pool of candidate points that can be added to the substructure. With each point is associated a cost of addition to the substructure. A winner atom is chosen from the pool and added. The cost for the remaining points in the pool is recalculated with respect to the new structure and if there are any candidates whose cost is low, a new winner is selected and added to the structure. This can speed up the reconstruction significantly.

Once the promotion and relegation has been carried out at a division, the algorithm moves to the next division and repeats the process. The Liga algorithm starts at level 0 and

systematically goes through the N levels and populates them with candidates. A traversal through all the levels is called a season. After the algorithm reaches the Nth division, and if there is a candidate whose cost is below a user defined threshold, that candidate is declared as the solution. Else, the algorithm starts a new season from the lowest division and continues the search.

Liga succeeds in reconstructing a lot of structures like platonic solids (Fig. 2.2), lattice structures (Fig. 2.3), Lennard Jones clusters (Fig. 2.4) and C_{60} buckyball (Fig. 2.5).

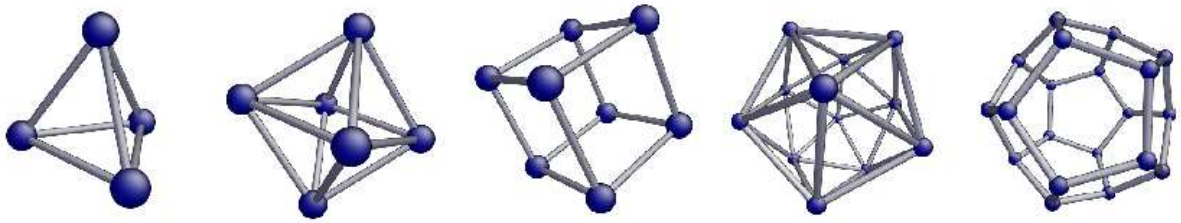

Figure 2.2: Reconstruction of various platonic solids using Liga.

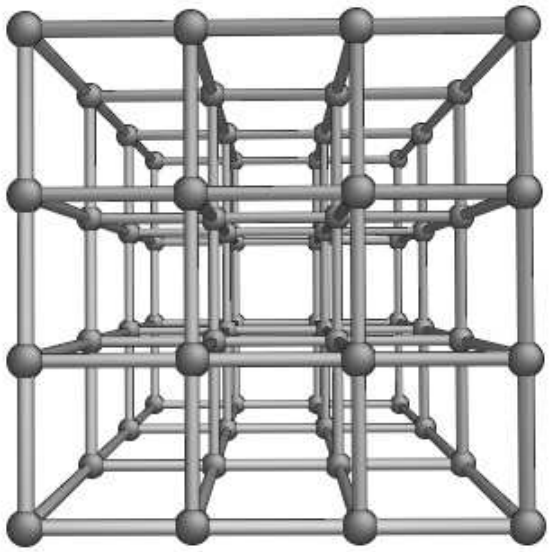

Figure 2.3: Reconstruction of a cubic grid with side equal to 4 and N=64 using Liga.

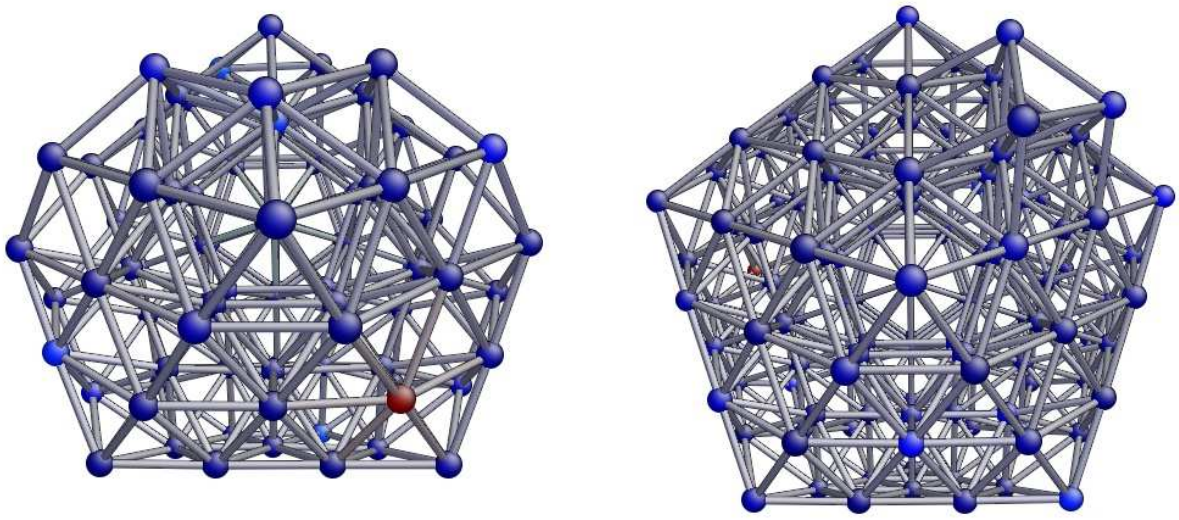

Figure 2.4: Reconstruction of Lennard Jones clusters using Liga. On the left is solution for N=88 and on the right is N=150. The solutions have a low error and are topologically identical to the LJ-88 and LJ-150 clusters. The atoms in blue have a low error, while those in red have high error.

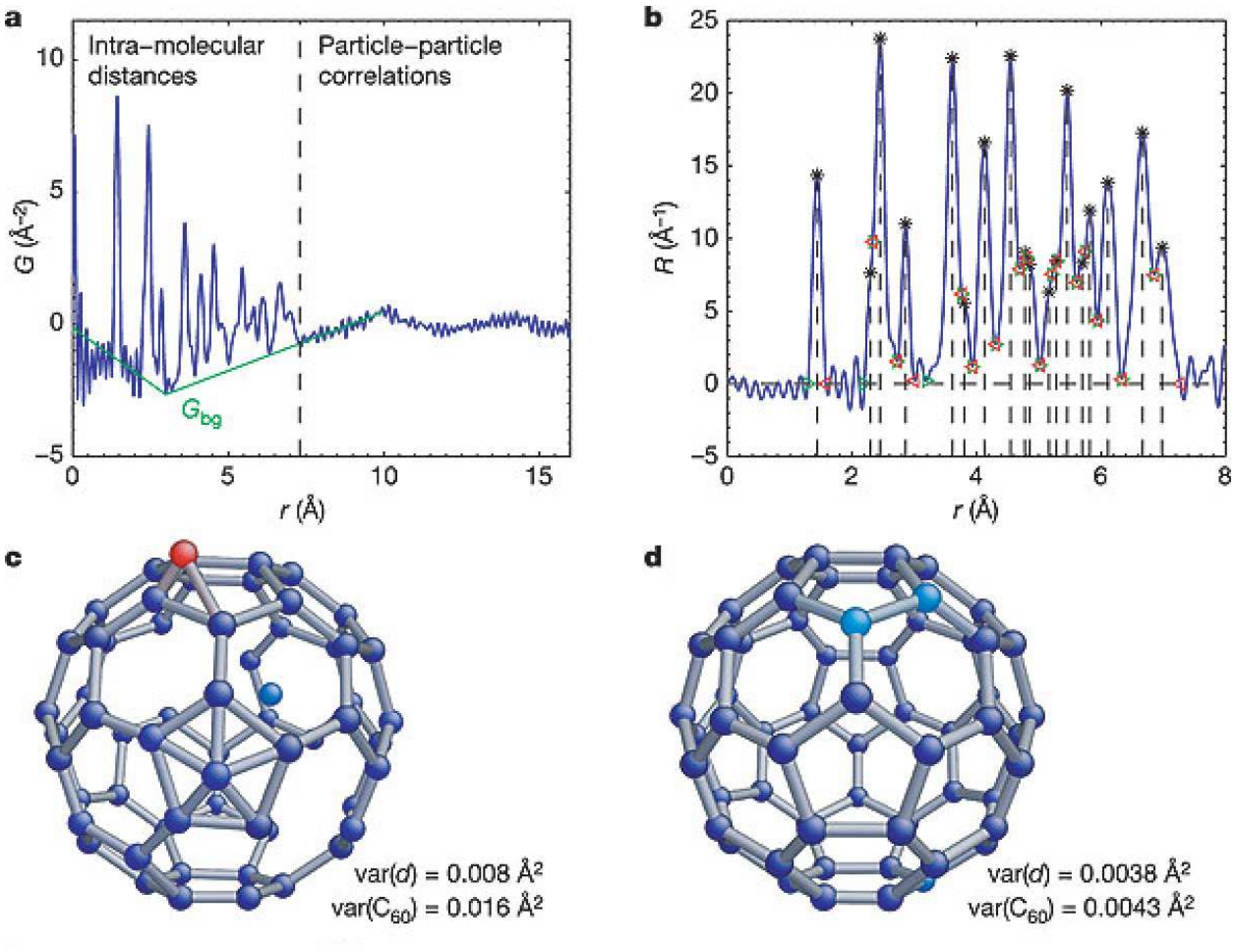

Figure 2.5: Reconstruction of C_{60} from experimental PDF data. a) Experimental pair distribution function (G) as a function of distance (r). The background (G_{bg}) arising from interparticle correlations is shown in green. b) The radial distribution function (R) as a function of the distance (r). It is obtained after subtracting the background from the PDF data. The interatomic distances are obtained by using the peak maxima and the multiplicities are set equal to the peak areas. c) The solution structure obtained using the exact distance list obtained in the previous step. d) The solution obtained after the multiplicities in the distances are relaxed by 10%. The atoms in blue have a low error, while those in red have high error.

2.2 Limitations

Liga fails with structures with a low symmetry and a large number of unique distances (Fig. 2.6). It is unable to reconstruct random point sets of size ten. Random point sets have

$N(N-1)/2$ interpoint distances, which with high probability are all unique. The reason for the difficulty in reconstruction arises from a large number of low cost candidate points and substructures that are not part of the target structure. It is not able to make progress while searching the phase space as it gets stuck in these rather large number of local minima.

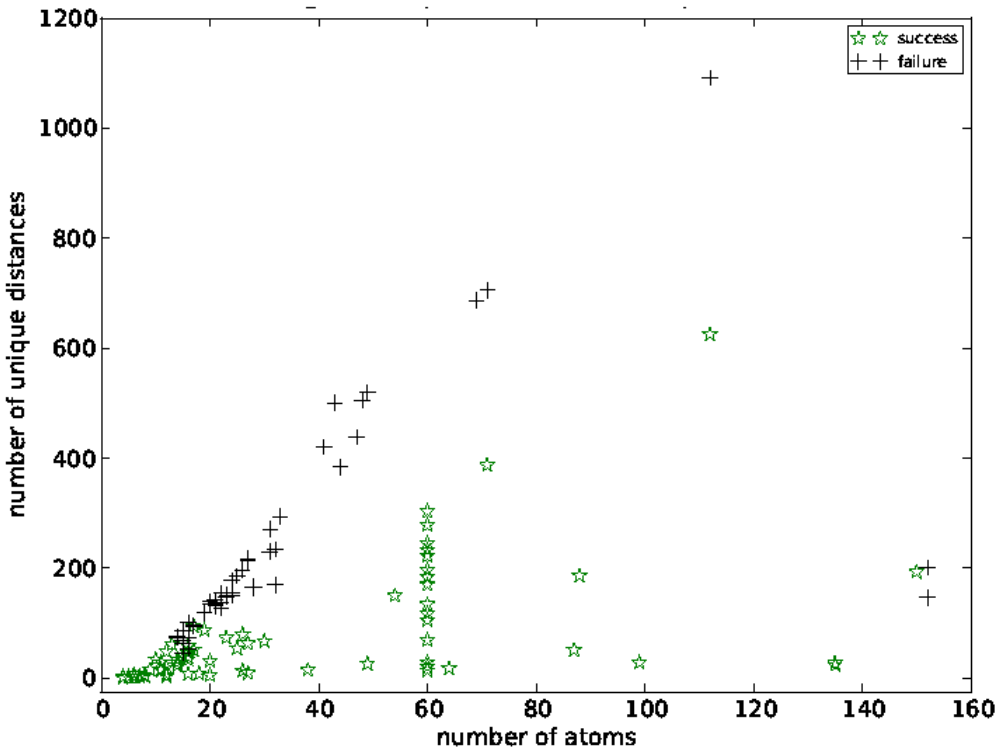

Figure 2.6: Liga's success and failures in reconstructing structures with different amounts of symmetry. Low symmetry structures have a large number of unique interpoint distances, while those with high symmetry have a small number of unique interpoint distances. Failure is represented by the plus symbol while success is denoted by the star symbol. Success mostly occurs in the region closer to the X axis, which is representative of the structures having high symmetry while failure mostly occurs mostly for structures having a large number of unique distances.

2.3 Extension of the Liga algorithm

The algorithm was extended to solve for the structure of periodic crystal structures from experimental PDF data (Fig. 2.7). It is a three step process, where the first step is the extraction of interatomic distances from experimental PDF data. In the second step, Liga is used to find the coordinates of the atoms in the unit cell. In the third step, the assignment of atom type to each site is carried out.

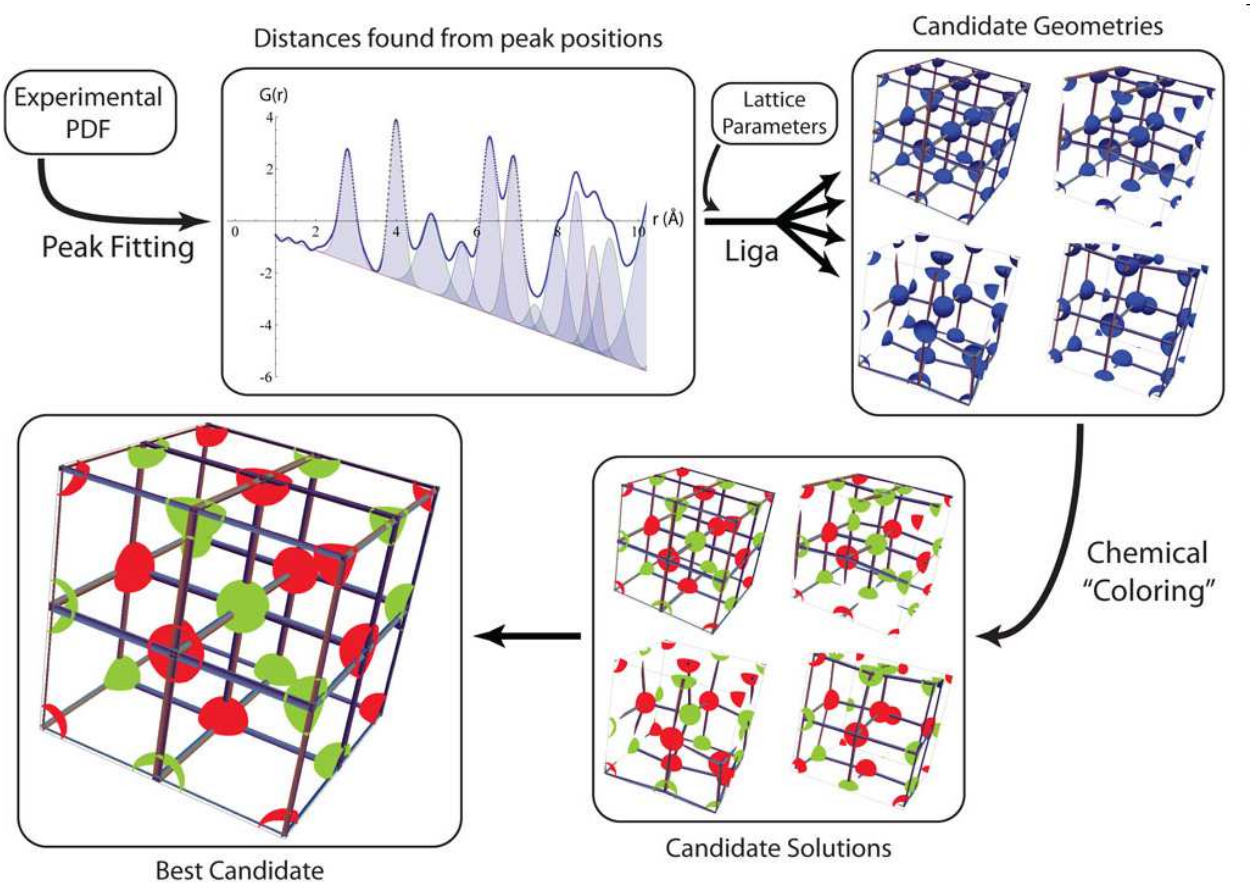

Figure 2.7: Steps involved in the crystal structure determination using experimental PDF. An automated peak extraction routine is used to obtain the distances and their multiplicities. This information along with the lattice parameters for the crystal structure is given as input to Liga. It gives as output a number of candidate solutions that are consistent with the input data. The next step is coloring, which assigns the atom species to each site by minimizing the atom radii overlap and the structure with the lowest cost is declared as the solution.

The process of extracting interatomic distances from experimental PDF was done using

an automated peak extraction algorithm developed by my colleague, Luke Granlund. The cost function is such that it guides the algorithm to generate peaks that fit well to the given data and also makes sure that it uses as few peaks as possible in order to avoid over-fitting. In the next step, Liga is used for reconstructing the position of atoms in the unit cell so that they are in good agreement with the distance list obtained in the previous step. This step is repeated with multiple starting random seeds to confirm the correctness and uniqueness of the solution. In most cases there was a unique solution but for some, there were multiple geometries that were a good match to the input distances. In such cases the correct structure was identified after the coloring step.

The next step is the assignment of atom type to the sites in the unit cell and this step is also known as "coloring". For a structure with size N, which is made up of k different atom species, then the number of ways of assigning the atom types to sites is given by $N!/(n_1!n_2!n_3!...n_k!)$. For a binary system with equal number of atoms of each type, having a chemical formula A_kB_k, the number of possible assignments grows exponentially as 2^N for large N. This growth makes exhaustive search impossible and a bad approach. The coloring cost can potentially be measured in two different ways. The first one is based on how well the model PDF matches with the experimental PDF. The second one is the average error in all the differences in the interatomic distances and the corresponding sum of radii of every atom pair in the structure. The cost defined in the second way is computationally less expensive and was chosen for this step. A simple downhill search was employed, where it starts with a random assignment. The coloring cost of the initial random assignment is calculated along with those from all possible 2 atom swaps. The swap that leads to largest decrease in the cost is accepted and the process is repeated until a minimum is found. This simple downhill search was able to find the correct assignment in all the samples that were tested.

The solutions obtained were compared with the structure known from literature to check if they had the same nearest neighbor coordination and also by calculating an overlay error. The extended algorithm successfully able to reconstruct 14 out of 16 crystal structures that were attempted. Ag, $BaTiO_3$, C graphite, CdSe, NaCl, Ni, PbS, PbTe, Si, $SrTiO_3$, Zn, ZnS sphalerite and ZnS wurtzite were solved successfully, while it failed for $CaTiO_3$ and TiO_2 rutile.

2.4 Summary

In this chapter, the concepts and steps behind the Liga algorithm are presented. Liga is successful in reconstructing structures which have a high symmetry like the C_{60} buckyball using only their unassigned interpoint distances. The Liga algorithm was extended in order to solve periodic crystal structures from experimental PDF data. It is successful in reconstructing structures of 14 well known inorganic crystalline materials that were studied.

Chapter 3

The Tribond 2D algorithm

In this chapter the problem of complex structure reconstruction in two dimensions is studied. The chapter is organized as follows. Section 3.1 summarizes the theoretical concepts upon which the UPD reconstruction algorithm is based. The key concepts are based on constraint counting and generic graph rigidity that have a long history in the physics and mathematics literature. Section 3.2 discusses implementation of the procedure, which broadly consists of two phases: identification of a rigid core and buildup from a rigid core. A naive implementation is quite inefficient, however a simple optimization where cores are found using a selected subset of the distance list provides a much more efficient implementation. A loose polynomial upper bound on the computational efficiency of the algorithm is also developed and compared with the actual data. Large random point sets may yield distance lists that are close to degenerate, leading to problems with reconstruction. Section 3.3 discusses the algorithm in the context of the broader problem of reconstructing non-crystalline and heterogeneous materials. Finally in Section 3.4 is the summary.

3.1 Rigidity theory of unassigned PD-IP

Graph rigidity theory addresses the issue of how many independent constraints are required to ensure that a graph is rigid [70, 71, 72, 73]. This subject was initiated by James Clerk Maxwell leading to the development of mathematical theories of graph rigidity and physical

approximations to the rigidity of glasses [74]. In D dimensions a point has D translational degrees of freedom, so a structure with N nodes has DN degrees of freedom. The number of internal degrees of freedom is $DN - D(D+1)/2$ as there are $D(D+1)/2 = D + D(D-1)/2$ degrees of freedom due to global translations (D) and rotations ($D(D-1)/2$). An object is rigid when its internal degrees of freedom are constrained leaving only its global rotations and translations.

A constraint such as an interpoint distance contributes to the rigidity of a structure only if it is linearly independent with respect to the other constraints in the structure, so that identification of degenerate constraints is key to accurate constraint counting. Several mechanisms for the degeneracy or linear dependence of constraints in small structures are illustrated in [20]. In a classic paper, Laman [75] presented a combinatorial characterization of the rigidity of graphs in the plane and Hendrickson [20] provided the basis for efficient algorithms that have been widely applied in physics, applied mathematics and in biology. Note that if a graph is rigid it can support an applied stress. Addition of further bonds or edges to a rigid graph does not increase its rigidity, though of course the elastic moduli continue to increase as further bonds are added. These additional bonds are called redundant bonds as they do not contribute to the graph rigidity. An important feature of redundant bonds in graphs is that, except in special cases, each redundant bond leads to overconstraint that in most physical situations leads to an internal stress.

Now the question as to whether there is enough information or constraint to solve the UPD problem is addressed. If there are B independent distances between the nodes of a Euclidean network, the critical number of independent constraints, B_c, required to make the network rigid is,

$$B_c = DN - D(D+1)/2. \tag{3.1}$$

In an ideal NMR or PDF experiment all interparticle distances would be extracted so that the number of interparticle contraints would be $N(N-1)/2$ which appears to be more than enough to constrain the structure. However, it is not clear that this is the case as is evident by considering the C_{60} molecule as illustrated in Fig. 1.1, where there are only 21 different interatomic distances. Since for a buckyball, $B_c = 3 \times 60 - 6 = 174 >> 21$, it appears that there are far fewer distance constraints than is required to find the correct structure using the distance list alone. However, the distances with the same length are not necessarily degenerate as they may have different directions in the structure. Mathematical analysis of this issue is currently absent and is an important challenge. In contrast for generic random point sets that are of interest here, all of the distances are unique so that for a random Euclidean network with $N = 60$ nodes, there are 1770 different distance constraints, which is far more than is required to specify the Euclidean network in three dimensions.

The above discussion indicates that there are more than enough pair constraints in complex Euclidean networks to specify the network structure. As described in the next section, these rigidity concepts may be used to develop an efficient reconstruction algorithm. However it is important to keep in mind the limitations of this approach, including the issues of degeneracy and the fact that Laman's theorem [75] only strictly applies to planar graphs.

The theoretical foundation of efficient algorithms for the UPD problem rests on rigidity theory discussed above that states that an isostatic structure in two dimensions (from Eq. 3.1) has $B_c = 2N - 3$ independent distance constraints. However, the key test of whether the assignment of distances to natural lengths is correct is to place at least one additional, overconstrained Euclidean distance into the structure. A distance incompatible with the isostatic structure leads to a finite strain energy cost in Eq. 1.1, due to stretched or compressed springs, while a distance compatible with the isostatic structure has zero energy cost. Note

that many isostatic structures that are *inconsistent* with the final structure can be made, but with high probability, no overconstrained zero cost structures can be made that are inconsistent with the final reconstruction.

3.2 Tribond 2D algorithm

In two dimensions the smallest structure with at least one overconstrained bond is $N = 4$ where the total number of bonds is $\binom{4}{2} = 4 \times 3/2 = 6$, while the number required for isostaticity is (from Eq. 3.1) $2N - 3 = 5$. The key observation is that if six Euclidean distances are found that form a point set structure, and the cost function for this structure and these distances is zero, then a unique substructure has been found. A zero cost, correct, substructure with six distances and four sites is called a *core*. Once a core is found, and if there is no degeneracy, then this core is a correct substructure of the complete reconstruction. One may then build up from the core iteratively to find the complete structure. At each step there is an existing, correct substructure. Then add one site and search for three edges that are compatible with the new node and with three nodes that are in the existing structure. The addition of one site and two edges is an isostatic addition, while the addition of one site and three edges is overconstrained. If three edges that are compatible with one additional site and three sites in the existing structure are found, with high probability, this site is part of the correct reconstruction.

In practice, to construct a core we choose the smallest bond as the base for all our triangles and loop over all triangle pairs which are feasible according to the triangle inequality. For every triangle pair we calculate the length of the bond that connects the two apex points which we call the bridge bond. The length of the bridge in the candidate core is tested

against the lengths in the distance list. If the candidate bridge length is equal to an unused distance in the distance list, we have found a core (Fig. 3.1 and Fig. 3.2).

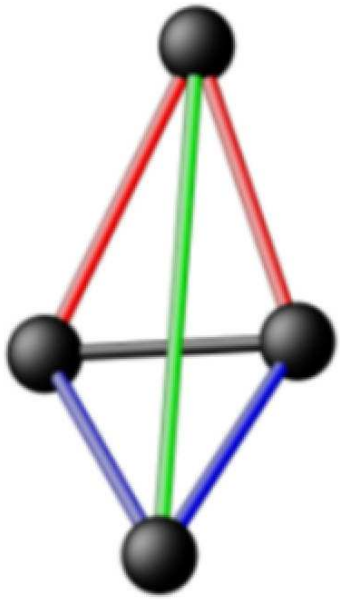

Figure 3.1: (color online) An example of a core. In 2D, it consists of 4 points. The horizontal bond is the base (in black), the bonds below it (in blue) make up the base triangle while those above it (in red) make up the top triangle. The vertical bond is the bridge (in green).

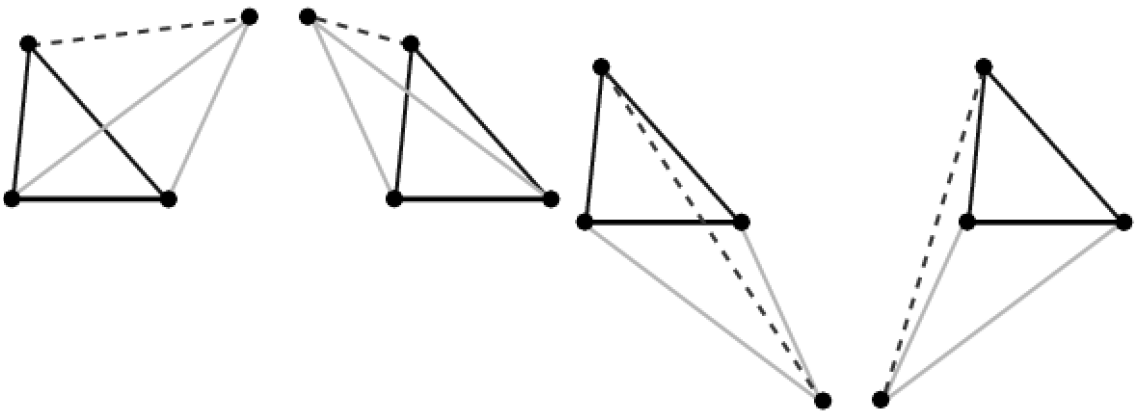

Figure 3.2: Four possible positions for the top triangle are shown. The corresponding bridge bonds are shown using a dashed line.

The build up procedure consists of choosing a triangle as the base triangle in the existing structure, followed by an attempt to add a site to it. The addition of a site consists of

choosing an edge in the base triangle as the base bond and generating test triangles with that base bond and using two distances from the distance list. After we place this site, we carry out bridge testing to determine whether the structure has zero strain energy.

Our Tribond implementation of the above procedure for the unassigned PD-IP algorithm may be summarized as follows:

We are given the sorted distance list $\{d_l\}$ with the number of nodes in the network N. We start with an empty set, then

A. Core finding procedure

1. Choose the shortest bond as the base bond and a window (subset) of $W = 6$ entries in the distance list for the core finding search.

2. Iterate over all triangles constructed with the triangle inequality that have the same base bond using distances in the window W.

3. Recursively search over all the pairs of the feasible triangles generated above and over all lengths in the Euclidean distance list to find a bridge. If a compatible core is found, remove the edges used from the distance list and exit.

4. Increment W and return to (1), making sure not to retest bond combinations.

B. Buildup procedure.

1. Choose a base triangle to be the reference for our buildup.

2. Search over all sets of two edges from the distance list to find a set compatible with the base triangle in the existing structure. Search over the distance list to find a bridge bond.

3. If successful, remove from the distance list the edges that are used in connecting the newly added node. If size of reconstructed structure is $< N$ return to step 1 of the Buildup procedure.

A coarse upper bound on the computational time for this procedure consists of two parts: (i) the time to find the core; (ii) the time to carry out the buildup procedure. The number of unique cores in the point set is $\binom{N}{4}$, the number of ways of choosing 4 sites from N total sites. The number of ways of choosing six distances from the set of $M = N(N-1)/2$ distances is $\binom{M}{6}$. A brute force search then finds a core in computational time $\tau_{core} \sim \binom{M}{6}/\binom{N}{4} \sim N^8/1920$. Using similar reasoning, a brute force buildup algorithm takes a computational time that scales as $\tau_{buildup} \sim \binom{M}{3} \sim N^6/48$. This clearly shows that the method is polynomial though the power of the polynomial is too high for this to be practical.

The simple methods we have developed reduce the computational time very significantly from the coarse upper bounds of the last paragraph. The key observation is that many of the distances in the distance list violate the triangle inequality $d_1 + d_2 \geq d_3$, so they clearly cannot form a triangle together. A large fraction of the computational time in a brute force search is spent exploring these trivially inconsistent distance combinations. If we fix the base bond and the bridge bond is found using binary search, using simple combinatorial arguments we get $\tau_{core} \sim \binom{M}{4} ln(N)/\binom{N}{2} \sim N^6 ln(N)$. For a triangle with base bond a and second side b, the range of values for third side c is $(b-a, b+a)$. So for a larger base bond a, there is a much bigger range of feasible values for the the third side and hence the number of feasible

triangles goes up. But the actual number of triangles in the target structure is the same for any choice of base bond. This is seen in Fig. 3.3, where the number of feasible triangles goes up with the fractional position of the base bond in the distance list. Hence statistically, we can find a core in the least time if we choose the shortest bond in the distance list as our base.

Distances are also more likely to satisfy the triangle inequality if they are drawn from a list of comparable, rather than disparate, lengths. Since the base bond is short, a core is more likely to be found quickly by searching over other short distances first (the small-core hypothesis, Fig. 3.4), and including longer distances only as necessary. This is implemented as a window of the W shortest distances in the distance list, which increases periodically as core finding proceeds. Of the six bonds in the core, the base is fixed, four are drawn from the window, and the bridge bond may appear anywhere in the distance list. We observe that a window of size $W \sim N$ is usually sufficient to find a core. Therefore, typical computation time is $\tau_{core} \sim \binom{N}{4} ln(N) \sim N^4 ln(N)$.

From Fig. 3.3 and Fig. 3.4, we can guess that when we use the smallest bond as the base it will lead to the core finding and reconstruction in the shortest possible time. This is confirmed from our runs and can be seen in Fig. 3.5 where we used base bonds that were picked from 10 different places uniformly spread along the sorted distance list. If the smallest bond is chosen as the base, it took significantly less time for the entire construction.

Attempting to find the core for large point sets ($N > 200$) frequently leads to bad cores. Bad cores are over constrained clusters whose distances are part of the given distance list, within our tolerance, but the substructure is not present in the target structure. This occurs due to the fact that we are using finite tolerance when checking for the bridge bond and we also have finite precision when we are doing the triangulation while placing the points.

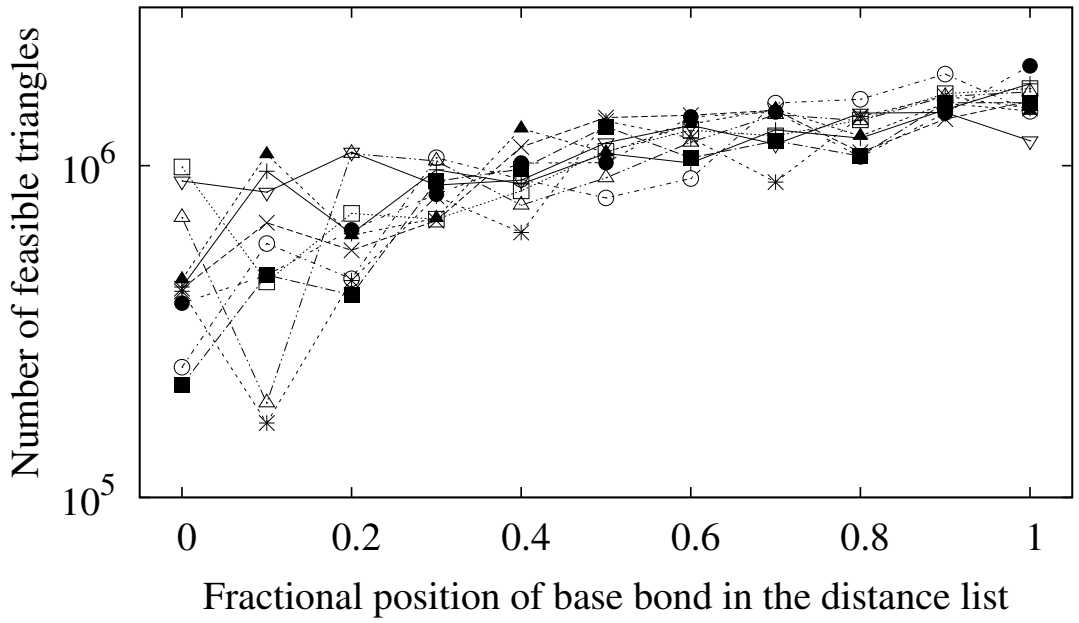

Figure 3.3: Number of feasible triangles using the bonds from a given distance list go up when we choose a larger bond as base for the triangle. Statistically, using the shortest bond in the distance list as the base leads us to the core in the shortest time. This plot shows data from runs using 10 different structures with $N = 128$.

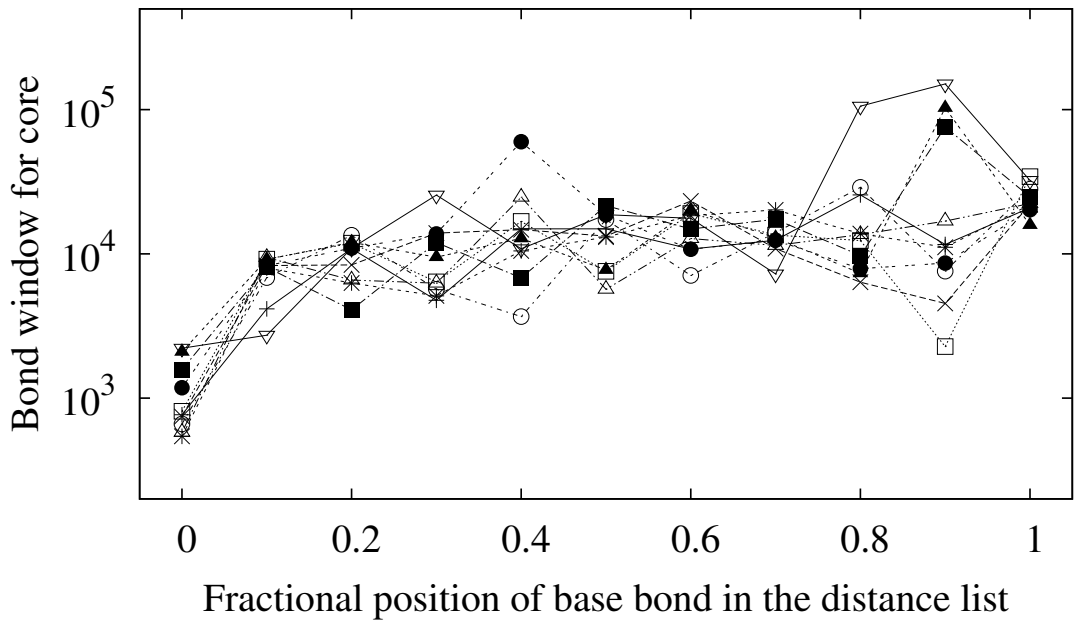

Figure 3.4: Small core hypothesis: ($N = 1024$) We see that when we have the smallest bond in the distance list as the base, the first core is in a distance window an order of magnitude smaller than other choices for the base bond. Hence, statistically, using the first bond as base is our best bet when searching for the core.

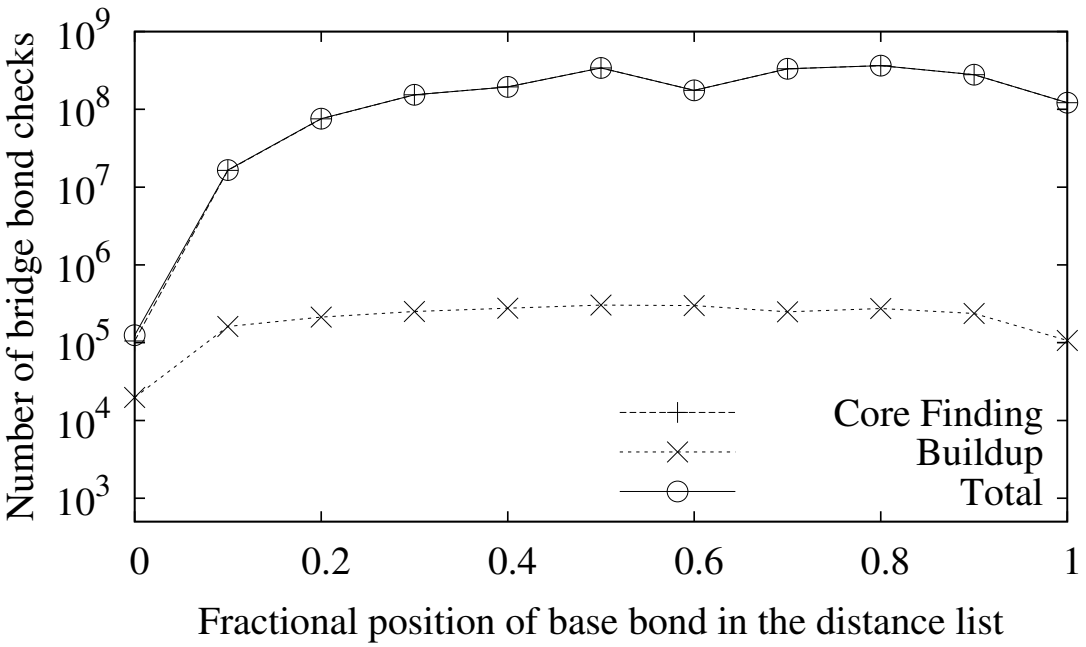

Figure 3.5: Plot illustrating the role of the base bond. For $N = 32$, the Tribond algorithm ran using base bonds that were picked from 10 different places spread along the sorted distance list. If the smallest bond is chosen as the base, we see that it takes 3 orders of magnitude less time for the core finding stage and an order of magnitude less time for the buildup stage.

We also see a loss in precision when placing points by doing triangulation while using the smallest length in the distance list as the base bond. So, we try to use all 6 bonds (in the core) as the base bond and check if the corresponding bridge bond is valid or not. We only take cores for which the bridge bond is valid in all of the 6 cases. This stringent check is very good at identifying bad cores. A confirmatory test is to use a structure comparison routine that flags the core as bad.

We have developed a structure comparison routine that overlays the points in the reconstructed structure ($\vec{r}$) with the points in the target structure ($\vec{R}$) and calculate an overlay error (Eq. 3.2) which can tell us how good the fit is. This routine can also tell us if a given substructure is part of the target structure or not. This is useful for testing purposes and not for the practical application, where we do not know the answer (target structure) ahead

of time.

$$\epsilon_{overlay} = \sum_i |\vec{r}_i - \vec{R}_i|^2. \tag{3.2}$$

When we find a core, we attempt buildup and add more points to the substructure. If after looping over a certain number of bonds from the distance list, it is unable to add any points, then we call it a bad core and get back to the core finding stage and find the next one. This is a heuristic method that helps us identify bad cores in a short amount of time. We have observed that the core is bad because even when given a large amount of time, it fails to add a significant number of points while doing the buildup.

It is important to choose the appropriate amount of tolerance when checking if the bridge bond is part of the given distance list. Using a very loose tolerance leads to a large number of bad cores. On the other hand, if we use a very tight tolerance we miss out on good cores, because we have finite precision when carrying out the triangulation to place the points in our substructure. We use floating point numbers for the input distances which has an accuracy of 18 digits. We found that using a relative tolerance of 10^{-12} is optimal to make sure that we get the good cores and filter out the bad ones. We observe a loss of precision when trying to place points that are collinear to the points that form the base bond. In such situations we relax the tolerance when checking for the correctness of the bridge bond.

To check the validity of a new point while doing buildup, in addition to the bridge bond check, we check 10 additional distances that it creates with the points already in the substructure. Only if these are part of the distance list do we add this new point to the structure. Whenever a new point is added to the substructure, we note the 3 bond lengths (two from the new triangle created and the third is the bridge) that were used and make

them unavailable during further reconstruction. This reduces the list of available distances by 3. When placing the n^{th} point if we update all $n-1$ distances created between the new point and the points already in the substructure this reduces the number of available distances substantially but we found that it does not lead to any significant speedup in the buildup routine.

If after doing the buildup we still don't have the desired number of points (N), we relax the tolerance for the bridge bond checks and rerun the buildup procedure. If we are still short, we choose a different bond as the base and attempt to do the buildup using that bond. Once we have a full reconstruction, we calculate the distance error, which is based on the agreement between the given distance list and the distances derived from the reconstructed structure. We also calculate the overlay error (Eq. 4.1) using the structure comparison routine which is useful for testing purposes.

The Tribond algorithm was run for $N = 8, 16, ..., 512$ and the computational cost for the core finding and buildup stages was recorded. The cost is the number of bridge bonds that were checked when placing a point in the structure. It is useful as it is a system independent measure of the cost. The timing runs are given in Fig. 3.6. We can see that the time required for doing the buildup is about an order of magnitude less than that for the core finding stage. The scaling is $\tau_{total} \sim N^{3.32}$, which shows that our algorithm has a polynomial run time albeit a higher order one. This scaling proves to be better than our estimate obtained earlier using simple combinatorial arguments as we had not accounted for the speedup obtained by using the triangle inequality.

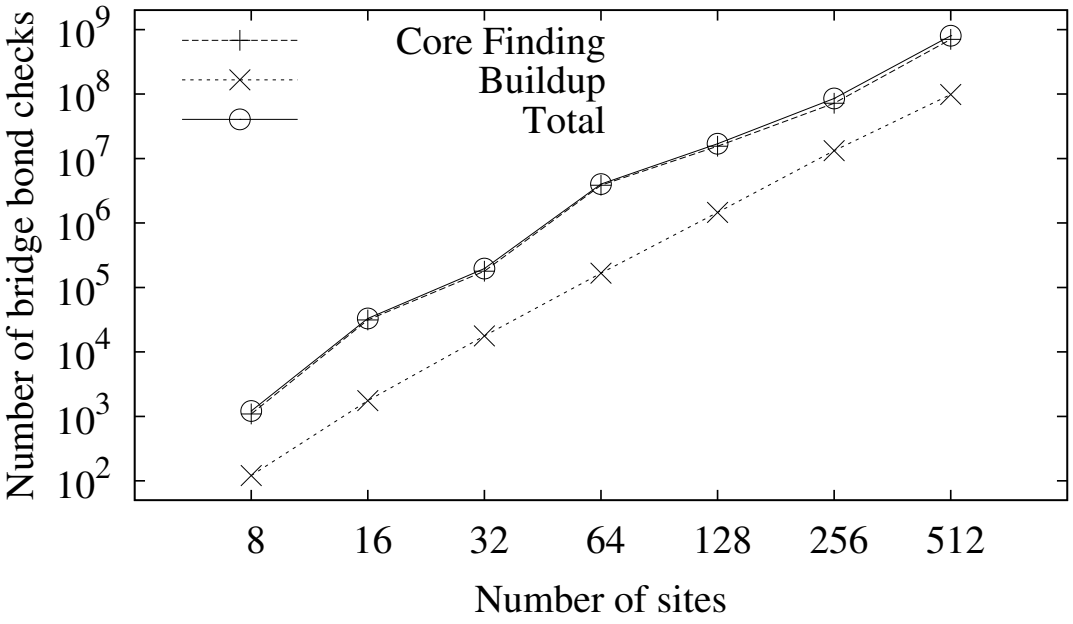

Figure 3.6: Experimental results for a series of reconstructions from distances lists generated from random point sets in two dimensions. The time for finding the core, the time for doing the buildup starting with the core and the total time are presented as a function of the number of bridge bond checks that were performed. Bridge bond checking is a fundamental process in Tribond and provides a system-independent measure of computational time. Each point on the plots is an average over 25 different instances of random point sets. We find that the total time scales as $\tau_{total} \sim N^{3.32}$.

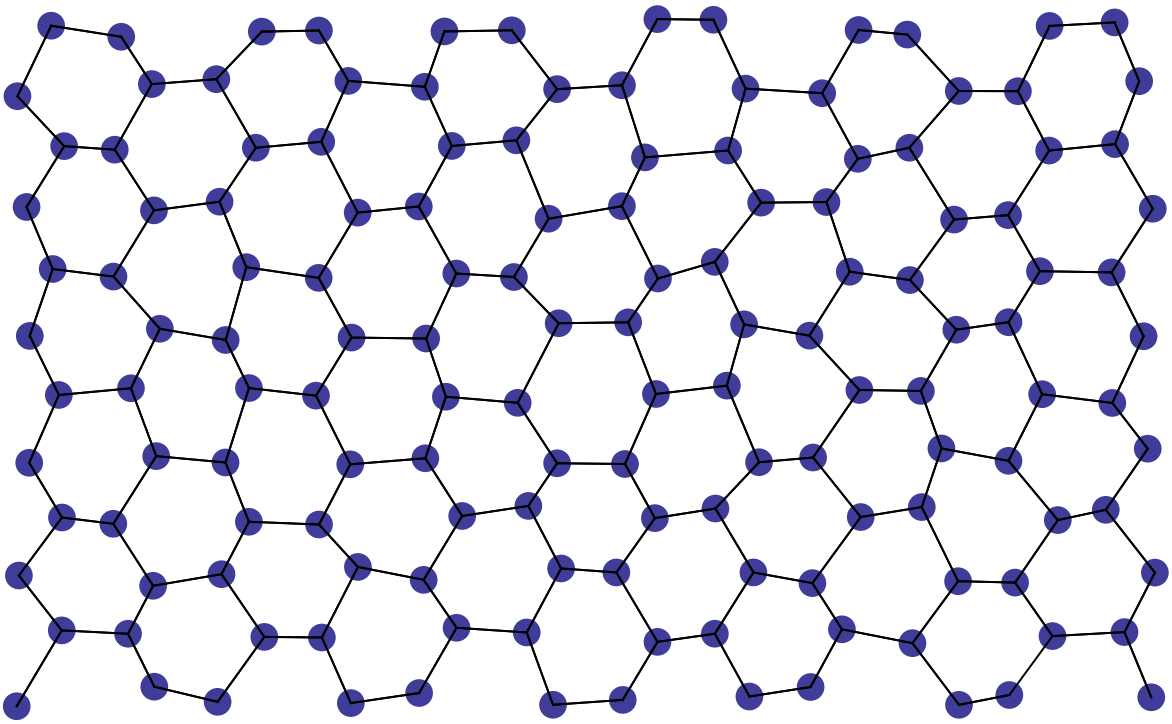

Figure 3.7: A perturbed graphene cut out made from 144 atoms. The Tribond algorithm successfully reconstructed a similar structure in a few minutes.

3.3 Applications

In the previous section we showed that Tribond is successfully able to reconstruct random point sets. We now try to solve some structures which occur in the real world. Structures occuring in nature are usually symmetric but because of finite size effects they have defects which cause small deviations in their "ideal" locations. Fig 3.7 shows a graphene nanoparticle with 144 atoms. The location of each point differs from their "ideal" ones via a small noise added to simulate natural imperfections. Tribond reconstructed this graphene sheet in a few minutes.

Tribond can also solve 2D polymers modeled by self avoiding random walk (Fig. 3.8). If the polymer is modeled as a random walk in the continuum, then it is just like a random point set. Since we have all the pairwise distances for the structure, it forms a complete graph.

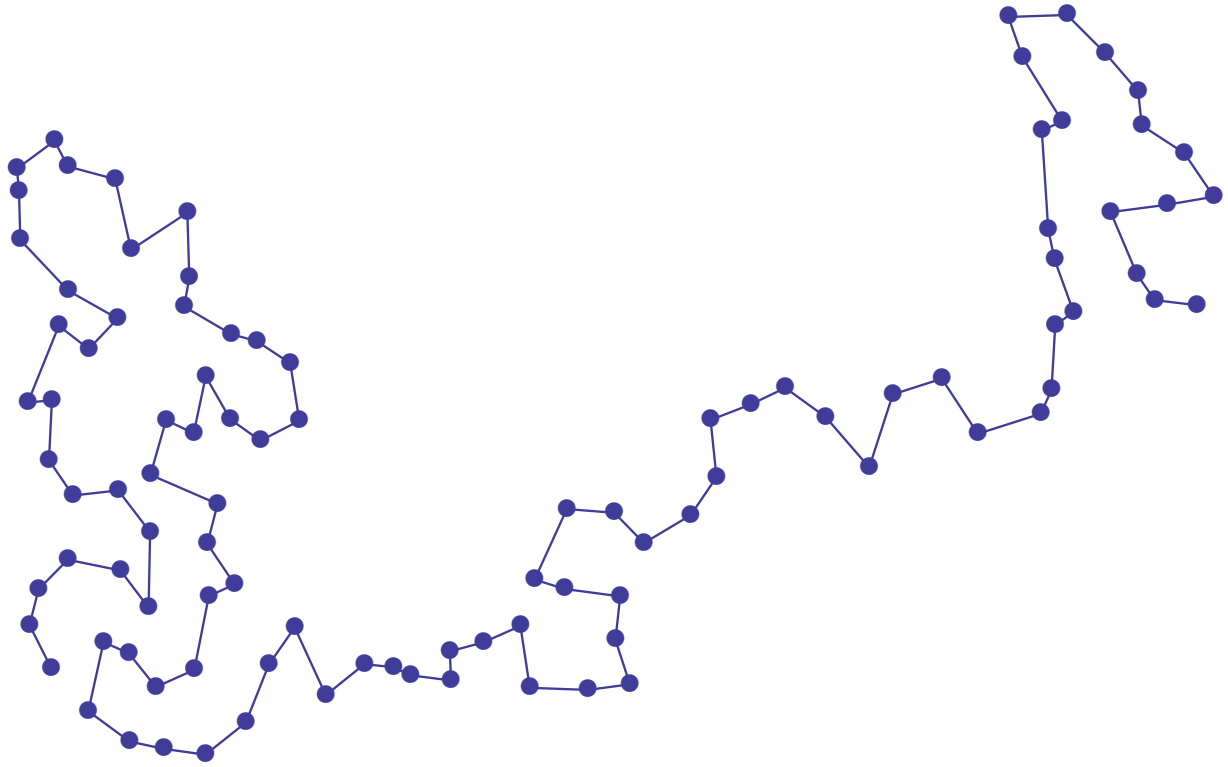

Figure 3.8: Self-avoiding walk is a sequence of moves that does not visit the same point more than once and is used to model polymers. Tribond was able to successfully reconstruct the above structure ($N = 100$) in a few minutes.

From graph theory we know that every complete graph is Hamiltonian, i.e. there is a path that visits every point exactly once. Hence polymers modeled as a self avoiding walk equate to a random point set for reconstruction purposes. We are able to reconstruct polymers (like in Fig. 3.8) knowing only their unassigned distances using the Tribond algorithm.

We also reconstructed a 100 site point set on a square grid as shown in Fig. 3.9, that was gently perturbed, in a few minutes by our algorithm.

3.3.1 Tribond for structures with high symmetry

We refined the idea behind Tribond to make a modified algorithm that can deal with structures having symmetry which have a highly degenerate distance list. We use only one instance

of each distance from the given distance list and find a core. During the buildup, at each step of the reconstruction we use only one instance of each distance and keep track of its multiplicities. So, at any given time, the distances that are available to the algorithm are all unique. This cuts down on the number of bad cores and points and helps guide our search. Using this modified approach we have been able to solve square grids with up to N=1024 (32×32) points in under 10 minutes on a desktop computer (but there is some trouble for N=400, 676, 900 as the algorithm tries to grow into a bigger lattice instead of completing the grid).

3.3.2 Reconstruction from an imprecise distance list

So far, we always started with a distance list which had entries that were known to a very high precision of 18 digits. Imprecise distance lists have less information in them and that makes it more difficult to solve our inverse problem. So we modified our original algorithm to deal with this situation as follows.

1. Start with a core (or substructure) and an empty pool which can save the coordinates and their associated cost for up to a maximum of 20 candidate points.

2. We randomly choose a bond in the substructure as the base for our buildup. Then we search over all sets of two edges from the distance list to make a test triangle and the new vertex is our test point. Evaluate the complete cost of the new substructure if this test point were to be added. If this cost is low and is less than that of the worst point in the pool, then add this point to the pool in the correct place based on its cost. Remove the worst point from the pool so that its size never exceeds the maximum.

3. Now choose another base bond randomly in the current substructure and repeat the

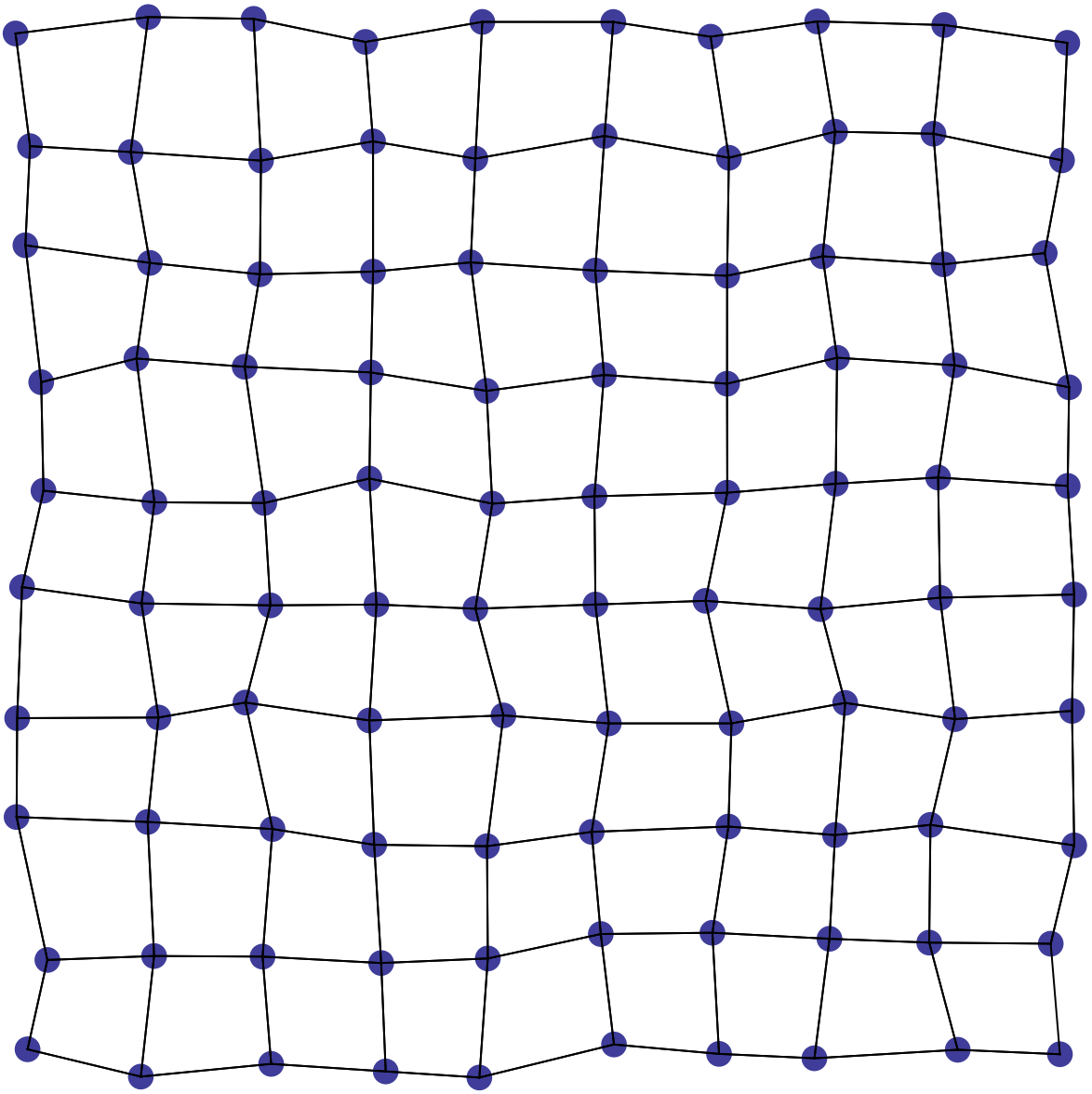

Figure 3.9: Gently perturbed square grid made of 100 sites. Our algorithm was successfully able to solve such a structure in a few minutes.

previous step. Combine the two pools obtained so far based on the cost such that we have up to 20 candidate points.

4. Iterate over all possible 2 point combinations (in 20 choose 2 ways) from the pool and find the pair which will have the minimum cost if added to the substructure.

5. Add the 2 points found in the above step to get a bigger substructure. If its size is $< N$, then go to step 1.

As compared to our buildup procedure (when we have precise data), we now do the buildup in multiple stages, first generating a pool of candidate points which have a low error with respect to the current substructure (based on single point addition). Then 2 points are added to the substructure from the pool which have the lowest error. We do this iteratively until we have the complete structure. As we gradually grow the structure and generate the pool of candidate points multiple times, we avoid all the bad points (low cost but wrong) and correctly guide the search.

Our results can be seen in Fig. 3.10, which shows the minimum core size needed to reconstruct a structure of size $N = 26, 50, 76, 100$ for different values of the precision (P) of the input distance list. The units for the precision of the distances is the number of digits. Our criteria for success was that the algorithm should be able to successfully reconstruct at least 5 out 10 different random point set structures. We can see that as the distances become less precise, a core of a larger size is needed for successful reconstruction. The typical run time for $N = 26, 50, 76, 100$ was about 1 minute, 10 minutes, 4 hours and 20 hours respectively, on a node in our high performance computing center. When the input data has a higher precision ($P > 8$) than what is shown in the plot, we found that a core size of 4 was sufficient to reconstruct the structure.

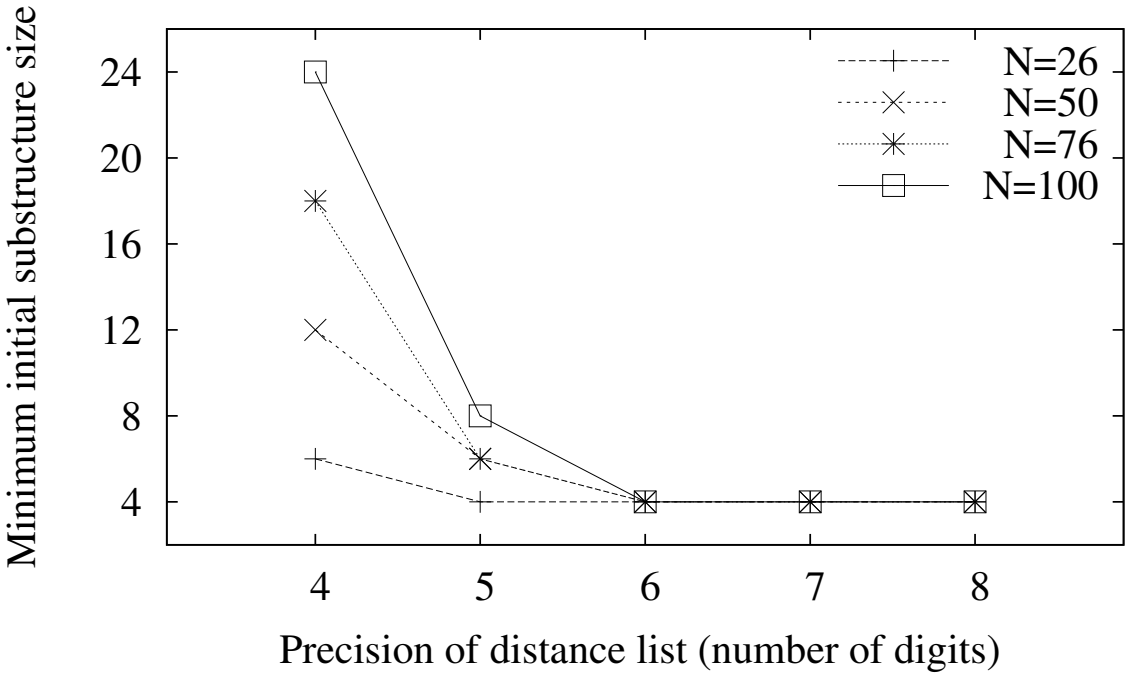

Figure 3.10: Plot of minimum core size vs precision of the input distance list for $N = 26, 50, 76$ and 100. We can see that a bigger core is needed for a less precise distance list.

3.4 Summary

In this chapter, the details of the Tribond 2D algorithm for the reconstruction of low symmetry structures represented by random point sets were presented. The Tribond 2D algorithm consists of two steps: core finding and buildup. The core is the smallest substructure with at least one over-constrained bond and is of size 4 in two dimensions. Choosing the smallest bond as the base bond for reconstruction had a dramatic improvement in performance. Computational cost of core finding was orders of magnitude more than buildup. Tribond 2D was able to reconstruct random point sets in 2 dimensions of size ~ 1000 in about 24 hours on a desktop computer when given precise distances.

A modified approach was presented for the buildup step with less precise data and given a known substructure. As precision decreases, it is clear that we need a substructure of larger size, underscoring the importance of the core finding step. We successfully reconstructed

random point sets of size 100, with the distances having 4 digits of precision, given a known substructure of size 24.

Chapter 4

The Tribond 3D algorithm

The extension of Tribond algorithm to three dimensions is discussed in this chapter.

4.1 Tribond 3D algorithm

In three dimensions the smallest structure with at least one overconstrained bond is $N = 5$, where the total number of bonds is $\binom{5}{2} = 5 \times 4/2 = 10$, while the number required for isostaticity is (from Eq. 3.1) $3N - 6 = 9$. The key observation is that if we find ten Euclidean distances that form a point set structure, and the cost function for this structure and these distances is zero, then we have found a unique substructure. We call a zero cost correct substructure with ten distances and five sites a *core*. If the distance list was non-degenerate, then with high probability, this core is a correct substructure of the target structure. We may then build up from the core iteratively to find the complete structure. At each step we have an existing, correct substructure. We then add one site and search for four edges that are compatible with the new node and with four nodes that are in the existing structure. The addition of one site and three edges is an isostatic addition, while the addition of one site and four edges is overconstrained. If we find four edges compatible with one additional site then, with high probability, this site is part of the target structure.

In practice, to construct a core (Fig. 4.1) we choose the smallest bond as the "base bond". We then test all the bond combinations using the triangle inequality to generate

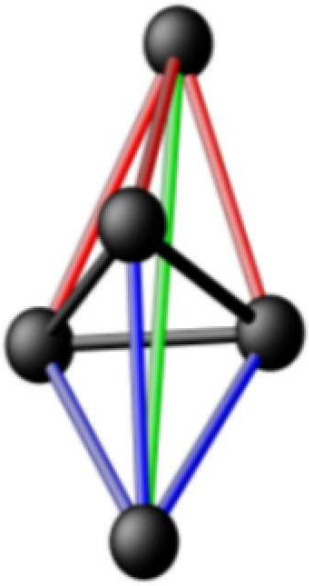

Figure 4.1: (color online) An example of a core. In 3D, it consists of 5 points. The points at the top and at the bottom are the apex points. The three points in the middle form the base triangle (in black). The base triangle along with the apex point at the bottom forms the base tetrahedron (in blue), while the base triangle along with the apex point at the top forms the top tetrahedron (in red). The vertical bond connecting the two apex points is the bridge (in green).

feasible tetrahedron pairs. This is performed in two steps, first we fix a tetrahedron as the "base tetrahedron" and then search through all other candidate ("top") tetrahedra that share the same base triangle. After all the top tetrahedra have been exhausted then a new base tetrahedron is selected and the process continues. For every tetrahedron pair we calculate the length of the bond that connects the two apex points, which we call the bridge bond. The length of the bridge in the candidate core is tested against the lengths in the distance list. If the candidate bridge length matches an unused distance in the distance list, we have found a core.

In the buildup procedure, we try to add more sites to the core. The addition of a site consists of generating candidate top tetrahedra using the base triangle and three distances from the distance list. After we place this site, we carry out bridge testing to determine

whether the structure has zero strain energy. While core finding requires a search over all possible base and top tetrahedra, buildup requires only a search through top tetrahedra as the base tetrahedron is a known part of the structure. Consequently, buildup requires significantly fewer computations than core finding.

Our Tribond implementation of the above procedure for the unassigned PD-IP algorithm may be summarized as follows:

We are given the sorted distance list $\{d_l\}$ with the number of nodes in the network N. (The target network is generated by randomly placing N points in a cubic box with side of length N.) We start with an empty set, then

A. Core finding procedure

1. Choose the shortest bond as the base bond and a window (subset) of $W = 10$ smallest entries in the distance list for the core finding search.

2. Iterate over all triangles constructed with the triangle inequality that have the same base bond using distances in the window W and generate tetrahedra.

3. Search over all pairs of the feasible tetrahedra generated above and calculate the bridge bond. Using a binary search, test if there is an unused distance that matches the bridge bond. If such a value is found, we have a core. Remove the edges used from the distance list and exit.

4. Increment W by 10 and return to (1), making sure not to retest bond combinations.

B. Buildup procedure

1. Search over all sets of three edges from the distance list to find a set compatible with the base tetrahedron in the existing structure. Search over the distance list to test the bridge bond.

2. If successful, remove from the distance list the edges that are used in connecting the newly added node. If size of reconstructed structure is $< N$, return to previous step and resume the search.

A coarse upper bound on the computational time for this procedure consists of two parts: (i) the time to find the core; (ii) the time to carry out the buildup procedure. The number of unique cores in the point set is $\binom{N}{5}$, the number of ways of choosing 5 sites from N total sites. The number of ways of choosing ten distances from the set of $M = N(N-1)/2$ distances is $\binom{M}{10}$. If we had done a brute force search then we would find a core in computational time $\tau_{core} \sim \binom{M}{10}/\binom{N}{5} \sim N^{15}$. Similarly, using brute force for the buildup would take a computational time that scales as $\tau_{buildup} \sim \binom{M}{4} \sim N^8$. This clearly shows that the brute force approach is polynomial, although a high order one.

The simple methods we have developed reduce the computational time significantly from the coarse upper bounds of the last paragraph. The key observation is that many of the distances in the distance list violate the triangle inequality $d_1 + d_2 \geq d_3$. A large fraction of the computational time in a brute force search is spent exploring these trivially inconsistent distance combinations. If we fix the base bond and the bridge bond is found using binary search, using simple combinatorial arguments $\tau_{core} \sim \binom{M}{8} ln(N)/\binom{N}{3} \sim N^{13} ln(N)$. For a triangle with base bond a and second side b, the range of values for third side c is $(b-a, b+a)$. So a larger base bond requires a much larger range of feasible values for the third side and,

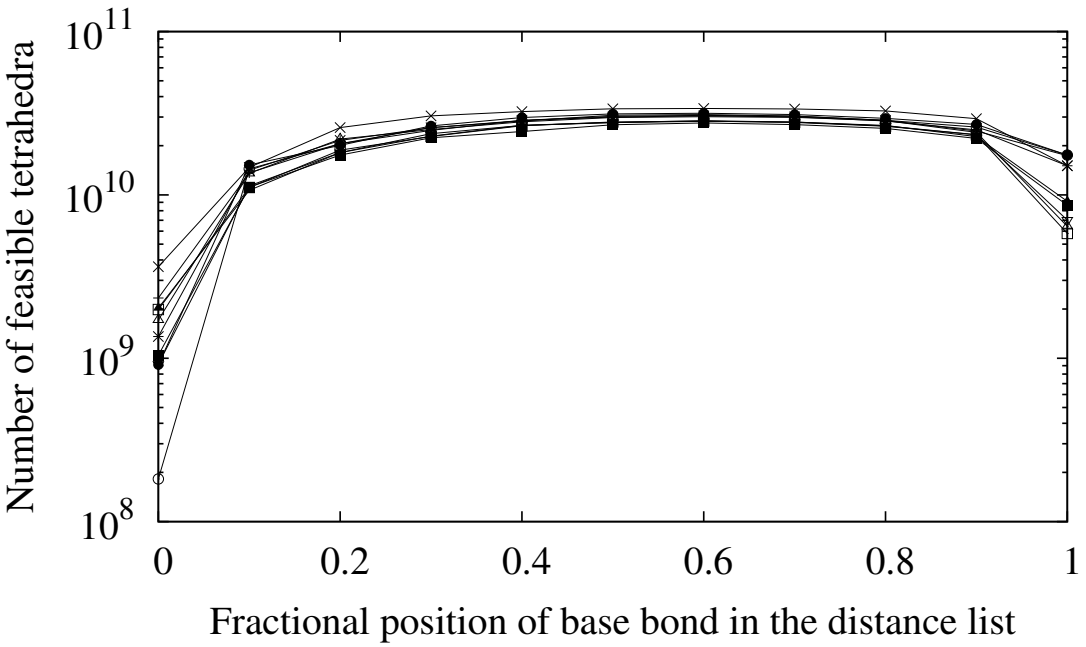

Figure 4.2: Number of feasible tetrahedra using the bonds from a given distance list go up when we choose a larger bond as base for the base triangle. Statistically, using the shortest bond in the distance list as the base bond leads us to the core in the shortest time. This plot shows data from runs using 10 different structures with $N = 20$.

hence, the number of feasible triangles and tetrahedra increases. But the actual number of triangles and tetrahedra in the target structure is the same for any choice of base bond. This is seen in Fig. 4.2, where the number of feasible tetrahedra increases with fractional position of the base bond in the distance list. Hence, statistically, we find a core in the least time if we choose the shortest bond as our base.

Distances are also more likely to satisfy the triangle inequality if they are drawn from a list of comparable, rather than disparate, lengths. Since the base bond is short, a core is more likely to be found quickly by searching over other short distances first (the small-core hypothesis, Fig. 4.3), and including longer distances only as necessary. This is implemented as a window of the W shortest distances in the distance list, and increased periodically as core finding proceeds. Of the ten bonds in the core, the base is fixed, eight are drawn from

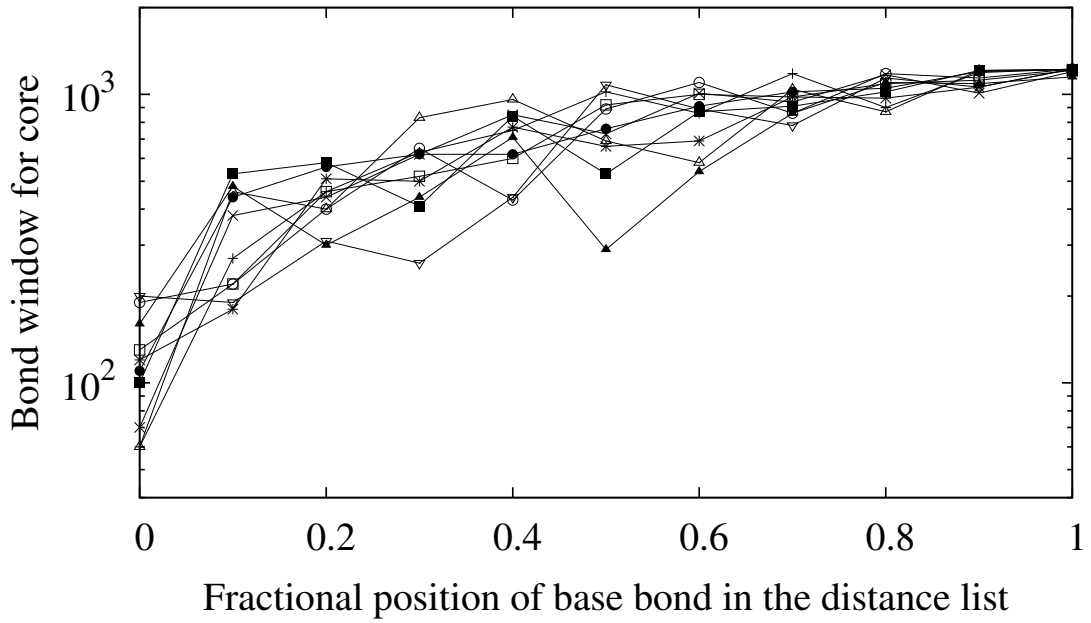

Figure 4.3: Empirical example of the small-core hypothesis. The hypothesis states that there exists a core where at least 9 of the 10 total bonds are drawn from a relatively small window of the shortest bonds in the structure. Varying the base bond's fractional position in the distance list for ten different $N = 50$ structures, core finding shows that using the smallest distance as the base bond reduces the typical size of the window required to find a core by an order of magnitude.

the window and the bridge bond need not be in the window. It is observed that for small structures, a window of size $W \sim N$ is usually sufficient to find a core. Therefore, typical computation time is $\tau_{core} \sim \binom{N}{8} ln(N) \sim N^8 ln(N)$.

From these arguments, supported by Fig. 4.2 and Fig. 4.3, we expect that using the smallest bond as the base will lead to the core finding and buildup in a much shorter time. Fig. 4.4 shows the improvement is about 2 orders of magnitude.

Attempting to find the core for large point sets ($N > 10$) frequently leads to bad cores. Bad cores are overconstrained substructures whose distances are part of the given distance list within a given numerical tolerance, but the substructure is not present in the target structure. This occurs due to finite tolerance when checking for the bridge bond and also finite precision

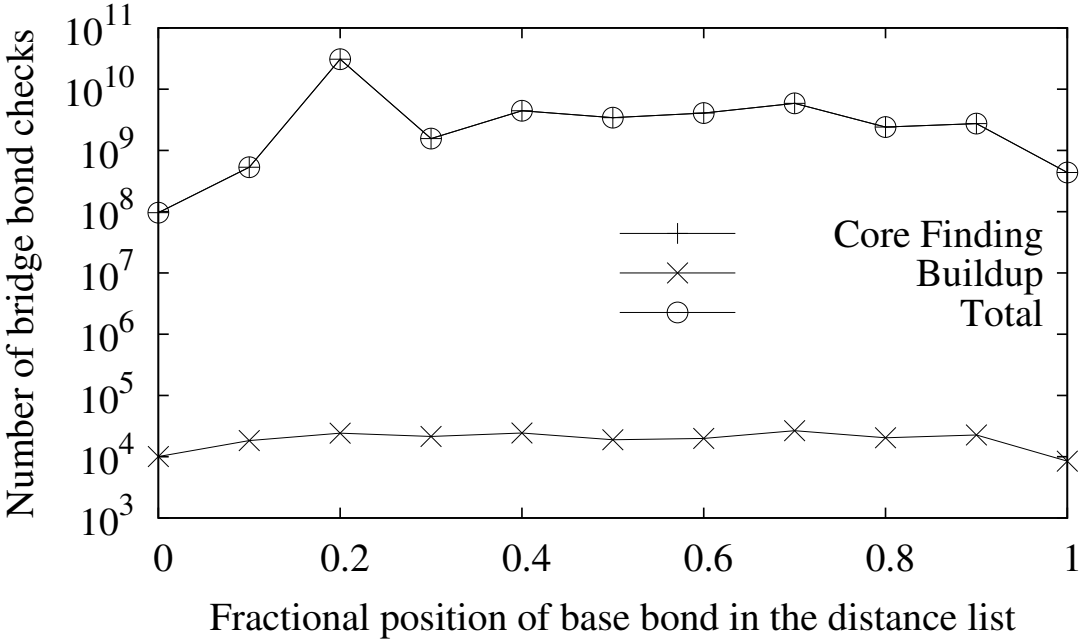

Figure 4.4: Figure illustrating the effect of base bond size on the computational cost (bridge bond checks) of reconstruction for $N = 10$. The plots for the total and core finding steps are nearly indistinguishable because the core finding is orders of magnitude more expensive than buildup. If the smallest bond is chosen as the base, the total computational cost of reconstruction is nearly 2 orders of magnitude lower than larger bonds.

while placing the points using triangulation. Triangles with both small and large distances are likely to have small angles, resulting in a greater loss of numerical precision. A base bond of intermediate length would limit this loss, but is not sufficient to forsake the performance benefits of a small base bond previously outlined. Instead, we try to use all 10 bonds (in the core) as the base bond and check if the corresponding bridge bond is valid or not. We only take cores for which the bridge bond is valid in all of the 10 cases. This check is very good at identifying bad cores.

A structure comparison routine provides another test for bad cores by overlaying the points in the reconstructed structure ($\vec{r}$) with the points in the target structure ($\vec{R}$) and calculates an overlay error,

$$\epsilon_{overlay} = \sum_i |\vec{r}_i - \vec{R}_i|^2. \tag{4.1}$$

This error tells us if a given (sub)structure is part of the target structure. This proves useful for testing purposes only, as in principle the latter remains unknown. It is also used to verify the correctness of the final structure.

If the buildup step fails to add any points after looping over a certain number of bonds from the distance list, likely due to a bad core, then we discard the substructure. We resume the core finding step and attempt another buildup from a new core. This heuristic helps identify probable bad cores efficiently.

It is important to choose an appropriate tolerance when checking if the bridge bond is part of the distance list. Using a very loose tolerance leads to a large number of bad cores. On the other hand, using a very tight tolerance excludes good cores, due to finite precision when carrying out the triangulation to place the points in our substructure. We use floating point numbers for the input distances which have an accuracy of 18 digits. We found that a relative

tolerance of 10^{-12} is optimal to retain good cores and filter out bad ones. When trying to place points which are nearly collinear to the base bond a loss of precision is observed due to small angles, as discussed earlier. In such situations we relax the tolerance when checking for the bridge bond.

To check the validity of a new point while doing buildup, in addition to the bridge bond check, we check 10 additional distances that it creates with the points already in the substructure. Only if these are part of the distance list does the new point get added to the structure. The 4 bond lengths (three from the new tetrahedron created and the fourth is the bridge) that were used are removed from further reconstruction. This reduces the list of available distances by 4. After placing the n^{th} point, updating all $(n-1)$ distances created between the new point and the points already in the substructure reduces the number of available distances substantially. However, due to the computational cost of this update procedure, we see only a small speedup in the buildup routine.

If after buildup, the structure has fewer than the desired number of points (N), we relax the tolerance for the bridge bond checks and rerun the procedure. If the structure remains incomplete, we choose a different bond as the base bond and attempt another buildup. After reconstruction, we calculate the distance error, which is based on the agreement between the given distance list and the distances derived from the final structure.

The Tribond algorithm ran for $N = 6, 7, 8, ..., 12$ and the computational cost was measured in a system-independent manner by counting the number of bridge bond checks while placing a point in both the core finding and buildup steps (Fig. 4.5). The time required for buildup is several orders of magnitude less than that for core finding. As core finding is computationally very expensive, complete reconstruction was attempted only for small structures. Assuming that the core is given, buildup was attempted for larger structures

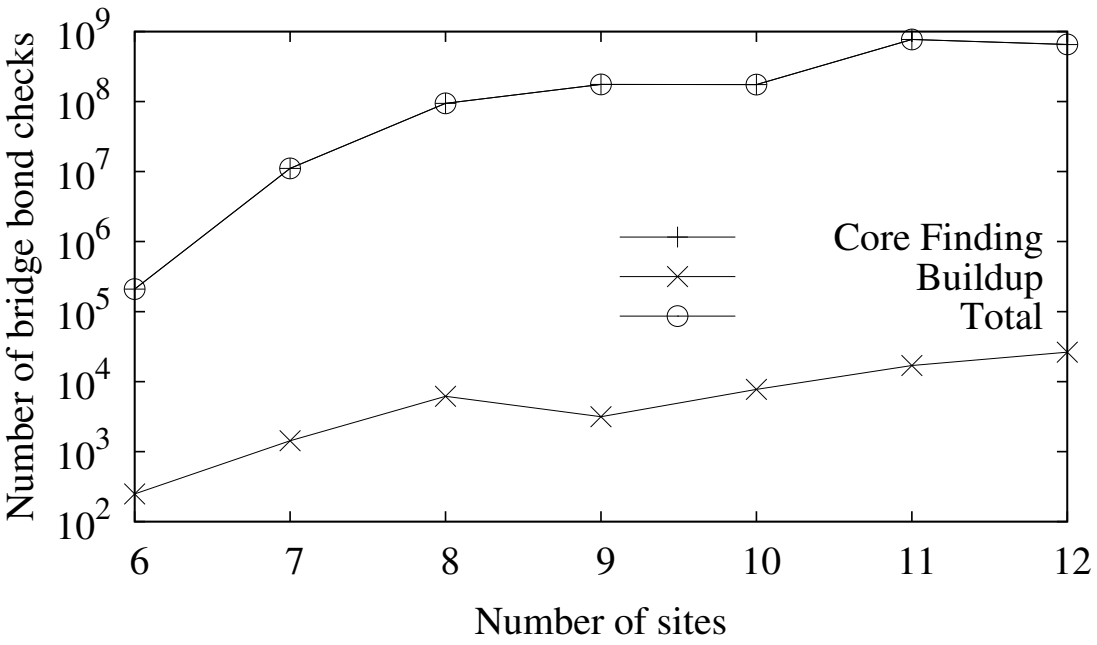

Figure 4.5: Experimental results for a series of reconstructions from distances lists generated from random point sets in three dimensions. The computational cost (bridge bond checks) for finding the core, performing buildup and their total is presented as a function of the number of points. The plots for the total and core finding steps are nearly indistinguishable because core finding takes orders of magnitude more time than buildup. Each point on the plots is the median value from 10 different instances of random point sets.

having size $N = 25, 50, 75, 100$. The timing results are shown in Fig. 4.6.

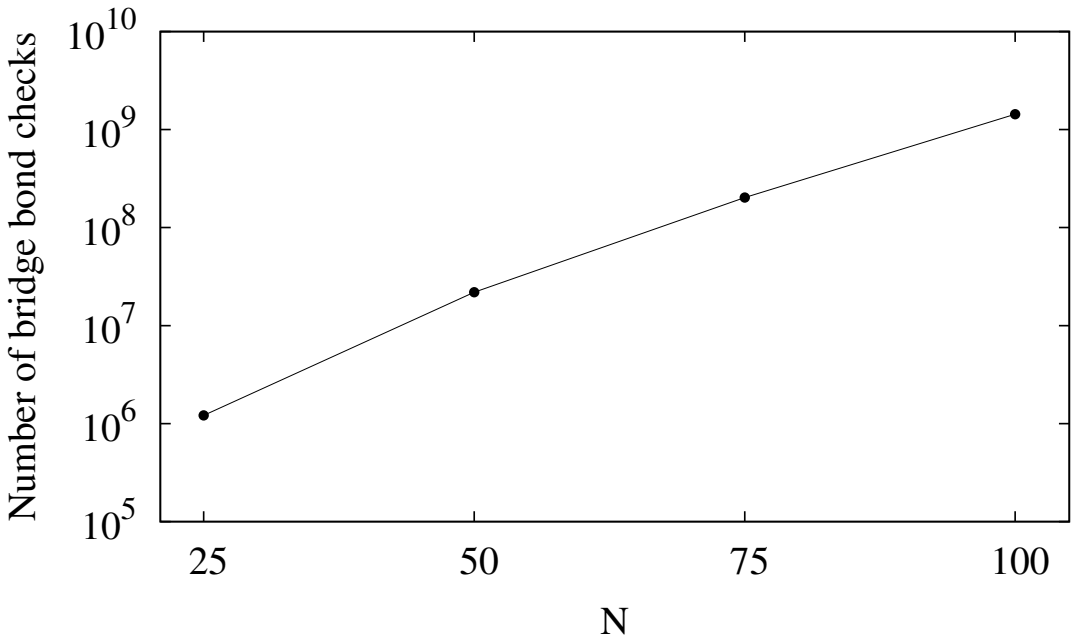

Figure 4.6: Experimental results for a series of reconstructions from distances lists generated from random point sets in three dimensions. The computational cost (bridge bond checks) for performing buildup is presented as a function of the number of points. Each point on the plots is the average over 10 different instances of random point sets. We find that the buildup time scales as $\tau_{buildup} \sim N^{4.98}$.

4.2 Applications

Tribond 3D was used to solve the structure of some well known organic molecules and the results were compared with those from the Liga algorithm. While Liga was able to do the complete reconstruction for 19 structures, Tribond was able to do so for 14 of them. If a starting 5 atom structure is given, then Tribond is successful in reconstructing 56 structures, while Liga can only reconstruct only 21. These organic molecules have a large number of unique distances and are of intermediate symmetry. Hence, Tribond is more successful in doing the buildup as compared to Liga.

Tribond first attempted core finding and buildup for 48 hours. If the core finding step

did not succeed, then only the buildup was attempted for 2 hours.

Table 4.1: The following table lists the results, where success is denoted by 1 and failure by 0. CF stands for core finding and BU for buildup.

structure	N	Liga-BU	Liga	Tribond-CF	Tribond-BU	Tribond
2me-3ane	14	1	1	1	1	1
adrenln	24	0	0	0	1	0
alanine	13	1	1	0	0	0
arginine	27	0	0	0	1	0
asa	21	0	0	0	1	0
asparagine	17	0	0	0	1	0
aspartate	15	0	0	0	1	0
aspirin	21	0	0	0	1	0
b-10ane	47	0	0	0	1	0
b-11ane	71	0	0	0	1	0
borane01	44	0	0	0	1	0
butane-a	14	1	1	1	1	1
butane-e	14	1	1	1	1	1
butane-g	14	1	1	1	1	1
caffeine	24	0	0	0	1	0
carboplatin	23	0	0	0	1	0
cdecalin	28	0	0	0	1	0
cholicac	69	0	0	0	1	0
cisplatin	11	1	1	1	1	1
cubane	16	1	1	0	1	0
cy-5ane	15	0	0	1	1	1
cystine	14	1	0	0	1	0
d-7ane	32	0	0	0	1	0
ethane	8	1	1	1	1	1
ethanol	9	1	1	1	1	1
glutamate	19	0	0	0	1	0
glutamine	20	0	0	0	0	0
glycine	10	1	1	1	1	1
heptane	23	1	1	0	1	0
histidine	20	0	0	0	1	0
i-cy5ane	21	0	0	0	1	0
isoleucine	22	0	0	0	0	0
leucine	22	0	0	0	1	0
lsd	49	0	0	0	1	0
lysine	25	0	0	0	1	0
menthol	31	0	0	0	1	0
methane	5	1	1	1	1	1

Table 4.1 (cont'd).

structure	N	Liga-BU	Liga	Tribond-CF	Tribond-BU	Tribond
methanol	6	1	1	1	1	1
methionine	20	0	0	0	0	0
mustardgas	15	1	1	0	1	0
nicotine	26	0	0	0	1	0
octane	26	1	1	0	1	0
pbpc	41	0	0	0	1	0
pentane	17	1	1	1	1	1
phenylalanine	23	0	0	0	0	0
piperine	43	0	0	0	1	0
proline	17	0	0	0	1	0
propane	11	1	1	1	1	1
qcyclene	15	0	0	0	1	0
quinine	48	0	0	0	1	0
r2bu-ts	31	0	0	0	1	0
rr-tacid	16	0	0	0	1	0
rs-tacid	16	0	0	0	1	0
s34ane	22	0	0	0	1	0
serine	14	1	0	0	1	0
srdimecp	15	0	0	0	1	0
ssdimecp	15	0	0	1	1	1
threonine	17	0	0	0	1	0
tnt	21	0	0	0	1	0
transplatin-hack	11	1	1	0	1	0
tryptophan	27	0	0	0	1	0
tyrosine	24	0	0	0	1	0
valine	19	0	0	0	0	0
valium	33	0	0	0	1	0
vanillin	19	1	1	0	0	0
total		21	19	14	56	14

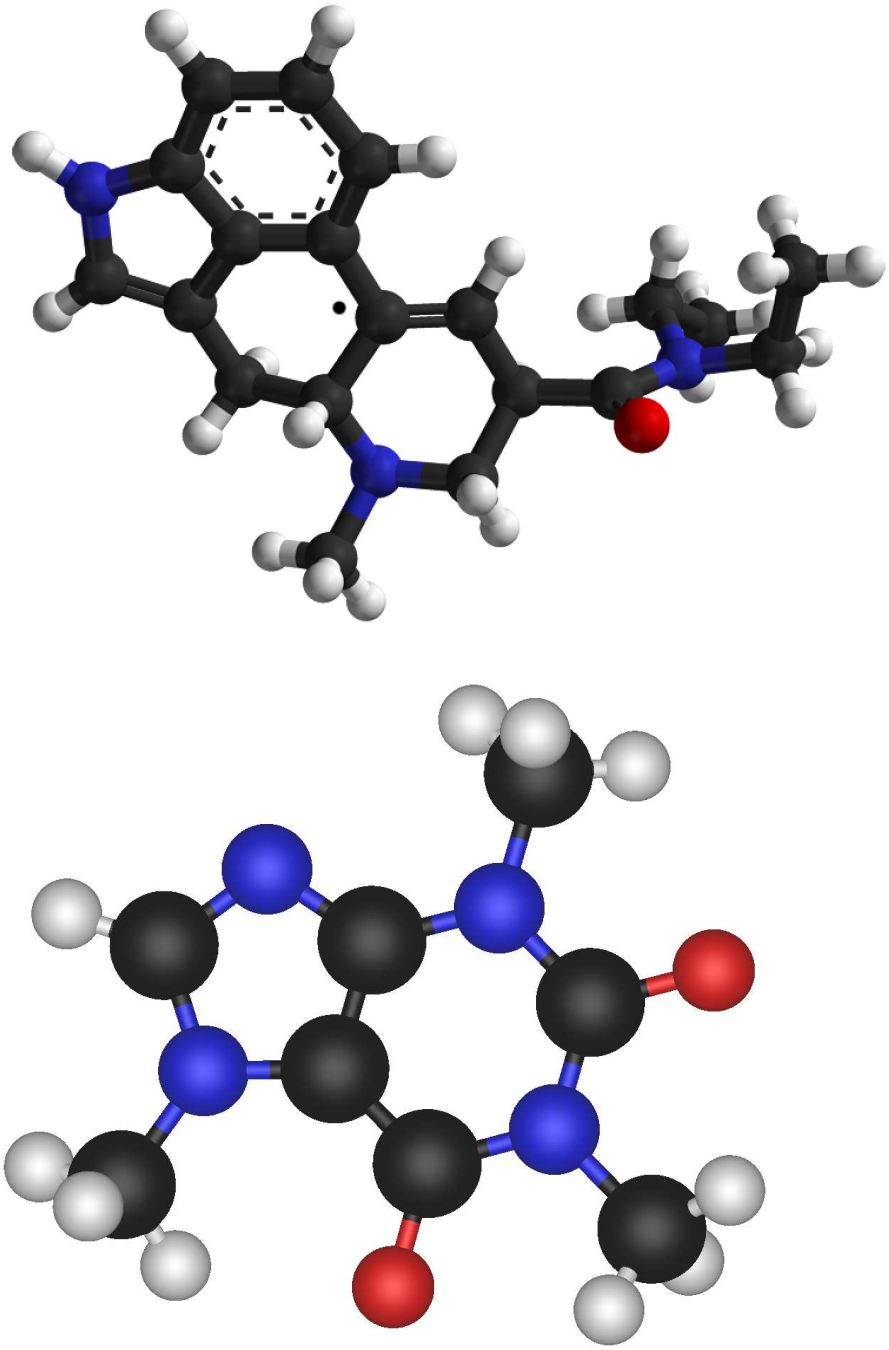

Figure 4.7: Buildup for LSD (top) and Caffeine (bottom) molecules was done in 48.9 seconds and 2.1 seconds respectively.

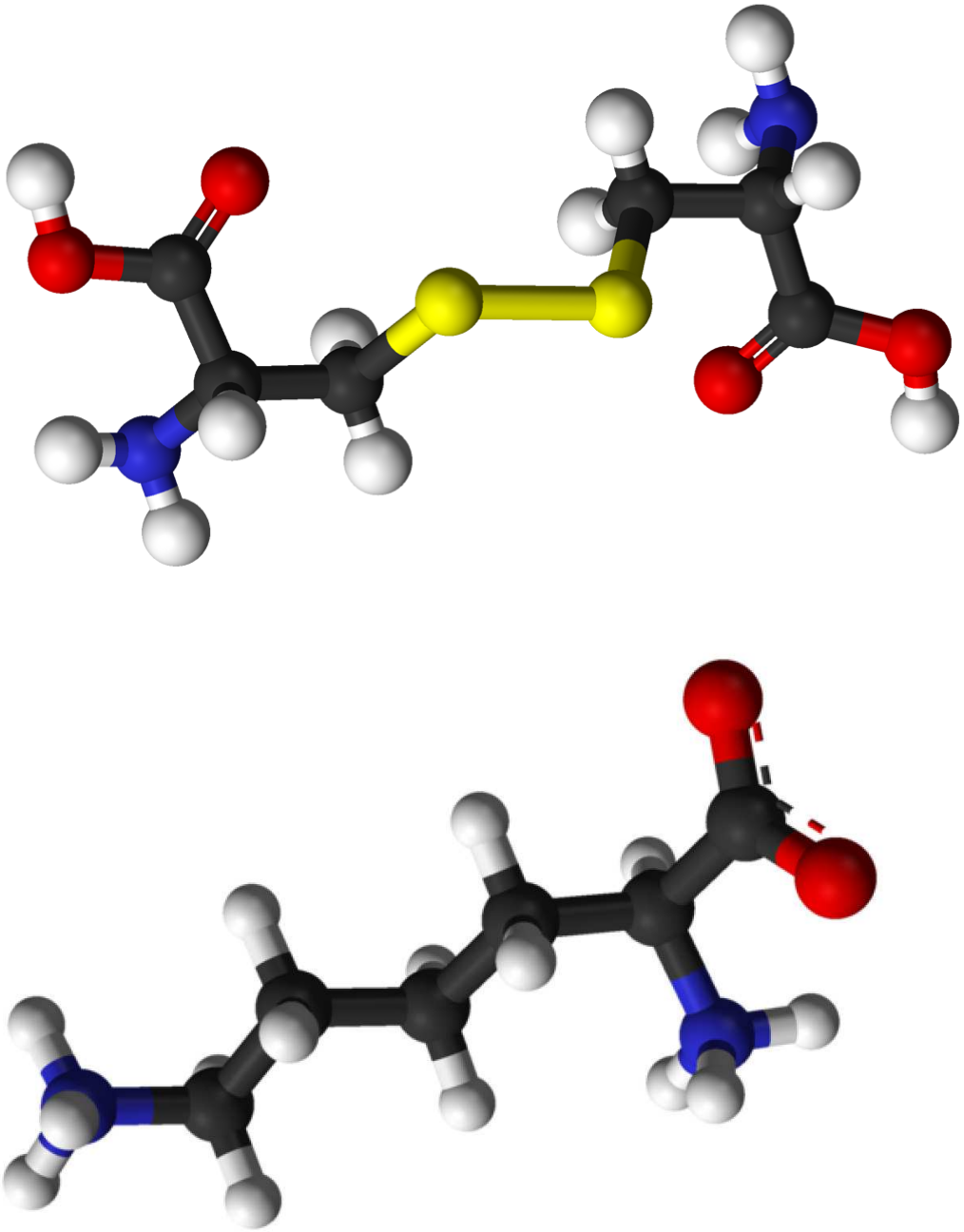

Figure 4.8: Buildup for Cystine (top) and Lysine (bottom) molecules was done in 0.24 seconds and 2.8 seconds respectively.

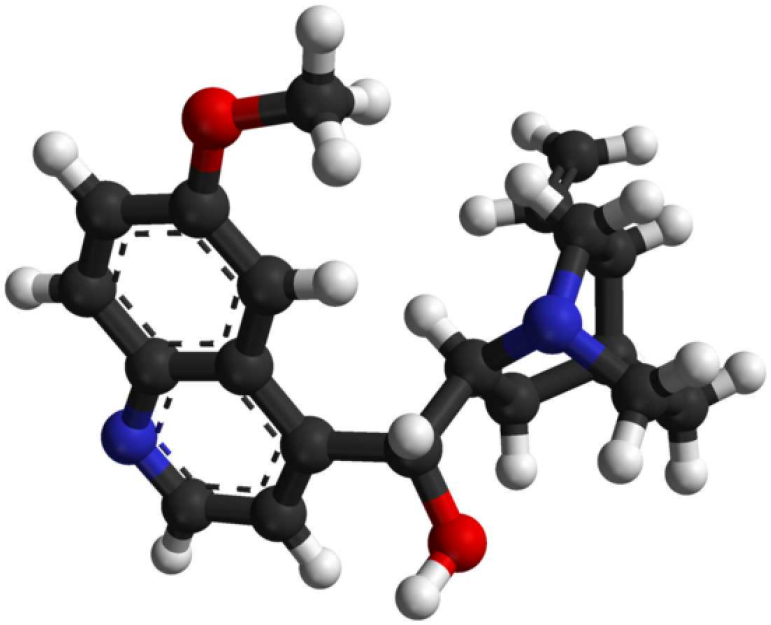

Figure 4.9: Buildup for Quinine molecule was done in 84.4 seconds.

4.2.1 Reconstruction from an imprecise distance list

Thus far distances have been known to a precision of about 18 digits, such that in our trials substructures are indistinguishable (to within a very small tolerance) to those consistent with a theoretical distance list of infinite precision. When we have an imprecise distance list, many small sub structures may be consistent with the distance list, though they may not be part of the target structure. The inverse problem under these conditions is significantly more challenging, both theoretically and practically. We have attempted to address structure buildup from a known core with an imprecise distance list in the case of random point sets. The modification of the original buildup algorithm described in Section 4.1 is as follows.

Assume a known substructure that serves as the core (seed) for reconstruction. The modified buildup step (adding a point) now has multiple stages; it uses a pool of candidate points which have low error with respect to the current substructure, and adds the two candidates

which jointly lead to the lowest cost substructure. Because the pool examines many possible ways to grow the substructure, the likelihood of adding bad points is reduced. Adding two points at once is justified empirically, as this appeared to make the most acceptable trade off between success and run-time. The detailed steps follow.

1. Define an empty pool that saves the coordinates of $k_1 \leqslant 20$ candidate points to add to the current substructure. Associated with each candidate is the cost of the new substructure if that point were added. Populate the pool with candidate points. First, randomly choose a triangle in the current substructure. Generate all tetrahedra using distances from the distance list which share the chosen triangle. Calculate the cost for each candidate point (the new vertices). If this cost is below a user-defined threshold add it to the pool, and if the pool exceeds its maximum size remove the worst candidate. The threshold significantly improves runtime without affecting the final structure.

2. Randomly choose another triangle in the current substructure and generate a new pool of size $k_2 \leqslant 20$ as described above.

3. Select the best candidates from either pool to make a combined pool with $k \leqslant 20$ points.

4. Calculate the pair cost for adding 2 candidates to the current substructure for each of the $\binom{k}{2}$ possible pairs.

5. Add the 2 candidates with least pair cost to the substructure. If its size is less than target size then go to step 1.

The results can be seen in Fig. 4.10, which shows the minimum core size needed to reconstruct structures of size $N = 9, 17$ and 25 for different values of the precision (P) of the

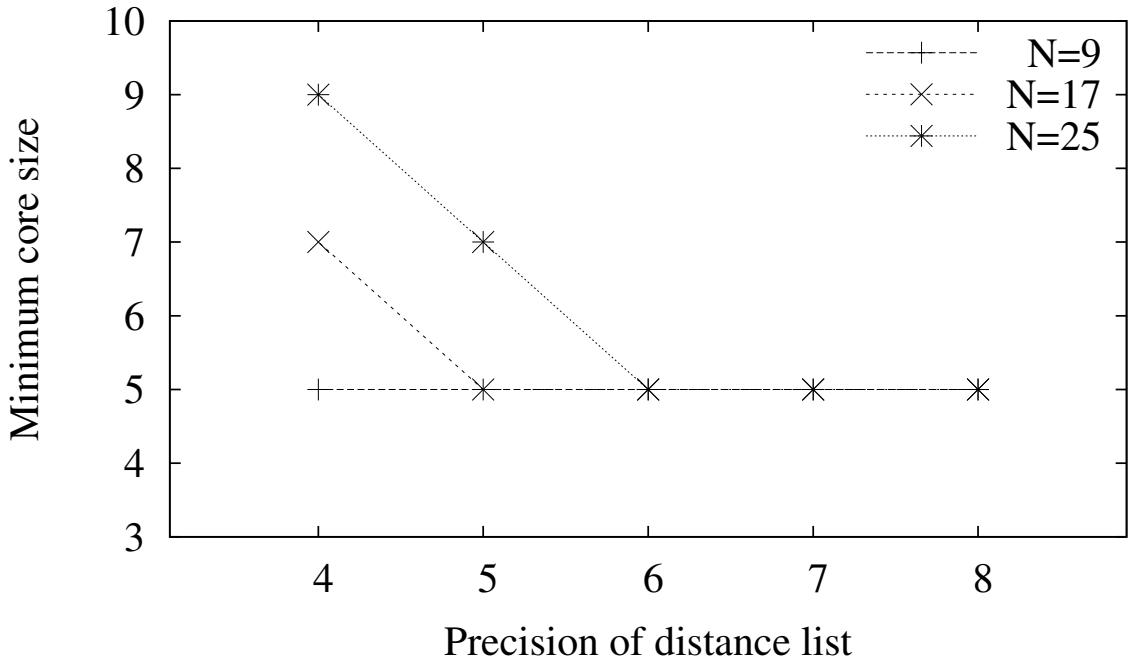

Figure 4.10: Plot of minimum core size vs precision of the input distance list for $N = 9, 17$ and 25. We can see that a bigger core is needed for a less precise distance list. The typical run time for $N = 9, 17, 25$ was about 1 second, 20 minutes and 15 hours respectively, on a computer with a 2.2 GHz processor and 2 GB of memory.

input distance list. The criterion for success was that the algorithm successfully reconstructs at least 5 of 10 different random point sets. It can be seen that as the distances become less precise, a core of a larger size is needed for successful reconstruction.

A notable case with imprecise distances is the PDF of nanostructured materials, which can give distance lists with uncertainties of order 0.01Å. For a typical nanoparticle of size $\sim$ 15Å, this means the input distances from experimental data will have 3-4 digits of precision and the algorithm is a promising approach. Future work will involve working on an algorithm that can better deal with missing, incorrect or less precise distances. Chemical information like the presence of functional groups (aromatic rings, etc) can serve as a core and also help construct the larger core necessary for buildup in the case of less precise distances. Some approaches to these issues are discussed in the context of reconstructing high symmetry

nanostructures from experimental PDF data using the Liga algorithm [28, 29, 30]. A hybrid approach using Tribond (low symmetry) and Liga (high symmetry) could potentially solve structures of intermediate symmetry.

4.3 Summary

In this chapter, the details of the Tribond 3D algorithm for the reconstruction of low symmetry structures represented by random point sets were presented. The Tribond 3D algorithm consists of two steps: core finding and buildup. The core is the smallest substructure with at least one over-constrained bond and is of size 5 in three dimensions. Choosing the smallest bond as the base bond for reconstruction had a dramatic improvement in performance. Computational cost of core finding was orders of magnitude more than buildup. Tribond 3D was able to reconstruct random point sets in 3 dimensions of size about ten in a short amount of time and if the core is assumed to be given it is able to complete the reconstruction for structures with size $N = 100$ in about 2 hours on a desktop computer. A modified approach was presented for the buildup step with less precise data and given a known substructure. As precision decreases, it is clear that we need a substructure of larger size, underscoring the importance of the core finding step.

Chapter 5

Statistical physics of the optimal Golomb ruler

In this chapter, a statistical physics approach to the combinatorial optimization problem of the optimal Golomb ruler (OGR) is taken and the resulting phase transition is studied.

5.1 Statistical mechanics formulation

We define the Golomb lattice gas on a chain of length L, where each site $i = 1, 2, ..., L$ of the chain has a lattice gas variable y_i that may take the values zero or one. If $y_i = 1$ the site i is occupied while if $y_i = 0$ it is unoccupied. For example one of the two degenerate $n = 4$ OGR states has marker set $\{m\} = \{0, 1, 4, 6\}$. The lattice gas representation of this OGR is a chain of length $L = 7$, with site occupancies $\{y_i\} = \{1, 1, 0, 0, 1, 0, 1\}$. We introduce the Golomb lattice gas Hamiltonian which consists of a chemical potential term and an energy term associated with the Golomb ruler constraints.

$$H_1 = -\mu \sum_{i}^{L} y_i + \gamma \sum_{j \neq i, l \neq k}{}' y_i y_j y_k y_l \delta(|j - i| - |l - k|). \tag{5.1}$$

The chemical potential (μ) is the amount by which the energy of the system would change if an additional site were occupied. The first term tries to maximize the density of our lattice

gas. The prime on the second sum indicates that degenerate cases where both $j = l$ and $i = k$ are omitted. The second term imposes the Golomb ruler constraints that the distances between occupied sites should be non-degenerate. The parameter γ tunes the constraint energy and in the limit $\gamma/\mu \to \infty$, and at low temperature, OGR states are the ground states of this Hamiltonian. This is due to the fact that OGR states make no contribution to the constraint energy, while the lowest energy state has the highest density, ensuring the optimal energy from the chemical potential term.

An alternative formulation is to define a distance degeneracy function, $D(d)$, in terms of the occupancy variables y_i through,

$$D(d) = \sum_i y_i y_{i+d}$$

that gives the number of times a distance, d, appears in a marker set $\{m\}$. It is clear that we have the property $\sum_d D(d) = n(n+1)/2$. Moreover we also have,

$$\sum_d [D(d)]^2 \geq n(n+1)/2$$

where equality holds *iff* the distances satisfy the uniqueness condition. The uniqueness condition is then imposed by the equation,

$$\sum_d [D(d)]^2 = n(n+1)/2 = \sum_d D(d)$$

This motivates introduction of an alternative energy function,

$$E_2 = \sum_d \Big[(D(d))^2 - D(d) \Big]$$

This energy or cost function is always a positive integer or zero, with zero being correct for OGR marker sets. This leads to the lattice gas Hamiltonian,

$$H_2 = -\mu \sum_{i=1}^{L} y_i + \gamma_2 \sum_{d=1}^{L-1} \Big[(D(d))^2 - D(d) \Big]$$

which in lattice gas variables is,

$$H_2 = -\mu \sum_{i=1}^{L} y_i + \gamma_2 \sum_{d=1}^{L-1} \left(\sum_i y_i y_{i+d} \right) \left(\sum_i y_i y_{i+d} - 1 \right) \tag{5.2}$$

It is easy to show that the Hamiltonians obtained in two different ways are indeed equivalent. In Eq. 5.1 the second summand can be broken into 2 parts: one exhaustive counting over all possible index combinations and the other part which subtracts off the binary terms.

$$\sum_{j \neq i, l \neq k}^{\prime} = \sum_{i,j,k,l} - \sum_{j=i, l=k}$$

The term with the binaries leads to the factor of -1 in Eq. 5.2. In Eq. 5.2 let $j = i + d$, and $k = i$, $l = k + d$, thus we have $l = k + j - i$. The -1 term is for the the binaries which has to be subtracted off because they are the cases where $(i, j) = (k, l)$. The difference of the summands will give us the second summand in Eq. 5.1.

In the low temperature limit and with $\gamma \to 0$ the chemical potential is the only term and the sites are all occupied so the density $\rho = \sum_i < y_i > /L = 1$. At high temperatures, entropy is maximized as empty and occupied sites occur with equal probability so that $\rho = 1/2$. Three limiting states of the Golomb lattice gas are then: (i) The dense phase $\rho = 1$, (ii) the high temperature phase $\rho = 1/2$ and (iii) the OGR state occuring at low temperatures and as $\gamma/\mu \to \infty$. The mean field analysis also confirms these three limiting

states. In the OGR phase, the density approaches zero in the large lattice limit. The way in which it approaches zero can be estimated based on probabilistic reasoning as follows. Define the probability that both sites i and j are occupied to be D_{ij}. This probability can be related to the probability that any other pair of sites (k, l) share the same distance through,

$$D_{ij} = \prod_k \left(1 - D_{k,j+k-i}\right)$$

Within a uniform approximation, this reduces to $D = (1-D)^L$, or $D^{1/L} = e^{lnD/L} = 1 - D$. Solving to leading order gives, $D \sim 1/L$. Since average density is $\rho = n/L$ and $D \sim \rho^2$, we have $(n/L)^2 \sim 1/L$, so that $n \sim L^{1/2}$, which is consistent with rigorous bounds derived from analysis of Sidon sets [47]. A rigorous upper bound on the length of optimal Golomb rulers, $m_n \leq n(n+1)$, indicates that the density of the high constraint ground state goes to zero at large L as $\rho_{OGR} \propto a/L^{1/2}$. Now we explore the statistical physics of the Hamiltonian using an effective medium approach that contains the exact OGR state as a limiting solution.

5.2 Mean field approach

The Golomb lattice gas mean field theory is developed in the usual way, by writing $y_i = \rho_i + \delta y_i$, where $\delta y_i = y_i - \rho_i$ is the fluctuation and $\rho_i =< y_i >$ is the average density at site i. Substitute this into Eq. 5.1.

$$H_{MF} = -\mu \sum_i (\rho_i + \delta y_i) + \gamma \sum_{j \neq i, l \neq k}{}' (\rho_i + \delta y_i)(\rho_j + \delta y_j)(\rho_k + \delta y_k)(\rho_l + \delta y_l)\delta(|j-i| - |l-k|)$$

Now, keep terms having δy_i and ignore higher order terms.

$$H_{MF} = -\mu \sum_i (\rho_i + \delta y_i) +$$
$$\gamma \sum_{j\neq i, l\neq k}{}' (\rho_i\rho_j\rho_k\rho_l + \rho_j\rho_k\rho_l\delta y_i + \rho_i\rho_k\rho_l\delta y_j + \rho_i\rho_j\rho_l\delta y_k + \rho_i\rho_j\rho_k\delta y_l)\delta(|j-i| - |l-k|)$$

Then substitute $\delta y_i = y_i - \rho_i$ to get an equation which only has terms in y_i and ρ_i.

$$H_{MF} = -\mu \sum_i y_i +$$
$$\gamma \sum_{j\neq i, l\neq k}{}' (-3\rho_i\rho_j\rho_k\rho_l + y_i\rho_j\rho_k\rho_l + y_j\rho_i\rho_k\rho_l + y_k\rho_i\rho_j\rho_l + y_l\rho_i\rho_j\rho_k)\delta(|j-i| - |l-k|)$$

Using the symmetry of the variables, we get the following equation.

$$H_{MF} = -\mu \sum_i y_i + \gamma \sum_i (4y_i - 3\rho_i)\alpha_i \tag{5.3}$$

where

$$\alpha_i = \sum_{j\neq i, l\neq k}{}' \rho_j\rho_k\rho_{j+k-i}. \tag{5.4}$$

Alternatively

$$H_{MF} = \sum_i H_i, \tag{5.5}$$

where

$$H_i = -\mu y_i + \gamma(4y_i - 3\rho_i)\alpha_i. \tag{5.6}$$

The density at a site may then be found using

$$\rho_i =< y_i >= \sum_{y_i} y_i e^{-\beta H_{MF}} / [\sum_{y_i} e^{-\beta H_{MF}}] \tag{5.7}$$

$$\rho_i = \frac{0 + e^{-\beta[-\mu+\gamma(4-3\rho_i)\alpha_i]}}{e^{-\beta[\gamma(-3\rho_i)\alpha_i]} + e^{-\beta[-\mu+\gamma(4-3\rho_i)\alpha_i]}} \tag{5.8}$$

yielding the Golomb lattice gas mean field equations,

$$\rho_i = \frac{e^{\beta\mu-4\beta\gamma\alpha_i}}{1 + e^{\beta\mu-4\beta\gamma\alpha_i}} \tag{5.9}$$

The partition function for a lattice site is given by

$$Z_i = \sum_{y_i=0,1} e^{-\beta H_i} = e^{-\beta[\gamma(-3\rho_i)\alpha_i]} + e^{-\beta[-\mu+\gamma(4-3\rho_i)\alpha_i]} \tag{5.10}$$

Please note that this is also the denominator in Eq. 5.8.

$$Z_i = e^{3\beta\gamma\rho_i\alpha_i}(1 + e^{\beta\mu-4\beta\gamma\alpha_i}) \tag{5.11}$$

The Golomb lattice gas mean field free energy is given by

$$F = -kTln(Z) = -kTln(\prod_i Z_i) = -kT\sum_i ln(Z_i) \tag{5.12}$$

$$F = -kT\sum_i ln[e^{3\beta\gamma\rho_i\alpha_i}(1 + e^{\beta\mu-4\beta\gamma\alpha_i})] \tag{5.13}$$

Hence, we get

$$F = -3\gamma\sum_i \rho_i\alpha_i - kT\sum_i ln[1 + e^{\beta\mu-4\beta\gamma\alpha_i}] \tag{5.14}$$

The mean field equations Eq. 5.4 and Eq. 5.9 are a coupled set of non-linear equations that have many metastable solutions at large γ/μ and at low temperatures. In this regime, the optimal or equilibrium state of the system is the lowest free energy solution to these equations. The fact that the exact ground state in the OGR regime is known, for $n \leq 26$, it enables a systematic study of the phase diagram and behavior of the MFT equations over the whole phase space.

Mean field theory may also be developed from Eq. 5.2, where a similar analysis leads to,

$$\rho_i = \frac{e^{\beta\mu - 4\beta\gamma_2\alpha'_i}}{1 + e^{\beta\mu - 4\beta\gamma_2\alpha'_i}}$$

or equivalently,

$$\rho_i = \frac{1}{1 + e^{-(\beta\mu - 4\beta\gamma\alpha'_i)}} \tag{5.15}$$

where

$$\alpha'_i = \sum_d \left[2\left(\sum_j \rho_j \rho_{j+d} \right) - 1 \right] (\rho_{i+d} + \rho_{i-d}) \tag{5.16}$$

with the constraints $i + d \leq L,\ i - d \geq 1$ on the quantities ρ_{i+d} and ρ_{i-d} respectively. We use Eq. 5.15 for ρ_i as it needs evaluating only one exponent. It is also more robust as the exponent will not overflow at low temperatures (when β becomes very large). If we look closely at Eq. 5.16, one can see that there are two nested summations to be carried out and α'_i (and hence ρ_i) computation has a $O(L^2)$ complexity. The innermost summation over j for a given value of d, $\left(\sum_j \rho_j \rho_{j+d}\right)$, only 3 terms change when doing the site updates. Hence we store the values for $\left(\sum_j \rho_j \rho_{j+d}\right)$ and use an intelligent update procedure instead of evaluating the entire sum every time. Trading memory for computational time we have an implementation that has a $O(L)$ complexity for α'_i calculation and $O(L^2)$ complexity for

one sweep of ρ_i.

We now try to solve Eq. 5.16 and Eq. 5.15 iteratively. If we do a sequential site update we find that it leads to a trivial oscillation between a fully occupied state with $n_i = 1$ and an unoccupied state with $n_i = 0$. Random site updates prevent this state from occuring. The Golomb lattice gas mean field free energy is given by

$$F = -k_b T \sum_i ln \left[1 + e^{\beta\mu - 4\beta\gamma\alpha'_i}\right] - 3\gamma \sum_i \rho_i \alpha'_i. \tag{5.17}$$

Using the random site update procedure we obtain the results presented in Fig. 5.1. It exhibits a transition from a smooth dependence on γ/μ at low values of $\gamma/\mu < (\gamma/\mu)_c$ to an irregular behavior at higher values of $\gamma/\mu > (\gamma/\mu)_c$. Nevertheless, in both cases the steady solution we find ρ_i at long times is highly dependent on i so in all cases translational symmetry is broken. Moreover for $\gamma/\mu > (\gamma/\mu)_c$, the spatial variation in ρ_i is more extreme and consists of a large number of sites with $\rho_i = 0$, while for $\gamma/\mu < (\gamma/\mu)_c$ sites with $\rho_i = 0$ rarely occur. The trajectories of all sites for $\gamma/\mu > (\gamma/\mu)_c$ are asymmetrical as illustrated by a typical trace as shown in Fig. 5.2. We use the crossing points of the free energy curves and get the phase diagram as shown in Fig. 5.5. Fig. 5.3 gives the scaling behavior of the density in the symmetric phase. In the next section we do an scaling calculations and see that it is close to what is observed in our simulations.

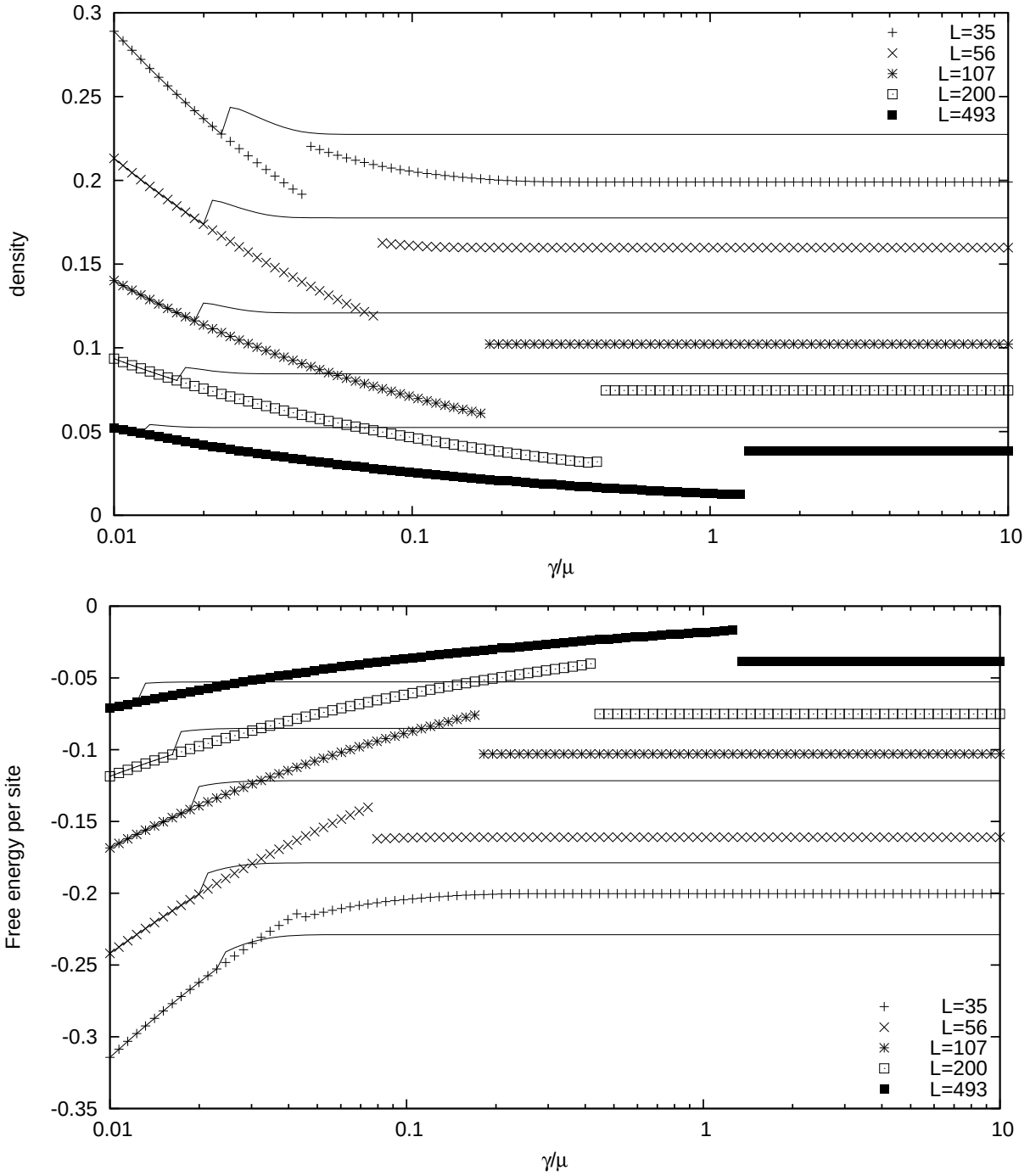

Figure 5.1: Density (top figure) and free energy per site (bottom figure) as a function of γ/μ for $T = 0.2$ and $L = 35, 56, 107, 200, 493$. For each chain length two calculations obtained by iterating through the Golomb lattice gas mean field equations are presented. One trace represented by the symbols is obtained by starting at $\gamma/\mu = 0.01$, choosing a uniform initial condition and then gradually increasing γ/μ. The solid lines are obtained by starting at $\gamma/\mu = 10$, choosing an exact OGR state as the initial condition and then gradually decreasing γ/μ. The mean field solutions are clearly strongly metastable. Though the spinodal lines are strongly size dependent the equilibrium transition is relatively size independent.

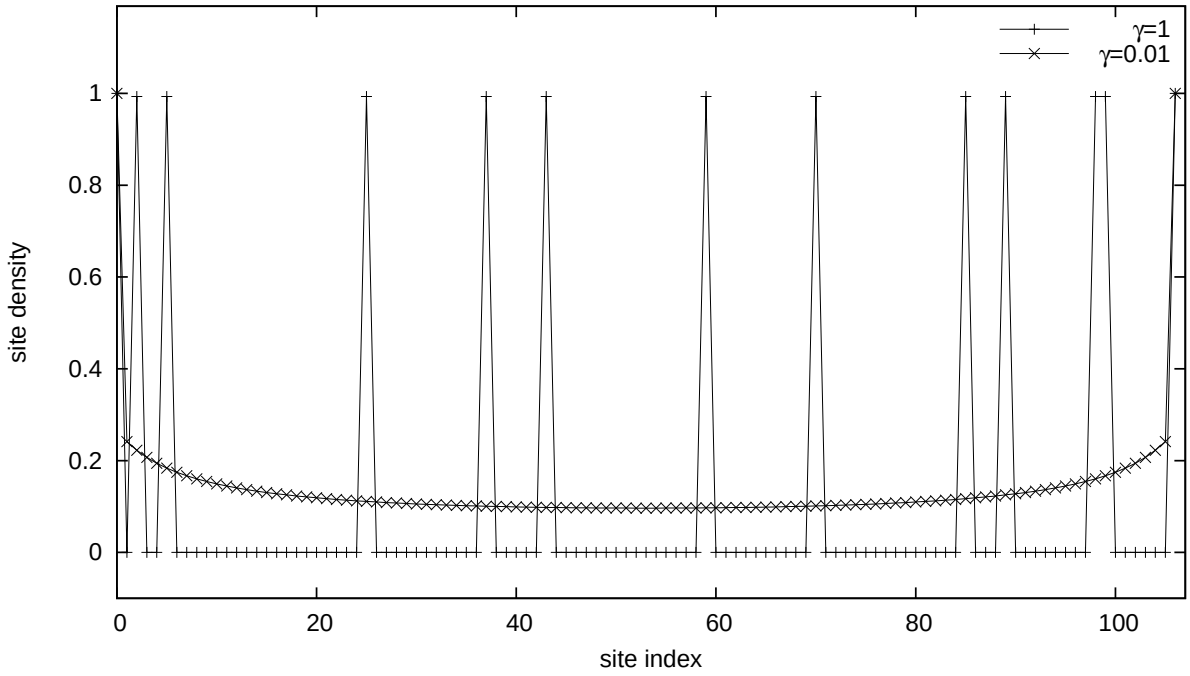

Figure 5.2: The symmetric (crosses) and symmetry broken (plusses) states of the mean field theory for $L = 107$, $T = 0.2$, $\gamma/\mu = 0.01$ and $\gamma/\mu = 1$.

5.3 Asymptotic analysis

5.3.1 Scaling

In the symmetric phase we can use a uniform approximation $\rho_i = \rho_s$ that is justified by Fig. 5.2 to give us an estimate for the free energy per site,

$$f_s \approx -\mu\rho_s + a\gamma L^2 {\rho_s}^4 + T[\rho_s ln(\rho_s) + (1-\rho_s)ln(1-\rho_s)]. \tag{5.18}$$

The optimal density is found from $\delta f/\delta\rho_s$, which gives,

$$-\mu + 4a\gamma L^2 {\rho_s}^3 + T[ln(\rho_s) - ln(1-\rho_s)] = 0. \tag{5.19}$$

so that, when ρ_s is small we can ignore the $Tln(1-\rho_s)$ term to obtain,

$$\rho_s \approx \left(\frac{\mu - Tln(\rho_s)}{4a\gamma L^2}\right)^{1/3}$$

As ρ is small but finite, when we are at sufficiently low temperatures we can also ignore the $Tln(\rho_s)$ term. The value of a can be found by considering the sums in Eq. 5.1 and our estimate is $a = 1/3$. This gives us,

$$\rho_s = \left(\frac{3\mu}{4\gamma L^2}\right)^{1/3} \tag{5.20}$$

The density in the symmetric phase is then predicted to scale as $L^{-2/3}$ and is verified by numerical solutions of the mean field equations Eq. 5.15 and Eq. 5.16 at small values of γ/μ (please refer Fig. 5.3). If we use the above equation and substitute $\rho_s \sim L^{-2/3}$ into Eq. 5.18, we see that at low temperature free energy also has the same scaling behavior ($f_s \sim L^{-2/3}$). We rescale the free energy and it is plotted in bottom part of Fig. 5.4 where we can see that the curves overlap and they follow the same scaling behavior. At low γ/μ there is some deviation for the small rulers which is due to finite size effect and we can see that it matters less and less for large L.

5.3.2 Phase boundary

5.3.2.1 Low temperature

To find the phase boundary, we equate the free energy in the symmetric phase to the free energy of the asymmetric phase ($f_s = f_a$). To get f_s we substitute ρ_s from Eq. 5.20 into the low temperature free energy expression given in Eq. 5.18. In the asymmetric phase the

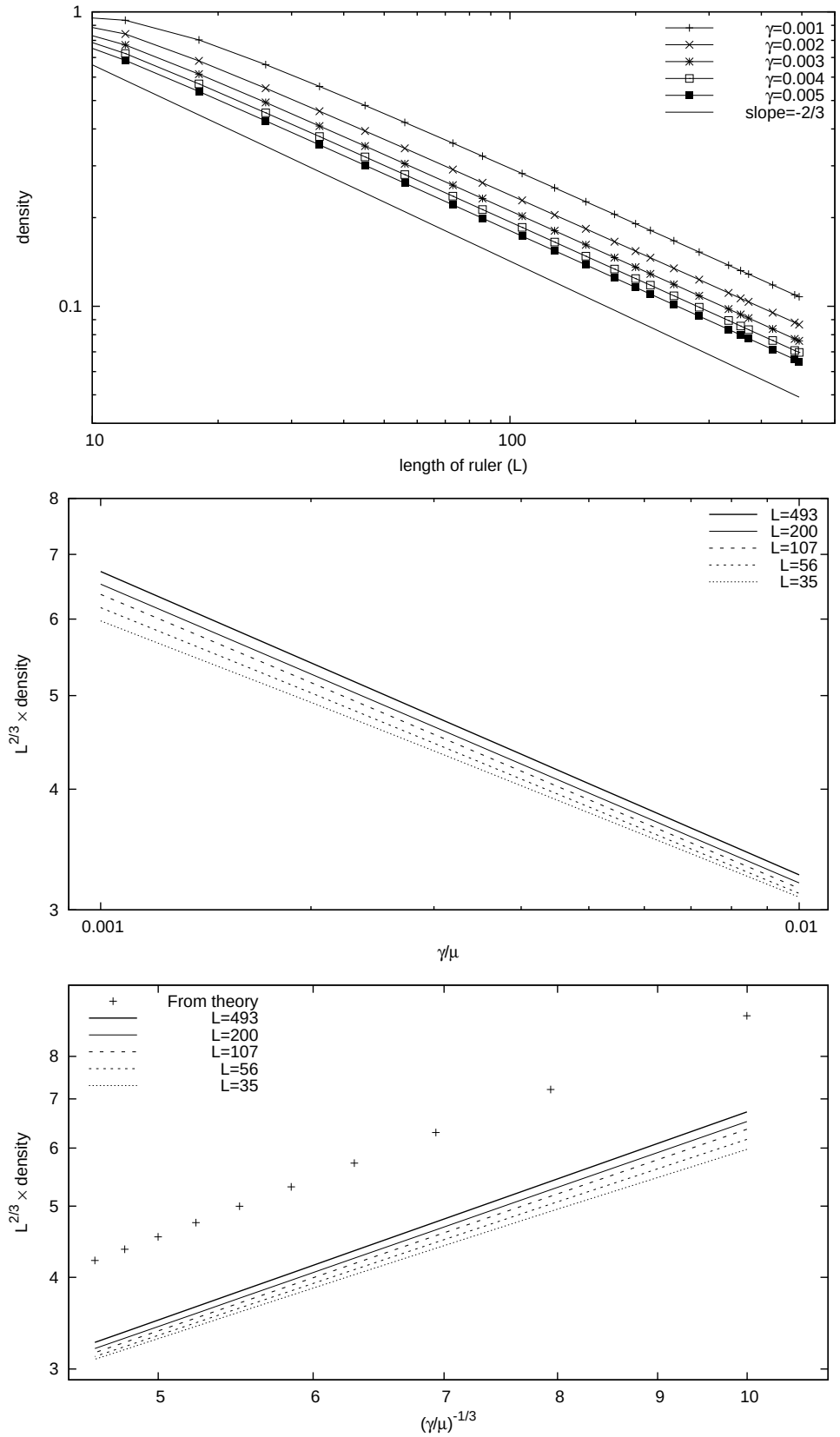

Figure 5.3: Finite size scaling behavior of the density in the symmetric phase for $T = 0.2$ and for different values of γ/μ. The line with slope $-2/3$ is the prediction from scaling theory given by Eq. 5.20.

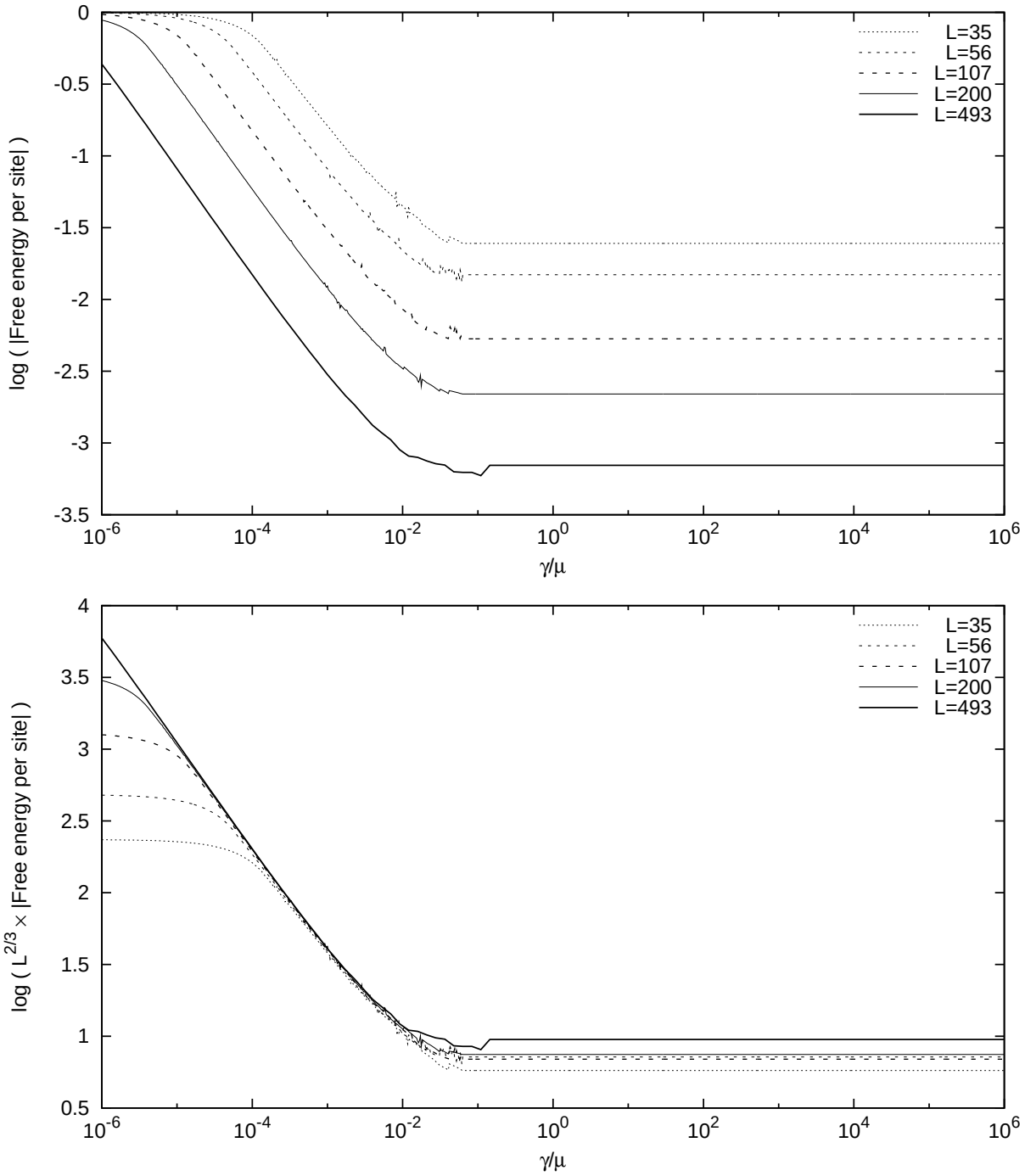

Figure 5.4: Rescaled free energy per site vs γ/μ for $T = 2 \times 10^{-6}$. At low γ/μ and large L, we can see that it follows a $L^{2/3}$ scaling.

ground state is the OGR state. From Eq. 5.1 we can see that the constraint energy is zero and we have $f_a = f_{OGR} = -\mu n_{OGR}/L$. At low temperatures we can ignore the contribution due to entropy and using $n_{OGR} \sim L^{1/2}$, we can get an estimate of the zero temperature phase boundary,

$$\left(\frac{\gamma}{\mu}\right)_c = \frac{81}{256}\frac{1}{L^{1/2}} \tag{5.21}$$

For $L = 493$ we get $(\gamma/\mu)_c = 0.014$ which is close to the intercept plotted in our phase diagram on a log log scale. If we make a plot of $(\gamma/\mu)_c$ vs $1/L$ (Fig. 5.7) and calculate the y- intercept for $T = 2 \times 10^{-6}$ we get $(\gamma/\mu)_c = 0.005$ which is close order of magnitude to what we have seen earlier.

5.3.2.2 High temperature

We start with Eq. 5.18 for the free energy per site in the symmetric phase. Eq. 5.19 gives us the optimal density,

$$-\mu + 4a\gamma L^2 {\rho_s}^3 + Tln\left(\frac{\rho_s}{1-\rho_s}\right) = 0.$$

At large T as ρ_s is small, we can use $\frac{\rho_s}{1-\rho_s} \approx \rho_s$ and in the symmetric phase $\rho_s \sim \frac{1/2}{L^{2/3}}$, we get

$$-\mu + 4a\gamma L^2 {\rho_s}^3 - Tln(2L^{2/3}) = 0$$

As $\mu = 1$, the first term is small compared to the other terms and we can ignore it to get

$${\rho_s}^3 = \frac{T}{\gamma/\mu}\frac{ln(2L^{2/3})}{4aL^2} \tag{5.22}$$

Now expanding the expression for the free energy per site Eq. 5.18 we get

$$f_s \approx -\mu\rho_s + a\gamma L^2 {\rho_s}^4 + T[\rho_s ln(\rho_s) + ln(1-\rho_s) - \rho_s ln(1-\rho_s)]$$

$$f_s \approx -\mu\rho_s + a\gamma L^2 {\rho_s}^4 + T\left[\rho_s ln\left(\frac{\rho_s}{1-\rho_s}\right) + ln(1-\rho_s)\right]$$

Using $\frac{\rho_s}{1-\rho_s} \approx \rho_s$ and $\rho_s \sim \frac{1/2}{L^{2/3}}$ as we did earlier,

$$f_s \approx -\mu\rho_s + a\gamma L^2 \rho_s \frac{1}{8L^2} + T\left[-\rho_s ln(2L^{2/3}) + ln(1-\rho_s)\right] \tag{5.23}$$

When we start at high γ/μ and initialize the ruler with the OGR state then the number of possible states W for the ruler is given by 2^{nOGR}, where n_{OGR} is the number of marks in the optimal Golomb ruler for length L. At high temperature the individual density for occupied sites will be 0.5 and the entropy per site will be given by

$$s = \frac{1}{N} ln(W) = \frac{1}{N} ln(2^{nOGR}) = \frac{n_{OGR}}{N} ln(2) = 2\rho_a ln(2) \tag{5.24}$$

Now the free energy per site in the asymmetric phase at large T when starting with the OGR state is given by

$$f_a = -\mu\rho_a - 2T\rho_a ln(2) \tag{5.25}$$

Now equating f_s and f_a from Eq.5.23 and Eq.5.25 we get the phase boundary

$$\gamma \quad = \quad \left(\frac{8}{a\rho_s}\right) \quad \times \quad \left[\mu(\rho_s - \rho_a) + T((\rho_s ln(2L^{2/3}) - ln(1-\rho_s) - 2\rho_a ln(2))\right] \tag{5.26}$$

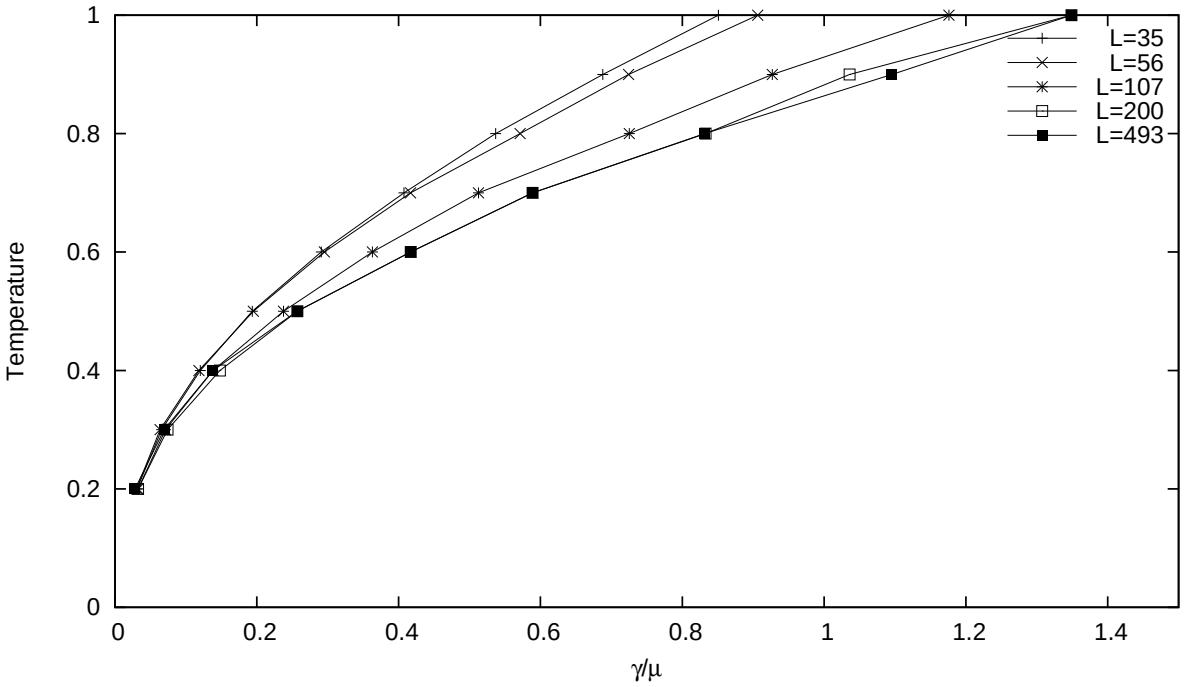

Figure 5.5: The equilibrium phase diagram determined from the crossing points of the free energy curves, such as those shown in the lower half of Fig. 5.1.

As ρ and ρ_{OGR} are small and $\mu = 1$, we can ignore the first term in the above equation. Using $a = \frac{1}{3}$, we get

$$\left(\frac{\gamma}{\mu}\right)_c = 24T\left(ln(2L^{2/3}) - \frac{ln(1-\rho_s)}{\rho_s} - 2\frac{\rho_a}{\rho_s}ln(2)\right) \tag{5.27}$$

This result is qualitatively correct as we see from the mean field runs in Fig. 5.6 that $(\gamma/\mu)_c$ increases with L and $(\gamma/\mu)_c$ also increases linearly with T. From the figure we see that $(\gamma/\mu)_c \approx 5T$ at large temperatures. For $L = 493$ and $T = 200$ our analytic expression gives $(\gamma/\mu)_c \approx 44T$.

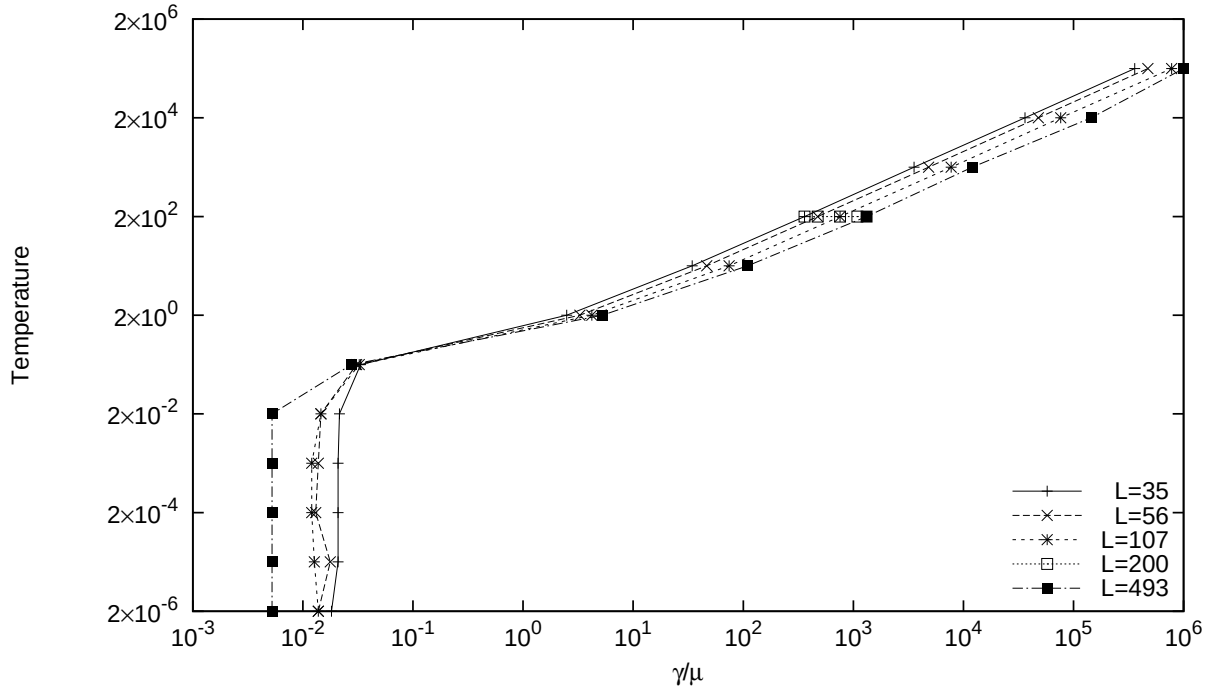

Figure 5.6: In this log-log plot for the equilibrium phase diagram we see that it has a finite non-zero value for the intercept.

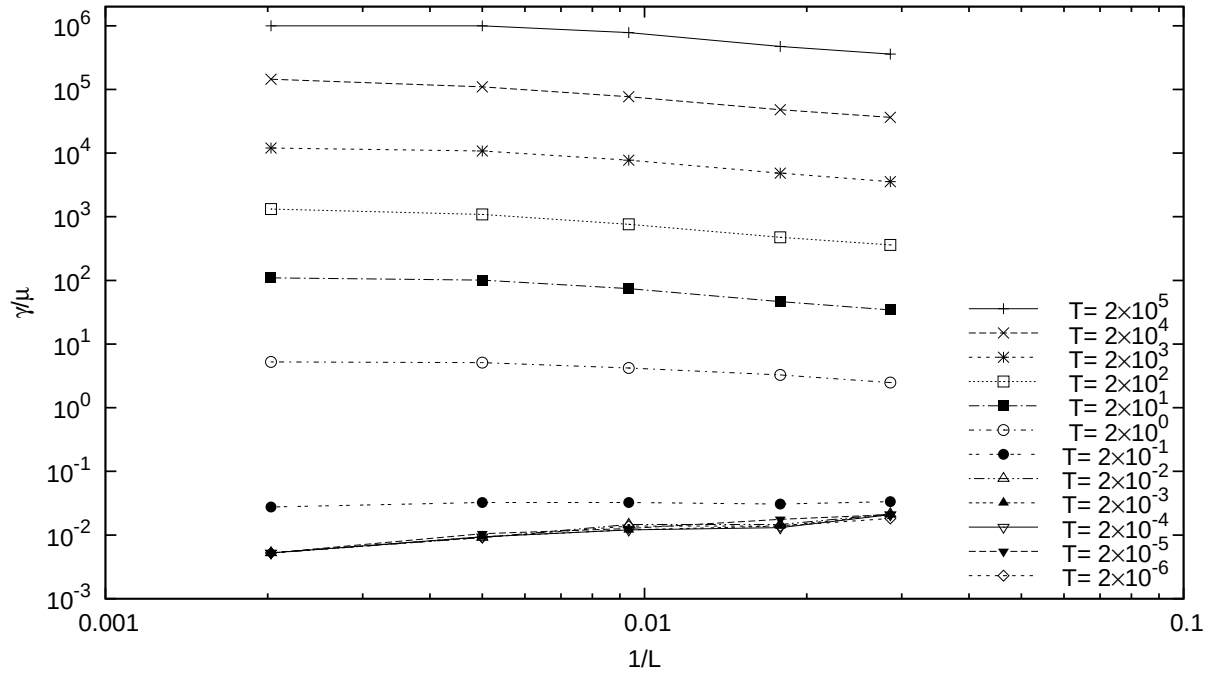

Figure 5.7: Plot showing the dependence of the critical γ/μ on T. At low T, the Y-intercept is $\gamma/\mu = 0.005$ which is the phase boundary for rulers in the large L limit.

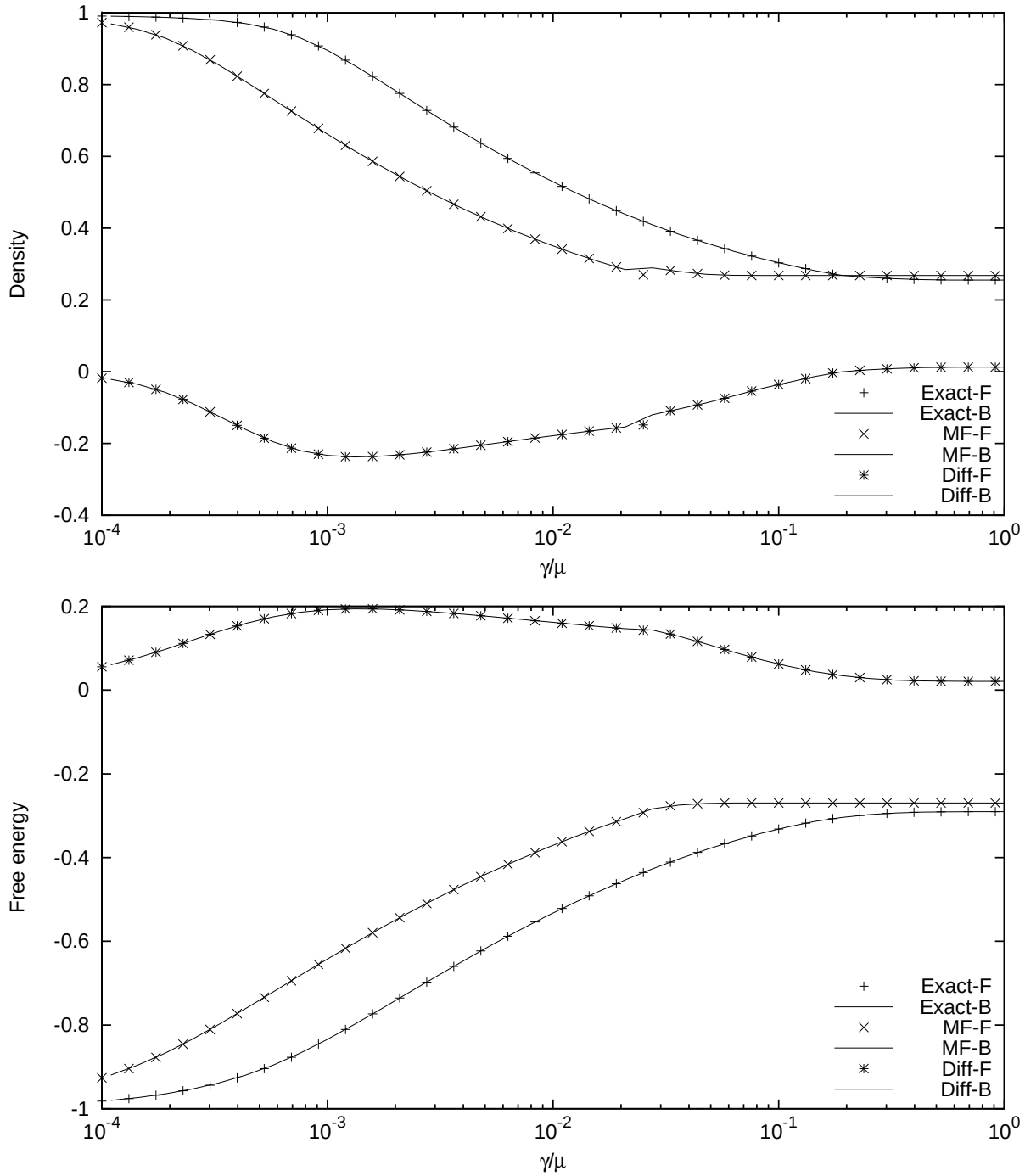

Figure 5.8: Density and Free energy calculations done exactly and using mean field theory for L=26 at T=0.2.

5.4 Exact calculations

We did exact and mean field calculation for $L = 26$ at T=0.2, to get the density and free energy. Fig. 5.8 shows the density and free energy as well as the difference in the mean field and the exact calculations. The end points of the ruler are always set to 1. So for $L = 26$ there are 2^{24} possible states. For the exact calculation we iterate over all these possible states and evaluate the density and free energy. We see that there is a lag in the lines obtained by exact and mean calculations and it is because of the hysteresis.

5.5 Search for OGR

Homotopy methods [76] have been used in statistical physics to obtain the global minimum. We tried to use a similar approach to find the optimal Golomb ruler. We start with a value of γ/μ very close to the phase boundary such that the ruler in the symmetric phase and then we gradually increase γ/μ so that there is a phase transition and it goes into the asymmetric phase. As the optimal Golomb ruler state is the global minimum, we were expecting that the asymmetric phase would be this global minima, but because of metastability this approach was only successful only for small rulers.

5.6 Symmetric theory

The lattice gas formulation of OGR starts with defining variables $n_i = 0, 1$ on a one dimensional chain where $i = 0, 1, 2...L$. Setting $n_i = 1$ for values of i are in an OGR marker set, with the other values of i having $n_i = 0$ maps an OGR solution to a lattice gas configuraiton. To construct the OGR lattice gas Hamiltonian, we need to ensure that the repeated

distances between the lattice gas particles are unfavorable. We define l to label a distance, so that $1 \leq l \leq L$ and we define the *degeneracy* of the distance l, to be D_l. A valid Golomb ruler must have degeneracy $D_l = 0, 1$. If a higher degeneracy were to occur, the distances would not be unique and the marker set would not be a maximally irregular set. A little thought reveals that the degeneracies D_l are related to the lattice gas variables n_i through the relation,

$$D_l = \sum_{i=0}^{L-1} n_i n_{i+l}. \tag{5.28}$$

When the degeneracies D_l are summed over all l, we must have the total number of distances, so that for any configuration $\{n_i\}$, we have the constraint,

$$\sum_{l=1}^{L} D_l = \frac{1}{2} m(m-1). \tag{5.29}$$

Now we define the lattice gas Hamiltonian in terms of the variables n_i and D_l, by noting that for an OGR system of length L, we want to maximize the lattice gas density which is given by $\sum_i n_i$, minimizing $\sum_l D_l(D_l - 1)$, where the latter sum is zero when the degeneracies D_l are zero or one as required for a Golomb ruler state. The OGR lattice-gas Hamiltonian is then,

$$H_{OGR} = -\mu \sum_{i=0}^{L} n_i + \gamma \sum_{l=1}^{L} D_l(D_l - 1) \tag{5.30}$$

This Hamiltonian may be written in terms of the variables n_i by using Eq. 5.28, which leads to a frustrated lattice Hamiltonian with long-range four-particle interactions. Provided the parameters μ and γ are positive, the first term in H_{OGR} maximizes the density of the lattice gas, while the second term minimizes the number of times an interparticle distance is repeated.

To find the scaling behavior of the optimal solutions to H_{OGR}, $L(m)$, we consider the partition function of OGR,

$$Z = \sum_{k=0}^{2^L-1} e^{-\beta H_k} \tag{5.31}$$

$$Z = \sum_{k=0}^{2^L-1} e^{-\beta\left(-\mu \sum_{i=0}^{L} n_i + \gamma \sum_{l=1}^{L} D_l(D_l-1)\right)} \tag{5.32}$$

$$Z = \sum_{k=0}^{2^L-1} e^{-\beta\left(-\mu m + \gamma \sum_{l=1}^{L} {D_l}^2 - \gamma \sum_{l=1}^{L} D_l\right)} \tag{5.33}$$

$$Z = \sum_{k=0}^{2^L-1} e^{\beta\mu \sum_{i=0}^{L} n_i - \beta\gamma(\sum_{l=1}^{L} D_l^2 - m(m-1)/2)} \tag{5.34}$$

where we used the identity Eq. 5.29. To reduce the D_l^2 term to linear form we introduce Gaussian integrals,

$$e^{D^2} = A \int e^{(-X^2+2XD)} dx \tag{5.35}$$

so that,

$$Z \quad = \quad A_G \sum_{k=0}^{2^L-1} e^{\beta\mu m + \beta\gamma m(m-1)/2)} \int ... \int \left(\prod_l dx_l\right) e^{-\sum_l x_l^2 + 2i\sqrt{\beta\gamma}\sum_l x_l D_l} \tag{5.36}$$

where A_G normalizes the Gaussian integrals. This remains intractable, but becomes tractable when we make the symmetric assumption $x_l = x$,

$$\int ... \int \left(\prod_l dx_l\right) e^{-\sum_l x_l^2 + 2i\sqrt{\beta\gamma}\sum_l x_l D_l} \quad = \quad \left(\int dx e^{-x^2 + 2i\sqrt{\beta\gamma}(x/L)\sum_l D_l}\right)^L \tag{5.37}$$

Then we use the identity Eq. 5.29 again to get,

$$\int ... \int \left(\prod_l dx_l \right) e^{-\sum_l x_l^2 + 2i\sqrt{\beta\gamma} \sum_l x_l D_l} = \left(\int dx e^{-x^2 + 2i\sqrt{\beta\gamma}(x/L)(m/2)(m-1)} \right)^L \quad (5.38)$$

We convert the integral to an exponent using the Gaussian integral used earlier to get,

$$\int ... \int \left(\prod_l dx_l \right) e^{-\sum_l x_l^2 + 2i\sqrt{\beta\gamma} \sum_l x_l D_l} = e^{-(\beta\gamma/L)[(m/2)(m-1)]^2 L}. \quad (5.39)$$

Thus,

$$Z = B_G \sum_{k=0}^{2^L - 1} e^{\beta\mu m + \frac{\beta\gamma}{2} m(m-1) - \frac{\beta\gamma}{4L}[m(m-1)]^2} \quad (5.40)$$

$$Z = B_G \times \sum_m \binom{L}{m} e^{\beta\mu m + \frac{\beta\gamma}{2} m(m-1) - \frac{\beta\gamma}{4L}[m(m-1)]^2} \quad (5.41)$$

where B_G is a constant. To find the scaling behavior of Optimal Golomb rulers we consider the strong interaction limit $\gamma \to \infty$ where the OGR constraints dominate, and solving yields the scaling law,

$$L(m) = m^2 - m \quad (5.42)$$

This is consistent with the known lower bound $L(m) > m^2 - 2m^{3/2} - m$ and with the Erdös conjecture $L(m) < m^2$ [47], as well as with large scale simulations (see Fig. 5.9).

5.7 Summary

In this work, a new connection between statistical physics and the combinatorial optimization problem of the optimal Golomb ruler is made. The statistical physics of the Golomb ruler problem is studied using the mean field approach and the phase diagram is obtained. It is

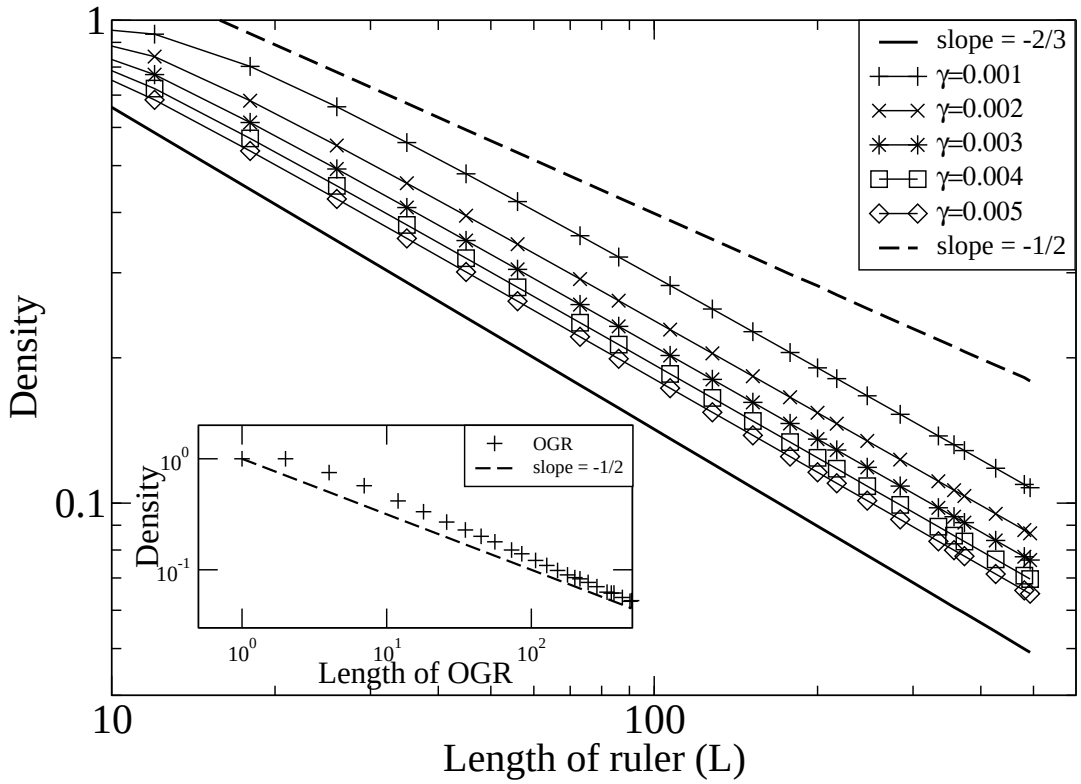

Figure 5.9: Comparison of numerical results (++++) for the length of optimal Golomb ruler with the best lower bound (solid line), and with the statistical physics scaling law (dotted line) that provides a useful upper bound on all best OGRs. The main figure is for exact OGR states, while the inset is for approximate OGR states of large size.

seen that even at a very low temperature it shows a first order phase transition at a finite non-zero value of the constraint parameter γ/μ. Analytic and exact calculations were done for the scaling of the density and free energy of the ruler and they were compared with those from the mean field. A new scaling law for the length of the OGR is derived, which is consistent with Erdös conjecture.

Chapter 6

Conclusion

In this work, efficient methods of reconstructing complex Euclidean networks in two and three dimensions, given only their unassigned Euclidean distance lists were presented. The unassigned problem is complicated due to the combinatorial explosion of ways that atoms may be assigned to the endpoints of each distance in the distance list, leading to interesting theoretical and algorithmic problems as elucidated.

It was found that there is enough information to uniquely reconstruct co-ordinates from distance lists that have no vector information in them and also found that this reconstruction is unique. Using the Tribond algorithm random point sets in 2 dimensions of size about one thousand were successfully reconstructed.

In 3 dimensions, the core finding step has a high computational cost and the algorithm was successfully able to do core finding for random point sets of size about ten in a small amount of time. If the core is assumed to be given, it was able to successfully do the buildup for random point sets for size about 100. The algorithm was also used to solve for the structures of various organic molecules and the results were compared with those from using the Liga algorithm.

While the Liga algorithm is successful in reconstructing structures which have a high symmetry (and a highly degenerate distance list), the Tribond algorithm is successful with random point sets which have a non-degenerate distance list. A hybrid algorithm should solve structures which fall between those having a high symmetry and a low symmetry.

Practical applications of the distance list method must overcome errors in the data including missing distances, shifted distances and errors in the multiplicity of peaks in degenerate cases. These issues are discussed in recent studies that attempt to reconstruct nanostructure from experimental PDF data [28, 29, 30]. The focus of this work was the broader theoretical and algorithmic issues related to the underlying inverse problem of finding structure from precise Euclidean distances.

A statistical physics approach to the combinatorial optimization problem of optimal Golomb ruler is also presented. The phase diagram is studied and scaling calculations for the density, free energy and phase boundary were done. An analytic calculation using a continuum field theory gives the scaling for the length of the OGR that is consistent with Erdös conjecture and also with the proposed optimal rulers of large lengths.

APPENDIX

```
1  // Tribond.h: The header file for all the Tribond 2D & 3D ↪
       ↩ functions.
2  //
3  #ifndef Tribond_h
4  #define Tribond_h
5
6  #include <iostream>
7  #include <cmath>
8  #include <cstdlib>
9  #include <cassert>
10 #include <vector>
11 #include <string>
12 #include <algorithm>
13 #include <iomanip>
14 using std::vector;
15 using std::string;
16 using std::cout;
17 using std::endl;
18 using std::setprecision;
19 using std::ios;
20 using std::setw;
21
22 //// Point Class ////
23 class Point
24 {
25   public:
26     long double x, y, z; // Coordinates
27     long double cost;
28     Point()
29     {
30       x = y = z = 0.0;
31       cost = -1;
32     }
33
34     friend bool compareCost( const Point& pt1, const Point& pt2 );
35 };
36
37
38 // Overloading "less than" operator so that points can be sorted.
39 bool operator<( const Point& pt1, const Point& pt2 );
40
41
42 // Overloading output operator for Point class.
43 std::ostream& operator<<( std::ostream& os, const Point& pt );
44
```

```
45
46 int print2structures( vector<Point>& stru1, vector<Point>& stru2 );
47 long double getAngle( Point pt1, Point pt2 );
48 Point getAxis( Point pt1, Point pt2 );
49
50
51 // Calculate the Euclidian distance between 2 points.
52 inline long double getDistance( const Point& pt1, const Point& pt2 )
53 {
54   // return sqrt( ( pt1.x - pt2.x )* ( pt1.x - pt2.x ) +
55   //              ( pt1.y - pt2.y )* ( pt1.y - pt2.y ) +
56   //              ( pt1.z - pt2.z )* ( pt1.z - pt2.z ) );
57   return sqrt( pow( pt1.x - pt2.x, 2.0L) +
58                pow( pt1.y - pt2.y, 2.0L) +
59                pow( pt1.z - pt2.z, 2.0L) );
60 }
61
62
63 //// Structure Class ////
64 class Structure
65 {
66   public:
67     // Relative tolerance to check if 2 distances are the close enough.
68     static long double toler;
69     static const int maxPoolSize;
70     vector<Point> atoms;
71     vector<Point> pool;
72     vector<long double> targetDL, currDL, freeDL;
73     vector<bool> usedDist;
74     long double cost;
75     int dim, targetSize, currSize;
76
77     Structure( int DIM, int N, string dlistFile )
78     {
79       targetSize = N;
80       dim = DIM;
81       atoms.resize( N );
82
83       int sizeDL = N* ( N - 1 )/ 2;
84       usedDist.resize( sizeDL, false );
85       cost = 0;
86
```

```
87          getDLfromFile( dlistFile );
88        }
89
90        Structure( int& N, string xyzFile )
91        {
92          getStruFromFile( xyzFile );
93
94          if( atoms.size() != N )
95          {
96            N = atoms.size();
97            cout << "Updating structure size to" << atoms.size() ↪
                 ↩ << endl;
98          }
99          targetSize = N;
100         dim = 3;
101         // atoms.resize( N );
102
103         int sizeDL = N* ( N - 1 )/ 2;
104         usedDist.resize( sizeDL, false );
105         cost = 0;
106
107       }
108
109       Structure( int DIM, int N, vector<long double> inputDL )
110       {
111         currSize = 0;
112         targetSize = N;
113         dim = DIM;
114         int sizeDL = N* ( N - 1 )/ 2;
115         usedDist.resize( sizeDL, false );
116
117         targetDL = inputDL;
118       }
119
120       int updateCurrDL()
121       {
122         currDL.clear();
123         for( int i = 0; i < atoms.size(); ++i )
124         {
125           for( int j = i + 1; j < atoms.size(); ++j )
126           {
127             currDL.push_back( getDistance( atoms[ i ], atoms[ j ] ↪
                  ↩ ) );
128           }
129         }
```

```
130         sort( currDL.begin(), currDL.end() );
131         // cout << "Size of current DL: " << currDL.size() << endl;
132         return 0;
133       }
134
135       int print()
136       {
137         int Precision = 8;
138         int Width = 12;
139         int Width2 = 6;
140
141         cout.precision( Precision );
142         cout.setf(ios::fixed,ios::floatfield);
143
144         for ( int i = 0; i < atoms.size(); ++i )
145         {
146           cout << setw( Width ) << atoms[ i ].x << '\t'
147                << setw( Width ) << atoms[ i ].y << '\t'
148                << setw( Width2 ) << atoms[ i ].z << '\t'
149                << atoms[ i ].cost << endl;
150         }
151         // for ( int i = 0; i < atoms.size(); ++i )
152         // {
153         //   cout << i << '\t' << setprecision( 10 ) << atoms[ i ↪
               ↩ ].x << '\t'
154         //        << setprecision( 10 ) << atoms[ i ].y << '\t' ↪
               ↩ << setprecision( 10 ) << atoms[ i ].z << endl;
155         // }
156         //
157         cout << "****" << endl;
158         return 0;
159       }
160
161       bool findCore();
162       bool findCore3D();
163       bool findCore3D( int baseIdx );
164       bool doBuildup();
165       bool doBuildup3D( vector<int> idxArr );
166       // bool doBuildup3D( int newBase );
167       bool doBuildup3Dv2( int basePt1, int basePt2, int basePt3 );
168       bool doBuildup2( int newBase );
169       bool doBuildup3( int newBase );
170       bool reconstruct();
171       bool reconstruct2();
172       bool reconstruct3( int baseIdx );
```

```
173     long double feasibleTetra( int baseIdx );
174
175     long double distListError();
176     long double distListError( vector<long double> dlist );
177     long double distListError( vector<long double> dlist1,
178                                vector<long double> dlist2 );
179     long double checkMoreBridges( int numChecks, Point testPt );
180
181     bool home();
182     bool home3D( int distIdx );
183     bool reflect( string axis );
184     bool rotate( long double angle );
185     bool rotate( long double angle, long double axisX,
186                  long double axisY, long double axisZ );
187
188     bool translate( long double distX, long double distY, long
        ↩ double distZ );
189
190     bool testCore( vector<int> idxArr, int idxM, int idxN );
191     int getDLfromFile( string fileName );
192     int getStruFromFile( string fileName );
193     int printDLtoFile( string fileName="" );
194
195     int reduceDLprecision( int precision );
196     vector<Point> getCore( int coreSize );
197     int updateUsedDists();
198     int updateFreeDL();
199     long double getPtCost( Point pt );
200     long double getPtsCost( Point pt1, Point pt2 );
201
202     bool updatePool( Point pt );
203     int insertPoint( Point pt );
204     bool growStru();
205     int printPool();
206     int getPools();
207     int getPools2();
208     bool findCoreMPI( int windowStart );
209 };
210
211
212 inline long double compareStru( Structure testStru, Structure
    ↩ targetStru )
213 {
214   testStru.updateCurrDL();
215   long double overlapError = 0;
```

```
216     sort( testStru.atoms.begin(), testStru.atoms.end() );
217
218     for( int i = 0; i < testStru.targetSize; ++i )
219     {
220       overlapError += pow( getDistance( testStru.atoms[ i ],
221                                         targetStru.atoms[ i ] ) , ↪
                                            ↩ 2.0 );
222     }
223
224     cout << "Overlap Error: " << overlapError << endl;
225     cout << "Distance Error: " << testStru.distListError() << endl;
226     return 0;
227 }
228
229
230
231 #endif
```

```
1  // Tribond.cpp: The implementation file for of all the 2D & 3D ↪
       ↩ functions.
2  //
3  #include "Tribond.h"
4  #include <algorithm>
5  #include <iostream>
6  #include <fstream>
7  #include <iomanip>
8  #include <sstream>
9  #include <cassert>
10 using namespace std;
11
12 long double Structure::toler = 1e-12L;
13 const int Structure::maxPoolSize = 20;
14
15
16 bool operator<( const Point& pt1, const Point& pt2 )
17 {
18   // Overloading the "less than" operator for the Point class. ↪
       ↩ Useful when
19   // sorting points in the structure so that their ordering is ↪
       ↩ unique.
20
21   Point zero;
22   return getDistance( pt1, zero ) < getDistance( pt2, zero );
23 }
24
```

```
25
26 std::ostream& operator<<( std::ostream& os, const Point& pt )
27 {
28   // Overloading output operator for Point class.
29
30   int outputPrecision = 8;
31   int colWidth = 12;
32
33   os.precision( outputPrecision );
34   os.setf(ios::fixed,ios::floatfield);
35
36   os << setw( colWidth ) << pt.x
37      << setw( colWidth ) << pt.y
38      << setw( colWidth ) << pt.z;
39   return os;
40 }
41
42
43 void placeApex( Point& apexPt, int idxA, int idxB, int idxC,
44                 vector<long double> dlist )
45 {
46   // Place the top point of the triangle by solving the loci ↪
       ↩ equations.
47   // For skinny triangles, the y-coordinate may turn out to be ↪
       ↩ negative.
48   // In those cases, I am setting it to zero.
49
50   apexPt.x = dlist[ idxA ]/ 2 - ( ( dlist[ idxC ] - dlist[ idxB ↪
       ↩ ] )*
51                                 ( dlist[ idxC ] + dlist[ idxB ] )/
52                                 ( 2* dlist[ idxA ] ) );
53
54   apexPt.y = sqrt( ( dlist[ idxC ] + apexPt.x - dlist[ idxA ] )*
55                    ( dlist[ idxC ] - apexPt.x + dlist[ idxA ] ) );
56
57   apexPt.z = 0;
58
59   if( apexPt.y != apexPt.y )
60   {
61     apexPt.y = 0;
62   }
63
64   return;
65 }
66
```

```
67
68 void placeTop( Point basePt1, Point basePt2, Point basePt3, ↪
     ↩ Point& apexPt,
69                 vector<int> idxArr, vector<long double> dlist )
70 {
71     /* Place Apex in space.
72      *
73      * Relating bonds to points:
74      * a <=> p1-p2       f <=> p3-p4
75      * b <=> p1-p3       e <=> p2-p4
76      * c <=> p2-p3       d <=> p1-p4
77      *
78      * p1 is placed at the origin.
79      * p2 is placed at (dlist[a], 0, 0).
80      * p3 is defined to have positive y-value.
81      * p4 is defined to have positive z-value.
82      */
83
84     long double d12, d13, d14, d23, d24, d34;
85     d12 = dlist[ idxArr[ 0 ] ]; d34 = dlist[ idxArr[ 5 ] ];
86     d13 = dlist[ idxArr[ 1 ] ]; d24 = dlist[ idxArr[ 4 ] ];
87     d23 = dlist[ idxArr[ 2 ] ]; d14 = dlist[ idxArr[ 3 ] ];
88
89     // placement of apex by solving simple loci equations
90     // faster than evaluating trig expressions
91     apexPt.x = ( d14* d14 - d24* d24 + d12* d12 )/ ( 2.0L* d12 );
92     apexPt.y = ( ( d14* d14 - d34* d34 + basePt3.x* basePt3.x +
93                    basePt3.y* basePt3.y )/ ( 2.0L* basePt3.y ) ↪
                        ↩ ) -
94                ( basePt3.x/ basePt3.y )* apexPt.x;
95
96     if( ( d14* d14 - apexPt.x* apexPt.x - apexPt.y* apexPt.y ) ↪
        ↩ < 0.0L )
97     {
98       // cout << "img dist!" << endl;
99       // apexPt.z = 0;
100      // cout << "pars: " << idxArr[ 0 ] << ',' << idxArr[ 1 ] ↪
           ↩ << ','
101      //                  << idxArr[ 2 ] << ',' << idxArr[ 3 ] ↪
           ↩ << ','
102      //                  << idxArr[ 4 ] << ',' << idxArr[ 5 ] ↪
           ↩ << endl;
103      // cout << "pts: " << endl;
104      // cout << basePt1 << endl;
105      // cout << basePt2 << endl;
```

```
106         // cout << basePt3 << endl;
107         // cout << apexPt << endl;
108         // getchar();
109         apexPt.z = 0; //sqrt( d14* d14 - apexPt.x* apexPt.x -
                ↩ apexPt.y* apexPt.y );
110       }
111       else
112       {
113         apexPt.z = sqrt( d14* d14 - apexPt.x* apexPt.x -
                ↩ apexPt.y* apexPt.y );
114       }
115   }
116
117
118   int closestDist( const long double& value, const vector<long
        ↩ double>& dlist )
119   {
120     // Function to find the index of the distance in the given list
121     // which is closest to the given value.
122
123     vector<long double>::const_iterator const it
124       = lower_bound( dlist.begin(), dlist.end(), value );
125
126     int bestIdx = distance( dlist.begin(), it );
127
128     if ( bestIdx == dlist.size() )
129     {
130       bestIdx -= 1;
131     }
132     else if ( bestIdx > 0 and
133     ( fabs( dlist[ bestIdx - 1 ] - value ) < fabs( dlist[ bestIdx
          ↩ ] - value ) ) )
134     {
135       bestIdx -= 1;
136     }
137
138     // cout << "value, bestIdx, bestDist: " << setprecision( 11 )
          ↩ << value << ", " << bestIdx << ", "
139     //      << setprecision( 11 ) << dlist[ bestIdx ] << '\t';
140     // cout << "( " << dlist[ bestIdx - 1 ] << ", " << dlist[
          ↩ bestIdx + 1 ] << " )"<< endl;
141     return bestIdx;
142   }
143
144   bool Structure::findCore()
```

```
145 {
146   // Find a core made of 4 points by iterating over all triangle
147   // combinations. Also, the function to do the buildup is ↪
        ↩ called after
148   // we find a core because it is more convenient this way.
149
150   int idxA = 0, idxB, idxC, idxD, idxE, idxF; // indices for ↪
        ↩ the bonds
151   int idxM, idxN; // indices for the orientation of the triangle
152   vector<int> idxArr( 6, -1 );
153   int inc = 6; // width of the bond window
154   int winStart = 0, winStop = inc; // indices for the window
155   vector<long double> dlist = targetDL;
156
157   long double bridgeDist = 0.0;
158   int bridgeIdx = 0;
159   int bridgeCount = 0; // count the number of bridge bond checks
160   long double fracError = 1e6;
161
162   Point basePt1, basePt2,
163         apexPt1, apexPt2;
164   basePt1.x = 0; basePt1.y = 0;
165   basePt2.x = dlist[ idxA ]; basePt2.y = 0;
166   cout << "basePt1: " << basePt1.x << " " << basePt1.y << endl;
167   cout << "basePt2: " << basePt2.x << " " << basePt2.y << endl;
168   cout << "Bond window: ";
169
170   while( true )
171   {
172     cerr << " -> " << winStop;
173
174     for( idxB = idxA + 1; idxB < winStop; ++idxB )
175     {
176       for( idxC = idxB + 1; idxC < winStop; ++idxC )
177       {
178         if( dlist[ idxA ] + dlist[ idxB ] + toler < dlist[ idxC ↪
              ↩ ] )
179         {
180           break;
181         }
182         placeApex( apexPt1, idxA, idxB, idxC, dlist );
183
184         for( idxD = idxA + 1; idxD < winStop; ++idxD )
185         {
186           if ( idxD == idxB or idxD == idxC )
```

```
187             {
188               continue;
189             }
190             for( idxE = idxD + 1; idxE < winStop; ++idxE )
191             {
192               if( idxB < winStart and idxC < winStart and
193                   idxD < winStart and idxE < winStart )
194               {
195                 continue;
196               }
197
198               if( ( idxD < idxB and idxE < idxC ) or
199                   ( idxE > idxC and idxD < idxB ) )
200               {
201                 continue;
202               }
203
204               if( idxE == idxB or idxE == idxC )
205               {
206                 continue;
207               }
208
209               if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ ↪
                    ↩ idxE ] )
210               {
211                 break;
212               }
213
214               placeApex( apexPt2, idxA, idxD, idxE, dlist );
215
216               for( idxM = 0; idxM < 2; ++idxM )
217               {
218                 // cout << "idxM: " << idxM << endl;
219                 if( idxM == 1 )
220                 {
221                   apexPt2.x = dlist[ idxA ] - apexPt2.x;
222                 }
223
224                 for( idxN = 0; idxN < 2; ++idxN )
225                 {
226                   // cout << "idxN: " << idxN << endl;
227                   if( idxN == 1 )
228                   {
229                     apexPt2.y = - apexPt2.y;
230                   }
```

```
231
232                 bridgeDist = getDistance( apexPt1, apexPt2 );
233                 bridgeCount += 1; // count number of bridge checks
234
235                 bridgeIdx = closestDist( bridgeDist, dlist );
236                 if( bridgeIdx == idxA or bridgeIdx == idxB or
237                     bridgeIdx == idxC or bridgeIdx == idxD or
238                     bridgeIdx == idxE )
239                 {
240                   // Make sure that the bridge bond is not the ↪
                        ↩ same as any of
241                   // the distances in use.
242                   continue;
243                 }
244
245                 fracError = fabs( dlist[ bridgeIdx ] - ↪
                      ↩ bridgeDist )/ bridgeDist;
246                 // cout << "fracError: " << fracError << endl;
247
248                 if( fabs( apexPt2.y ) < 0.5 )
249                 {
250                   // Skinny triangles have been found to have a ↪
                        ↩ large error,
251                   // hence reducing their error "by hand" so ↪
                        ↩ that we don't
252                   // miss out on them.
253                   fracError /= 1000;
254                 }
255
256                 if( fracError < toler )
257                 {
258                   idxArr[ 0 ] = idxA; idxArr[ 1 ] = idxB;
259                   idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxD;
260                   idxArr[ 4 ] = idxE; idxArr[ 5 ] = bridgeIdx;
261
262                   cout << endl;
263                   if( testCore( idxArr, idxM, idxN ) )
264                   {
265                     atoms.push_back( basePt1 );
266                     atoms.push_back( basePt2 );
267                     atoms.push_back( apexPt1 );
268                     atoms.push_back( apexPt2 );
269
270                     for( int i = 0; i < atoms.size(); ++i )
271                     {
```

```
272                     cout << "Point: " << i + 1 << '\t' << atoms[ i ] << endl;
273                   }
274
275                   usedDist[ idxA ] = usedDist[ idxB ] = true;
276                   usedDist[ idxC ] = usedDist[ idxD ] = true;
277                   usedDist[ idxE ] = usedDist[ bridgeIdx ] = true;
278
279                   updateCurrDL();
280                   // return true;
281
282                   // Attempt buildup to get the remaining points.
283                   doBuildup();
284
285                   if( atoms.size() >= min( 8, targetSize ) )
286                   {
287                     // If buildup was able to add 4 more points then with
288                     // high probability, we have the right structure.
289                     return true;
290                   }
291                   else
292                   {
293                     // If buildup could not even 4 points then with a very
294                     // high probability, we have the wrong structure. Start
295                     // over and find the next core.
296                     cout << "No buildup, bad core. "
297                             "Finding the next core... " << endl;
298                     atoms.clear();
299                     updateCurrDL();
300                     cout << "Bond window: ";
301                   }
302                 }
303               }
304
305             } // n loop
306           } // m loop
307         } // idxE loop
308       } // idxD loop
```

```
309         } // idxC loop
310       } // idxB loop
311 
312       // When idxB hits the window edge increment it.
313       if ( idxB == winStop )
314       {
315         winStart = winStop;
316         winStop += inc;
317 
318         if( winStart == dlist.size() )
319         {
320           cout << endl;
321           break;
322         }
323 
324         if( winStop > dlist.size() )
325         {
326           winStop = dlist.size();
327         }
328       }
329 
330     } // while( true ) loop
331 
332     return false;
333 }
334 
335 
336 bool Structure::doBuildup()
337 {
338     // Starting with a core of size 4, find the remaining points.
339     // Partial update of the new distances created used while ↪
        ↩ adding a point.
340 
341     bool successFlag = false;
342     int idxA = 0, idxB, idxC, idxD, idxE, idxF; // indices for ↪
        ↩ the bonds
343     int idxM, idxN;
344 
345     long double bridgeDist = 0.0;
346     int bridgeIdx = 0;
347     int bridgeCount = 0; // count the number of triangles
348     vector<long double> dlist = targetDL;
349     long double fracError = 1e6, fracError2 = 1e6;
350 
351     assert( ("In buildup, core present?", atoms.size() > 0 ) );
```

```
352     Point basePt1 = atoms[ 0 ], basePt2 = atoms[ 1 ],
353           apexPt = atoms[ 2 ], testPt;
354
355     for( idxD = 1; idxD < dlist.size(); ++idxD )
356     {
357       if ( usedDist[ idxD ] )
358       {
359         continue;
360       }
361
362       for( idxE = idxD + 1; idxE < dlist.size(); ++idxE )
363       {
364         if ( usedDist[ idxE ] )
365         {
366           continue;
367         }
368
369         if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ idxE ] )
370         {
371           break;
372         }
373
374         placeApex( testPt, idxA, idxD, idxE, dlist );
375
376         for( idxM = 0; idxM < 2; ++idxM )
377         {
378           if( idxM == 1 )
379           {
380             testPt.x = dlist[ idxA ] - testPt.x;
381           }
382
383           for( idxN = 0; idxN < 2; ++idxN )
384           {
385             if( idxN == 1 )
386             {
387               testPt.y = - testPt.y;
388             }
389
390             bridgeCount += 1; // count number of bridge checks
391             bridgeDist = getDistance( apexPt, testPt );
392             if( bridgeDist < dlist[ 0 ] )
393             {
394               // Make sure the test point is not too close to any ↪
                    ↩ point.
395               continue;
```

```
396             }
397
398             bridgeIdx = closestDist( bridgeDist, dlist );
399             if( bridgeIdx == idxD or bridgeIdx == idxE or
400                 usedDist[ bridgeIdx ] )
401             {
402               // Make sure the bridge bond is not the same as a used distance.
403               continue;
404             }
405
406             fracError = fabs( dlist[ bridgeIdx ] - bridgeDist )/ bridgeDist;
407
408             if( fabs( testPt.y ) < 0.5 )
409             {
410               // Found skinny triangles to have a high error. Hence, reducing
411               // the error "by hand" so that we don't miss out on them.
412               fracError /= 1000;
413             }
414
415             if ( fracError >  toler )
416             {
417               continue;
418             }
419             else
420             {
421               fracError2 = checkMoreBridges( 10, testPt );
422               if( fracError2 > sqrt( toler ) )
423               {
424                 continue;
425               }
426               else
427               {
428                 usedDist[ idxD ] = true;
429                 usedDist[ idxE ] = true;
430                 usedDist[ bridgeIdx ] = true;
431
432                 atoms.push_back( testPt );
433                 cout << "Point: " << atoms.size() << '\t'
434                      << testPt << endl;
435               }
436             }
```

```
437
438           } // n loop
439         } // m loop
440       } // idxE loop
441     } // idxD loop
442
443     currSize = atoms.size();
444     updateCurrDL();
445
446     if( atoms.size() == targetSize )
447     {
448       successFlag = true;
449     }
450
451     return successFlag;
452   }
453
454
455   long double Structure::distListError()
456   {
457     // Calculate the error based on the closeness of the current ↪
          ↩ and the target
458     // distance lists.
459
460     assert( currDL.size() == targetDL.size() );
461     long double error = 0;
462
463     for( int i = 0; i < currDL.size(); ++i )
464     {
465       error += pow( currDL[ i ] - targetDL[ i ] , 2.0 );
466     }
467
468     return sqrt( error/ currDL.size() );
469   }
470
471
472   long double Structure::distListError( vector<long double> dlist1 ,
473                                         vector<long double> ↪
                                              ↩ dlist2 )
474   {
475     // Calculate the error based on the closeness of 2 distance ↪
          ↩ lists.
476
477     if( dlist1.size() != dlist2.size() )
478     {
```

```
479       return 1e6;
480     }
481     sort( dlist1.begin(), dlist1.end() );
482     sort( dlist2.begin(), dlist2.end() );
483     long double error = 0;
484
485     for( int i = 0; i < dlist1.size(); ++i )
486     {
487       error += pow( dlist1[ i ] - dlist2[ i ] , 2.0 );
488     }
489
490     return sqrt( error/ dlist1.size() );
491   }
492
493
494   long double Structure::distListError( vector<long double> dlist )
495   {
496     // Function to calculate the error based on how close the 2 ↪
          ↩ dlists are.
497
498     long double dError = 0;
499     long double totalError = 0;
500     size_t bIdx = 0;
501
502     vector<int> countB( targetDL.size(), 0 );
503     int repeat = 0;
504     updateUsedDists();
505     vector<long double> freeDL;
506
507     for( int i = 0; i < targetDL.size(); ++i )
508     {
509       if( usedDist[ i ] == false )
510       {
511         freeDL.push_back( targetDL[ i ] );
512       }
513     }
514
515     for( int i = 0; i < dlist.size(); i++ )
516     {
517       // bIdx = closestDist( dlist[i], targetDL );
518       // dError = dlist[i] - targetDL[ bIdx ];
519       bIdx = closestDist( dlist[i], freeDL );
520       dError = dlist[i] - freeDL[ bIdx ];
521       totalError += dError* dError;
522
```

```
523       countB[ bIdx ] += 1;
524       if ( countB[ bIdx ] >= 2 )
525       {
526         ++repeat;
527       }
528       // FIXME: Commenting out case when there is a large error.
529       if ( dError > 0.5 )
530       {
531         // return 1.0;
532       }
533     }
534
535     dError += 0.01* repeat;
536     // cout << "pt cost: " << setprecision( 8 ) << sqrt( ↪
        ↩ totalError/ dlist.size() ) << endl;
537     // getchar();
538     return sqrt( totalError/ dlist.size() );
539 }
540
541
542 long double Structure::checkMoreBridges( int numChecks, Point ↪
      ↩ testPt )
543 {
544     // Check more bridge bonds for the testPt to make sure that ↪
        ↩ it is
545     // indeed part of the actual structure.
546
547     long double fracError = 0;
548     long double bridgeDist = 0;
549     int bridgeIdx = 0;
550     vector<long double> dlist = targetDL;
551
552     for( int i = 0; i < numChecks; ++i )
553     {
554       if( i >= atoms.size() ) break;
555       bridgeDist = getDistance( testPt, atoms[ i ] );
556       if( bridgeDist < dlist[ 0 ] )
557       {
558         fracError += 0.1;
559         continue;
560       }
561
562       bridgeIdx = closestDist( bridgeDist, dlist );
563       fracError += fabs( dlist[ bridgeIdx ] - bridgeDist )/ ↪
          ↩ bridgeDist;
```

```
564     }
565
566     return fracError/numChecks;
567 }
568
569
570 bool Structure::reconstruct()
571 {
572     // Reconstruct the structure by first finding the core and ↪
        ↩ then doing
573     // buildup(if needed).
574
575     // Need to find the core first.
576     if( atoms.size() == 0 )
577     {
578       if( dim == 2 )
579       {
580         bool findCoreFlag = findCore();
581         cout << "Core found? " << boolalpha << findCoreFlag << endl;
582       }
583       else if( dim == 3 )
584       {
585         // cout << "findCore3D(2)" << endl;
586         findCore3D();
587       }
588     }
589
590     // Core is already there, just do the buildup.
591     if( dim == 3 and atoms.size() >= 4 )
592     {
593       vector<int> idxArr( 6, 0 );
594       idxArr[ 0 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
          ↩ 1 ] ),
595                                  targetDL );
596       idxArr[ 1 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
          ↩ 2 ] ),
597                                  targetDL );
598       idxArr[ 2 ] = closestDist( getDistance( atoms[ 1 ], atoms[ ↪
          ↩ 2 ] ),
599                                  targetDL );
600       idxArr[ 3 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
          ↩ 3 ] ),
601                                  targetDL );
602       idxArr[ 4 ] = closestDist( getDistance( atoms[ 1 ], atoms[ ↪
          ↩ 3 ] ),
```

```
603                                     targetDL );
604     idxArr[ 5 ] = closestDist( getDistance( atoms[ 2 ], atoms[ 3 ] ),
605                                     targetDL );
606     cout << "idxArr: " << idxArr[ 0 ] << ',' << idxArr[ 1 ] << ','
607                       << idxArr[ 2 ] << ',' << idxArr[ 3 ] << ','
608                       << idxArr[ 4 ] << ',' << idxArr[ 5 ] << endl;
609     print();
610
611     int idxG, idxH, idxI, idxJ;
612
613     idxG = closestDist( getDistance( atoms[ 0 ], atoms[ 4 ] ), targetDL );
614     idxH = closestDist( getDistance( atoms[ 1 ], atoms[ 4 ] ), targetDL );
615     idxI = closestDist( getDistance( atoms[ 2 ], atoms[ 4 ] ), targetDL );
616     idxJ = closestDist( getDistance( atoms[ 3 ], atoms[ 4 ] ), targetDL );
617     cout << "idx: " << idxG << ',' << idxH << ','
618          << idxI << ',' << idxJ << endl;
619     usedDist[ idxG ] = usedDist[ idxH ] = usedDist[ idxI ]
620                      = usedDist[ idxJ ] = true;
621
622     doBuildup3D( idxArr );
623
624     // attempt buildup again if short of a few points
625     int attempts = 0;
626     long double oldToler = toler;
627     while( attempts < 5 and atoms.size() < targetSize )
628     {
629       toler *= 10;
630       doBuildup3D( idxArr );
631       ++attempts;
632       cout << "attempt: " << attempts << endl;
633     }
634     toler = oldToler;
635   }
636
637   if( atoms.size() == targetSize )
638   {
639     return true;
640   }
641
```

```
642   bool successFlag = doBuildup();
643   if( successFlag )
644   {
645     cout << "Finished reconstruction" << endl;
646   }
647   return successFlag;
648 }
649
650
651 bool Structure::reflect( string axis )
652 {
653   // Reflect the structure about the X or the Y axis, used by
        ↩ the overlap
654   // function.
655
656   if( axis == "X" )
657   {
658     for( int i = 0; i < atoms.size(); ++i )
659     {
660       atoms[ i ].y = - atoms[ i ].y;
661     }
662   }
663   else if( axis == "Y" )
664   {
665     for( int i = 0; i < atoms.size(); ++i )
666     {
667       atoms[ i ].x = - atoms[ i ].x;
668     }
669   }
670   else if( axis == "Z" )
671   {
672     for( int i = 0; i < atoms.size(); ++i )
673     {
674       atoms[ i ].z = - atoms[ i ].z;
675     }
676   }
677
678   return true;
679 }
680
681
682 bool Structure::home()
683 {
684   // Orient the structure in a unique manner so that it becomes
        ↩ easy to check
```

```
685     // if two or more structure are identical to one another or not.
686
687     long double minDist = 1e6;
688     int idx1 = -1, idx2 = -1;
689     long double dist = 1e6;
690
691     // Find the smallest bond in the structure
692     for( int i = 0; i < atoms.size(); ++i )
693     {
694       for( int j = i + 1; j < atoms.size(); ++j )
695       {
696         dist = getDistance( atoms[ i ], atoms[ j ] );
697         if( dist < minDist )
698         {
699           minDist = dist;
700           idx1 = i;
701           idx2 = j;
702         }
703       }
704     }
705     // cout << "idx1, idx2: " << idx1 << '\t' << idx2 << endl;
706
707     // Locate the apex point of the base triangle
708     long double minDist1 = 1e6, minDist2 = 1e6;
709     long double dist1, dist2;
710     int minIdx1, minIdx2;
711
712     for( int i = 0; i < atoms.size(); ++i )
713     {
714       if( i == idx1 or i == idx2 ) continue;
715
716       dist1 = getDistance( atoms[ i ], atoms[ idx1 ] );
717       if( dist1 < minDist1 )
718       {
719         minDist1 = dist1;
720         minIdx1 = i;
721       }
722
723       dist2 = getDistance( atoms[ i ], atoms[ idx2 ] );
724       if( dist2 < minDist2 )
725       {
726         minDist2 = dist2;
727         minIdx2 = i;
728       }
729     }
```

```
730
731     int idx3 = minIdx1;
732     if( minDist1 > minDist2 )
733     {
734       idx3 = minIdx2;
735       swap( idx1, idx2 );
736     }
737
738     // Correctly place the base triangle
739     translate( -atoms[ idx1 ].x, -atoms[ idx1 ].y, -atoms[ idx1 ].z );
740     // find angle and rotate
741     long double angle = atan2( atoms[ idx2 ].y, atoms[ idx2 ].x );
742     rotate( angle );
743
744     if( atoms[ idx3 ].y < 0 )
745     {
746       reflect( "X" );
747     }
748
749     // Sort the points so that there is some unique order
750     sort( atoms.begin(), atoms.end() );
751
752     return true;
753 }
754
755
756 bool Structure::testCore( vector<int> idxArr, int idxM, int idxN )
757 {
758     // Check to make sure that the core is correct by using all possible bonds
759     // as the base and checking if the resulting bridge bonds are part of the
760     // target distance list.
761
762     Point basePt1, basePt2,
763           apexPt1, apexPt2;
764     basePt1.x = 0; basePt1.y = 0;
765
766     vector<long double> dlist = targetDL;
767     long double bridgeDist, fracError;
768     int idxA, idxB, idxC, idxD, idxE, givenBridgeIdx;
769
770     int bondA = idxArr[ 0 ], bondB = idxArr[ 1 ], bondC = idxArr[
```

```
      ↩ 2 ],
771     bondBP = idxArr[ 3 ], bondCP = idxArr[ 4 ], bondBridge = ↪
          ↩ idxArr[ 5 ];
772
773   if( idxM == 1 )
774   {
775     swap( bondB, bondC );
776   }
777
778   vector<vector<int> > idxCombin( 6, idxArr );
779   idxCombin[ 0 ][ 0 ] = bondA; idxCombin[ 0 ][ 1 ] = bondB;
780   idxCombin[ 0 ][ 2 ] = bondC; idxCombin[ 0 ][ 3 ] = bondBP;
781   idxCombin[ 0 ][ 4 ] = bondCP; idxCombin[ 0 ][ 5 ] = bondBridge;
782
783   idxCombin[ 1 ][ 0 ] = bondB; idxCombin[ 1 ][ 1 ] = bondC;
784   idxCombin[ 1 ][ 2 ] = bondA; idxCombin[ 1 ][ 3 ] = bondBridge;
785   idxCombin[ 1 ][ 4 ] = bondBP; idxCombin[ 1 ][ 5 ] = bondCP;
786
787   idxCombin[ 2 ][ 0 ] = bondC; idxCombin[ 2 ][ 1 ] = bondB;
788   idxCombin[ 2 ][ 2 ] = bondA; idxCombin[ 2 ][ 3 ] = bondBridge;
789   idxCombin[ 2 ][ 4 ] = bondCP; idxCombin[ 2 ][ 5 ] = bondBP;
790
791   idxCombin[ 3 ][ 0 ] = bondBP; idxCombin[ 3 ][ 1 ] = bondB;
792   idxCombin[ 3 ][ 2 ] = bondBridge; idxCombin[ 3 ][ 3 ] = bondA;
793   idxCombin[ 3 ][ 4 ] = bondCP; idxCombin[ 3 ][ 5 ] = bondC;
794
795   idxCombin[ 4 ][ 0 ] = bondCP; idxCombin[ 4 ][ 1 ] = bondBP;
796   idxCombin[ 4 ][ 2 ] = bondA; idxCombin[ 4 ][ 3 ] = bondBridge;
797   idxCombin[ 4 ][ 4 ] = bondC; idxCombin[ 4 ][ 5 ] = bondB;
798
799   idxCombin[ 5 ][ 0 ] = bondBridge; idxCombin[ 5 ][ 1 ] = bondBP;
800   idxCombin[ 5 ][ 2 ] = bondB; idxCombin[ 5 ][ 3 ] = bondCP;
801   idxCombin[ 5 ][ 4 ] = bondC; idxCombin[ 5 ][ 5 ] = bondA;
802
803
804   bool success = true, localSuccess = false;
805   for( int i = 0; i < idxCombin.size(); ++i )
806   {
807     idxA = idxCombin[ i ][ 0 ]; idxB = idxCombin[ i ][ 1 ];
808     idxC = idxCombin[ i ][ 2 ]; idxD = idxCombin[ i ][ 3 ];
809     idxE = idxCombin[ i ][ 4 ]; givenBridgeIdx = idxCombin[ i ↪
          ↩ ][ 5 ];
810     basePt2.x = dlist[ idxA ]; basePt2.y = 0;
811
812     placeApex( apexPt1, idxA, idxB, idxC, dlist );
```

```
813       placeApex( apexPt2, idxA, idxD, idxE, dlist );
814
815       for( int idxY = 0; idxY < 2; ++idxY )
816       {
817         if( idxY == 1 )
818         {
819           apexPt2.y = - apexPt2.y;
820         }
821
822         bridgeDist = getDistance( apexPt1, apexPt2 );
823         fracError = fabs( dlist[ givenBridgeIdx ] - bridgeDist )/
824                           dlist[ givenBridgeIdx ];
825
826         if( fabs( apexPt2.y ) < 0.5 )
827         {
828           fracError /= 1000;
829         }
830
831         if( fracError < toler )
832         {
833           // cout << "fracError: " << fracError << endl;
834           localSuccess = true;
835           break;
836         }
837       }
838
839       if( localSuccess == false )
840       {
841         success = false;
842       }
843     }
844
845     cout << "Test core. Good? " << boolalpha << success << endl;
846     return success;
847   }
848
849
850   int Structure::getDLfromFile( string dlistFile )
851   {
852     // Read file to get the list of distances for the ↪
          ↩ reconstruction.
853     // If filename is "rand2", then generate a random point set.
854
855     long double inputDist;
856     assert( ( "Target distance list must be empty", ↪
```

```
      ↩ targetDL.size() == 0 ) );
857   ifstream inputFile;
858
859   if( dlistFile == "rand2" or dlistFile == "rand3" )
860   {
861     int N = targetSize;
862     for ( int i = 0; i < N; ++i )
863     {
864       atoms[ i ].x = N* float( rand() )/ RAND_MAX;
865       atoms[ i ].y = N* float( rand() )/ RAND_MAX;
866
867       if( dlistFile == "rand3" )
868       {
869         atoms[ i ].z = N* float( rand() )/ RAND_MAX;
870       }
871       atoms[ i ].cost = 0;
872     }
873
874     currSize = N;
875     updateCurrDL();
876     targetDL = currDL;
877   }
878   else
879   {
880     inputFile.open( dlistFile.data() );
881     assert( ("Trouble opening distance list file", inputFile ) );
882
883     while( inputFile >> inputDist )
884     {
885       targetDL.push_back( inputDist );
886     }
887     inputFile.close();
888
889     sort( targetDL.begin(), targetDL.end() );
890   }
891   return 0;
892 }
893
894
895 int Structure::printDLtoFile( string fileName )
896 {
897   // Save the list of distances to the given file.
898
899   ofstream outFile( fileName.data() );
900   outFile.precision( 20 );
```

```
901
902     cout << "currDL size: " << currDL.size() << endl;
903     if( outFile.is_open() )
904     {
905       for( int i = 0; i < currDL.size(); ++i )
906       {
907         outFile << currDL[i] << endl;
908       }
909       outFile.close();
910     }
911     else
912     {
913       cout << "unable to open file" << endl;
914     }
915
916     return 0;
917 }
918
919
920 int Structure::getStruFromFile( string fileName )
921 {
922     // Read the structure from the given file so that it can used
        ↩ as a core.
923     assert( ( "List of atoms should be empty", atoms.size() == 0
        ↩ ) );
924
925     ifstream inputFile;
926     Point inputPt;
927     inputPt.cost = 0;
928
929     inputFile.open( fileName.data() );
930
931     while( inputFile >> inputPt.x
932                      >> inputPt.y
933                      >> inputPt.z )
934     {
935       atoms.push_back( inputPt );
936       cout << "Pt: " << inputPt << endl;
937     }
938
939     inputFile.close();
940
941     currSize = atoms.size();
942     updateCurrDL();
943     targetDL = currDL;
```

```
944   sort( atoms.begin(), atoms.end() );
945
946   return 0;
947 }
948
949
950 bool Structure::translate( long double distX, long double distY,
951                            long double distZ )
952 {
953   // Shift all the points in the structure by the given amounts ↪
        ↩ in X, Y and Z
954   // directions.
955
956   for( int i = 0; i < atoms.size(); ++i )
957   {
958     atoms[ i ].x += distX;
959     atoms[ i ].y += distY;
960     atoms[ i ].z += distZ;
961   }
962
963   return true;
964 }
965
966
967 long double getAngle( Point pt1, Point pt2 )
968 {
969   // Assume that pt2 is along the x-axis
970   long double norm = sqrt( pt1.x*pt1.x + pt1.y*pt1.y + ↪
        ↩ pt1.z*pt1.z );
971   return acos( pt1.x/ norm );
972   // return atan2( sqrt( pt1.y* pt1.y + pt1.z* pt1.z ), pt1.x );
973 }
974
975
976 Point getAxis( Point pt1, Point pt2 )
977 {
978   // Assume pt2 is along the x-axis
979   // long double norm = sqrt( pt1.x*pt1.x + pt1.y*pt1.y + ↪
        ↩ pt1.z*pt1.z );
980   long double norm = sqrt( pt1.y*pt1.y + pt1.z*pt1.z );
981   // cout << "pt1 in getAxis: " << pt1 << endl;
982   // cout << "norm in getAxis: " << norm << endl;
983   Point axis;
984
985   axis.x = 0;
```

```
986     axis.y = pt1.z/ norm;
987     axis.z = -pt1.y/ norm;
988 
989     return axis;
990 }
991 
992 
993 bool Structure::rotate( long double angle )
994 {
995     // Turn the structure about the Z axis by the given angle (in
        ↪ ↩ radians).
996 
997     long double cosT = cos( angle );
998     long double sinT = sin( angle );
999     long double oldX, oldY;
1000 
1001     for( int i = 0; i < atoms.size(); ++i )
1002     {
1003       oldX = atoms[i].x;
1004       oldY = atoms[i].y;
1005       atoms[i].x =  oldX*cosT + oldY*sinT;
1006       atoms[i].y = -oldX*sinT + oldY*cosT;
1007     }
1008 
1009     return true;
1010 }
1011 
1012 
1013 bool Structure::rotate( long double angle, long double axisX,
1014                         long double axisY, long double axisZ )
1015 {
1016     // Make sure the axis components are normalized
1017 
1018     // If angle is really small, don't bother with anything
1019     if( fabs( angle ) < 1e-6 )
1020     {
1021       // cout << "small angle: " << angle << endl;
1022       return true;
1023     }
1024 
1025     // http://inside.mines.edu/~gmurray/ArbitraryAxisRotation/
1026     long double cosT = cos( angle );
1027     long double sinT = sin( angle );
1028     long double oldX, oldY, oldZ;
1029 
```

```
1030    for( int i = 0; i < atoms.size(); ++i )
1031    {
1032      oldX = atoms[i].x;
1033      oldY = atoms[i].y;
1034      oldZ = atoms[i].z;
1035      atoms[i].x =  axisX* ( axisX*oldX + axisY*oldY + axisZ*oldZ ↪
            ↩ )* ( 1 - cosT ) + oldX*cosT
1036                  + ( -axisZ*oldY + axisY*oldZ)* sinT;
1037      atoms[i].y =  axisY* ( axisX*oldX + axisY*oldY + axisZ*oldZ ↪
            ↩ )* ( 1 - cosT ) + oldY*cosT
1038                  + ( axisZ*oldX - axisX*oldZ)* sinT;
1039      atoms[i].z =  axisZ* ( axisX*oldX + axisY*oldY + axisZ*oldZ ↪
            ↩ )* ( 1 - cosT ) + oldZ*cosT
1040                  + ( -axisY*oldX + axisX*oldY)* sinT;
1041    }
1042
1043    return true;
1044  }
1045
1046
1047  int Structure::reduceDLprecision( int newPrecision )
1048  {
1049
1050    vector<Point> oldAtoms = atoms;
1051
1052    for( int i = 0; i < atoms.size(); ++i )
1053    {
1054      // FIXME declaring variable inside loop as a quick fix to ↪
            ↩ make it work
1055      stringstream lessPreciseX;
1056      lessPreciseX.precision( newPrecision );
1057      lessPreciseX << atoms[i].x;
1058      lessPreciseX >> atoms[i].x;
1059
1060      stringstream lessPreciseY;
1061      lessPreciseY.precision( newPrecision );
1062      lessPreciseY << atoms[i].y;
1063      lessPreciseY >> atoms[i].y;
1064
1065      stringstream lessPreciseZ;
1066      lessPreciseZ.precision( newPrecision );
1067      lessPreciseZ << atoms[i].z;
1068      lessPreciseZ >> atoms[i].z;
1069    }
1070
```

```
1071    updateCurrDL();
1072    targetDL = currDL;
1073    atoms = oldAtoms;
1074
1075    for( int i = 0; i < targetDL.size(); ++i )
1076    {
1077      stringstream lessPrecise;
1078      lessPrecise.precision( newPrecision );
1079
1080      lessPrecise << targetDL[i];
1081      lessPrecise >> targetDL[i];
1082    }
1083
1084    return 0;
1085  }
1086
1087
1088  vector<Point> Structure::getCore( int coreSize )
1089  {
1090    cout << "coreSize, atoms.size(): " << coreSize << ',' << ↪
         ↩ atoms.size() << endl;
1091    assert( ("Core size <= Size of structure", coreSize <= ↪
         ↩ atoms.size() ) );
1092    vector<Point> corePoints;
1093    corePoints.insert( corePoints.end(), atoms.begin(), ↪
         ↩ atoms.begin() + coreSize );
1094
1095    return corePoints;
1096  }
1097
1098
1099  int Structure::updateUsedDists()
1100  {
1101    int numUsed = 0;
1102    int usedNum = 0;
1103    // return 0; // FIX ME debug mode
1104    int idx;
1105    long double err1, err2;
1106    updateCurrDL();
1107
1108    for( int i = 0; i < currDL.size(); ++i )
1109    {
1110      idx = closestDist( currDL[i], targetDL );
1111      // cout << "idx, currD, tarD, flag: " << idx << ',' << ↪
           ↩ currDL[ i ] << ','
```

```
1112      //      << targetDL[ idx ] << ',' << usedDist[idx] << endl;
1113
1114
1115      // If idx is already excluded; then exclude the neighbour
1116      // FIX ME
1117      if ( usedDist[idx] )
1118      {
1119      //  ++numUsed;
1120      }
1121
1122      // if ( usedDist[idx] and idx -1 >= 0 and idx + 1 < ↪
             ↩ usedDist.size() -1 )
1123      if ( false )
1124      {
1125        err1 = fabs( targetDL[idx + 1] - currDL[i] );
1126        err2 = fabs( targetDL[idx - 1] - currDL[i] );
1127
1128        if ( err1 < err2 )
1129        {
1130          if ( err1 < 0.1 )
1131          {
1132            usedDist[idx + 1] = true;
1133            ++numUsed;
1134          }
1135        }
1136        else
1137        {
1138          if ( err2 < 0.1 )
1139          {
1140            usedDist[idx - 1] = true;
1141            ++numUsed;
1142          }
1143        }
1144      }
1145      else
1146      {
1147        usedDist[ idx ] = true;
1148        ++numUsed;
1149      }
1150      // cout << "excluding " << dlist[idx] << " for " << ↪
             ↩ coreDlist[ i ] << endl;
1151    }
1152
1153    // cout << "Number of used distances: " << numUsed << endl;
1154    return 0;
```

```
1155 }
1156
1157
1158 int Structure::updateFreeDL()
1159 {
1160   int idx;
1161   long double err1, err2;
1162   vector<bool> exclude( targetDL.size(), false );
1163
1164   // updateCurrDL();
1165   for( int i = 0; i < currDL.size(); ++i )
1166   {
1167     idx = closestDist( currDL[i], targetDL );
1168
1169     // If idx is already excluded; then exclude the neighbour
1170     // FIX ME
1171     if( exclude[idx] and idx -1 >= 0 and idx + 1 < ↪
        ↩ exclude.size() -1 )
1172     // if ( false )
1173     {
1174       err1 = fabs( targetDL[idx + 1] - currDL[i] );
1175       err2 = fabs( targetDL[idx - 1] - currDL[i] );
1176
1177       if( err1 < err2 )
1178       {
1179         if( err1 < 0.1 )
1180         {
1181           exclude[idx + 1] = true;
1182         }
1183       }
1184       else
1185       {
1186         if( err2 < 0.1 )
1187         {
1188           exclude[idx - 1] = true;
1189         }
1190       }
1191     }
1192     else
1193     {
1194       exclude[ idx ] = true;
1195     }
1196     // cout << "excluding " << targetDL[idx] << " for " << ↪
        ↩ currDL[ i ] << endl;
1197   }
```

```
1198
1199    freeDL.clear();
1200    vector<long double> usedDlist;
1201    for( int i = 0; i < exclude.size(); ++i )
1202    {
1203      if( exclude[ i ] == false )
1204      {
1205        freeDL.push_back( targetDL[i] );
1206      }
1207      else
1208      {
1209        usedDlist.push_back( targetDL[i] );
1210      }
1211    }
1212
1213    cout << "targetDLErr ( usedDl , coreDl ): "
1214         << distListError( usedDlist, currDL ) << endl;
1215    // assert( freeDlist.size() + currDL.size() >= dlist.size() );
1216    cout << "Number of free distances: " << freeDL.size() << endl;
1217    cout << "Number of used distances: " << usedDlist.size() << endl;
1218    return 0;
1219  }
1220
1221
1222  long double Structure::getPtCost( Point pt )
1223  {
1224    // Function to calculate the cost of an individual point wrt to the structure
1225    // and wrt to the target dlist
1226
1227    vector<long double> ptDlist( atoms.size(), 0 );
1228    // cout << "currSize: " << currSize << endl;
1229
1230    for( int i = 0; i < ptDlist.size(); ++i )
1231    {
1232      ptDlist[i] = getDistance( pt, atoms[i] );
1233      // cout << i << '\t' << ptDlist[i] << endl;
1234    }
1235
1236    // cout << endl;
1237    // getchar();
1238    // cout << "pt cost for " << pt << endl;
1239    return distListError( ptDlist );
1240  }
```

```
1241
1242
1243 bool compareCost( const Point& pt1, const Point& pt2 )
1244 {
1245   return ( pt1.cost < pt2.cost );
1246 }
1247
1248
1249 bool Structure::updatePool( Point pt )
1250 {
1251   if( pool.size() == maxPoolSize and pt.cost > pool[ ↪
       ↩ pool.size()-1 ].cost )
1252   {
1253     return false;
1254   }
1255
1256   // Make sure that the same point is not in the pool
1257   bool repeat = false;
1258   int orgIdx = -1;
1259
1260   for( int i = 0; i < pool.size(); ++i )
1261   {
1262     if( getDistance( pt, pool[i] ) < 0.2 )
1263     {
1264       if( pt.cost < pool[ i ].cost )
1265       {
1266         pool.erase( pool.begin() + i );
1267       }
1268       else
1269       {
1270         repeat = true;
1271         orgIdx = i;
1272       }
1273     }
1274   }
1275
1276   if( repeat == false )
1277   {
1278     vector<Point>::const_iterator const it
1279       = lower_bound( pool.begin(), pool.end(), pt, compareCost );
1280     int idx = it - pool.begin();
1281     pool.insert( pool.begin() + idx, pt );
1282     cout << "adding pt: " << pt << ',' << pt.cost << endl;
1283   }
1284
```

```
1285    while( pool.size() > maxPoolSize )
1286    {
1287      pool.pop_back();
1288    }
1289
1290    return true;
1291  }
1292
1293
1294  bool Structure::doBuildup2( int newBase )
1295  {
1296    Point smPt;
1297    smPt.x = 2.42671232;
1298    smPt.y = 0.24021148;
1299    long double triToler = 0.1;
1300    Point zero;
1301    zero.x = zero.y = zero.z = 0;
1302    if( newBase == 0 )
1303    {
1304      newBase = 1;
1305    }
1306
1307    // FIXME quick hack as pt1 of base is already at origin
1308    long double distA = getDistance( zero, atoms[ newBase ] );
1309    long double distB = 0, distC = 0;
1310    long double error = 0.0L;
1311    Point testPt;
1312    int numTestPts = 0;
1313    cout << "distA: " << distA << endl;
1314
1315    for( int bIdx = 0; bIdx < targetDL.size(); bIdx++)
1316    {
1317      if( usedDist[ bIdx ] )
1318      {
1319        continue;
1320      }
1321
1322      distB = targetDL[ bIdx ];
1323      for( int cIdx = bIdx + 1; cIdx < targetDL.size(); cIdx++)
1324      {
1325        if( usedDist[ cIdx ] )
1326        {
1327          continue;
1328        }
1329        distC = targetDL[ cIdx ];
```

```
1330
1331        if( distA > distC )
1332        {
1333          if( ( distB + distC + triToler ) < distA )
1334          {
1335            continue;
1336          }
1337        }
1338        else
1339        {
1340          if( ( distA + distB + triToler ) < distC )
1341          {
1342            break;
1343          }
1344        }
1345
1346        // Place testPt
1347        // cout << "placing testPt" << endl;
1348        testPt.x = distA/2 - ( distC - distB )* ( distC + distB ↪
              ↩ )/ ( 2* distA );
1349        testPt.y = sqrt( ( distC + testPt.x - distA )*
1350                         ( distC - testPt.x + distA ) );
1351
1352        if( getDistance( testPt, smPt ) < 0.2 )
1353        {
1354          cout << "Missing point!" << endl;
1355          // getchar();
1356        }
1357        // In the case of nearly collinear points, to caclulate ↪
              ↩ y, we maybe
1358        // taking the square root of a negative number. In such ↪
              ↩ cases, damp
1359        // it to zero.
1360        if( testPt.y != testPt.y )
1361        {
1362          testPt.y = 0;
1363        }
1364
1365        for( int m = 0; m < 2; m++ )
1366        {
1367          if( m == 1 )
1368          {
1369            testPt.x = distA - testPt.x;
1370          }
1371
```

```
1372            for( int n = 0; n < 2; n++)
1373            {
1374              if( n == 1 )
1375              {
1376                testPt.y = - testPt.y;
1377              }
1378
1379              testPt.cost = error = getPtCost( testPt );
1380              if( ( testPt.cost - 0.00478091 ) < 0.001 )
1381              {
1382                cout << "testPt, error: " << testPt << ", " << ↪
                     ↩ error << endl;
1383                cout << "bIdx, cIdx, m, n: " << bIdx << ',' << cIdx ↪
                     ↩ << ','
1384                     << m << ',' << n << endl;
1385                // getchar();
1386              }
1387              if( error < 0.2 )
1388              {
1389                updatePool( testPt );
1390                ++numTestPts;
1391              }
1392            } // n loop
1393          } // m loop
1394        } // c loop
1395      } // b loop
1396
1397      cout << "Number of points tested: " << numTestPts << endl;
1398      return 0;
1399    }
1400
1401
1402    bool Structure::doBuildup3( int newBase )
1403    {
1404      Point smPt;
1405      smPt.x = 2.42671232;
1406      smPt.y = 0.24021148;
1407      long double triToler = 0.1;
1408      Point zero;
1409      zero.x = zero.y = zero.z = 0;
1410      if( newBase == 0 )
1411      {
1412        newBase = 1;
1413      }
1414
```

```
1415    // FIXME quick hack as pt1 of base is already at origin
1416    long double distA = getDistance( zero, atoms[ newBase ] );
1417    long double distB = 0, distC = 0;
1418    long double error = 0.0L;
1419    Point testPt;
1420    int numTestPts = 0;
1421    cout << "distA: " << distA << endl;
1422
1423    // updateFreeDL();
1424    // for( int bIdx = 0; bIdx < targetDL.size(); bIdx++)
1425    for( int bIdx = 0; bIdx < freeDL.size(); bIdx++)
1426    {
1427      // if( usedDist[ bIdx ] )
1428      // {
1429      //   continue;
1430      // }
1431
1432      distB = freeDL[ bIdx ];
1433      for( int cIdx = bIdx + 1; cIdx < freeDL.size(); cIdx++)
1434      {
1435        // if( usedDist[ cIdx ] )
1436        // {
1437        //   continue;
1438        // }
1439        distC = freeDL[ cIdx ];
1440
1441        if( distA > distC )
1442        {
1443          if( ( distB + distC + triToler ) < distA )
1444          {
1445            continue;
1446          }
1447        }
1448        else
1449        {
1450          if( ( distA + distB + triToler ) < distC )
1451          {
1452            break;
1453          }
1454        }
1455
1456        // cout << "bIdx, cIdx: " << bIdx << '\t' << cIdx << endl;
1457        // Place testPt
1458        // cout << "placing testPt" << endl;
1459        testPt.x = distA/2 - ( distC - distB )* ( distC + distB ↪
```

```
      ↩ )/ ( 2* distA );
1460  testPt.y = sqrt( ( distC + testPt.x - distA )*
1461                   ( distC - testPt.x + distA ) );
1462
1463  if( getDistance( testPt, smPt ) < 0.2 )
1464  {
1465    cout << "Missing point!" << endl;
1466    // getchar();
1467  }
1468  // In the case of nearly collinear points, to caclulate ↩
      ↩ y, we maybe
1469  // taking the square root of a negative number. In such ↩
      ↩ cases, damp
1470  // it to zero.
1471  if( testPt.y != testPt.y )
1472  {
1473    testPt.y = 0;
1474  }
1475
1476  for( int m = 0; m < 2; m++ )
1477  {
1478    if( m == 1 )
1479    {
1480      testPt.x = distA - testPt.x;
1481    }
1482
1483    for( int n = 0; n < 2; n++)
1484    {
1485      if( n == 1 )
1486      {
1487        testPt.y = - testPt.y;
1488      }
1489
1490      testPt.cost = error = getPtCost( testPt );
1491      // if( ( testPt.cost - 0.00478091 ) < 0.001 )
1492      // {
1493      //   cout << "testPt, error: " << testPt << ", " << ↩
            ↩ error << endl;
1494      //   cout << "bIdx, cIdx, m, n: " << bIdx << ',' << ↩
            ↩ cIdx << ','
1495      //        << m << ',' << n << endl;
1496      //   // getchar();
1497      // }
1498      // cout << "testPt, cost: " << testPt << ',' << error ↩
            ↩ << endl;
```

```
1499
1500            if( error < 0.2 )
1501            {
1502              updatePool( testPt );
1503              ++numTestPts;
1504            }
1505          } // n loop
1506        } // m loop
1507      } // c loop
1508    } // b loop
1509
1510    cout << "Number of points tested: " << numTestPts << endl;
1511    return 0;
1512  }
1513
1514
1515  long double Structure::getPtsCost( Point pt1, Point pt2 )
1516  {
1517    // Replacing brute force implementation with the one based on
1518    // reusing the cost of p1, p2 calculated earlier.
1519
1520    vector<long double> ptsDlist;
1521    for( int i = 0; i < atoms.size(); ++i )
1522    {
1523      ptsDlist.push_back( getDistance( pt1, atoms[i] ) );
1524      ptsDlist.push_back( getDistance( pt2, atoms[i] ) );
1525    }
1526
1527    ptsDlist.push_back( getDistance( pt1, pt2 ) );
1528    sort( ptsDlist.begin(), ptsDlist.end() );
1529
1530    // cout << "pt1, pt2" << endl;
1531    // cout << pt1 << endl;
1532    // cout << pt2 << endl;
1533    // for( int i = 0; i < ptsDlist.size(); ++i )
1534    // {
1535    //   cout << "Dlist: " << i << '\t' << ptsDlist[ i ] << endl;
1536    // }
1537
1538    if ( ptsDlist[0] < 0.33* targetDL[ 0 ] )
1539    {
1540      // cout << "overlap " << endl;
1541      // getchar();
1542      return 1.0;
1543    }
```

```
1544
1545    return distListError( ptsDlist );
1546  }
1547
1548
1549  int Structure::insertPoint( Point pt )
1550  {
1551    // FIXME faster to use binary search
1552    Point zero;
1553
1554    for( int idx = 0; idx < atoms.size(); ++idx )
1555    {
1556      if( getDistance( pt, zero ) < getDistance( atoms[ idx ], ↪
           ↩ zero ) )
1557      {
1558        atoms.insert( atoms.begin() + idx, pt );
1559        return 0;
1560      }
1561    }
1562
1563    atoms.push_back( pt );
1564    return 0;
1565  }
1566
1567
1568  bool Structure::growStru()
1569  {
1570    printPool();
1571
1572    bool growthFlag = false;
1573    if( pool.size() < 2 )
1574    {
1575      cout << "Small pool" << endl;
1576      return false;
1577    }
1578
1579    long double cost2pts = 0;
1580    long double minCost = 1e6;
1581    size_t idx1best = 0, idx2best = 0;
1582
1583    for( int idx1 = 0; idx1 < pool.size(); ++idx1 )
1584    {
1585      for( int idx2 = idx1 + 1; idx2 < pool.size(); ++idx2 )
1586      {
1587        cost2pts = getPtsCost( pool[ idx1 ], pool[ idx2 ] );
```

```
1588        // cout << "idx1, idx2, cost: " << idx1 << ", " << idx2 << ", "
1589        //      << setprecision( 8 ) << cost2pts << endl;
1590        if( cost2pts < minCost )
1591        {
1592          idx1best = idx1;
1593          idx2best = idx2;
1594          minCost = cost2pts;
1595        }
1596      } // idx2 loop
1597    } // idx1 loop
1598
1599    if( minCost < 1 )
1600    {
1601      // cout << "best idx: " << idx1best << ',' << idx2best << ';'
1602      //      << " potential cost: " << setprecision( 8 ) << minCost << endl;
1603
1604      insertPoint( pool[ idx1best ] );
1605      insertPoint( pool[ idx2best ] );
1606      growthFlag = true;
1607      updateCurrDL();
1608      cout << "adding pts with idx: " << idx1best << ", " << idx2best << endl;
1609      cout << "*** " << pool[ idx1best ] << endl;
1610      cout << "*** " << pool[ idx2best ] << endl;
1611    }
1612
1613    // getchar();
1614    return growthFlag;
1615  }
1616
1617
1618  int Structure::printPool()
1619  {
1620    cout << "**** " << "Points in the pool and their cost" << " ****" << endl;
1621
1622    for( int i = 0; i < pool.size(); ++i )
1623    {
1624      // cout << i << '\t' << pool[i] << '\t' << setprecision( 8 ) << ) << pool[i].cost
1625      //      << endl;
1626      long double ptCost = getPtCost( pool[ i ] );
1627      cout << i << '\t' << pool[i] << '\t' << setprecision( 8 )
```

```
        ↩ << ptCost
1628          << endl;
1629    }
1630    cout << endl;
1631    // getchar();
1632
1633    return 0;
1634  }
1635
1636
1637  int Structure::getPools()
1638  {
1639    // func that gets and combines the pools from the 3 bonds in ↪
        ↩ the base triangle
1640    // and then is ready for doing buildup
1641
1642    // FIXME quick hack to fix some failed reconstructions
1643    // srand( time( NULL ) );
1644
1645    int numPools = 1;
1646    int basePt1 = 0, basePt2 = 1;
1647    int oldBasePt1 = basePt1, oldBasePt2 = basePt2;
1648    vector<Point> oldAtoms = atoms, newPool;
1649    long double newOx, newOy, newOz, angle;
1650
1651    for( int i = 1; i <= numPools; ++i )
1652    {
1653      do
1654      {
1655        basePt1 = rand() % atoms.size();
1656        basePt2 = rand() % atoms.size();
1657        if( basePt1 > basePt2 )
1658        {
1659          swap( basePt1, basePt2 );
1660        }
1661      } while( basePt2 == basePt1
1662               or basePt1 == oldBasePt1
1663               or basePt2 == oldBasePt2
1664               or basePt1 == oldBasePt2
1665               or basePt2 == oldBasePt1 );
1666
1667      oldBasePt1 = basePt1;
1668      oldBasePt2 = basePt2;
1669      cout << "newBase: " << basePt1 << '\t' << basePt2 << endl;
1670
```

```
1671        newOx = atoms[ basePt1 ].x;
1672        newOy = atoms[ basePt1 ].y;
1673        newOz = atoms[ basePt1 ].z;
1674
1675        // angle = atan2( atoms[ basePt2 ].y - atoms[ basePt1 ].y,
1676        //                atoms[ basePt2 ].x - atoms[ basePt1 ].x );
1677        // translate( -newOx, -newOy, -newOz );
1678        // rotate( angle );
1679
1680        // cout << "before buildup" << endl;
1681        // print();
1682        // doBuildup2( basePt2 );
1683
1684        // doBuildup3( basePt2 );
1685
1686        // translate( -newOx, -newOy, -newOz );
1687        // Point pt2;
1688        // pt2.x = 1; pt2.y = 0; pt2.z = 0;
1689        // long double angle = getAngle( atoms[ basePt2 ], pt2 );
1690        // Point axis = getAxis( atoms[ basePt2 ], pt2 );
1691        // rotate( angle, axis.x, axis.y, axis.z );
1692
1693        doBuildup3Dv2( 0, 1, 2 );
1694        atoms.insert( atoms.end(), pool.begin(), pool.end() );
1695
1696        // rotate( -angle, axis.x, axis.y, axis.z );
1697        // translate( newOx, newOy, newOz );
1698
1699        // rotate( -angle );
1700        // translate( newOx, newOy, newOz );
1701
1702        // cout << "after buildup" << endl;
1703        // print();
1704
1705        newPool.insert( newPool.end(), atoms.begin() + ↪
              ↩ oldAtoms.size(),
1706                        atoms.end() );
1707        pool.clear();
1708        atoms = oldAtoms;
1709    }
1710
1711    for( int j = 0; j < newPool.size(); ++j )
1712    {
1713        updatePool( newPool[ j ] );
1714    }
```

```
1715
1716    return 0;
1717  }
1718
1719
1720  int Structure::getPools2()
1721  {
1722    // func that gets and combines the pools from the 3 bonds in ↪
         ↩ the base triangle
1723    // and then is ready for doing buildup
1724
1725    // FIXME quick hack to fix some failed reconstructions
1726    // srand( time( NULL ) );
1727
1728    int numPools = 2;
1729    int basePt1 = 0, basePt2 = 1, basePt3 = 2;
1730    int oldBasePt1 = basePt1, oldBasePt2 = basePt2, oldBasePt3 = ↪
         ↩ basePt3;
1731    vector<Point> oldAtoms = atoms, newPool;
1732    long double newOx, newOy, newOz, angle;
1733
1734    for( int i = 1; i <= numPools; ++i )
1735    {
1736      do
1737      {
1738        basePt1 = rand() % atoms.size();
1739        basePt2 = rand() % atoms.size();
1740        basePt3 = rand() % atoms.size();
1741      // } while(  basePt3 == oldBasePt3
1742      //           or basePt3 == basePt1
1743      //           or basePt3 == basePt2 );
1744      } while( basePt2 == basePt1
1745               or basePt3 == basePt2
1746               or basePt3 == basePt1
1747               or basePt1 == oldBasePt1
1748               or basePt2 == oldBasePt2
1749               or basePt1 == oldBasePt3
1750               or basePt1 == oldBasePt2
1751               or basePt2 == oldBasePt1
1752               or basePt3 == oldBasePt3
1753               or basePt3 == oldBasePt1
1754               or basePt3 == oldBasePt2 );
1755
1756      cout << "newBase: " << basePt1 << '\t' << basePt2 << '\t'
1757                          << basePt3 << endl;
```

```
1758
1759        oldBasePt1 = basePt1;
1760        oldBasePt2 = basePt2;
1761        oldBasePt3 = basePt3;
1762
1763        int sum = basePt1 + basePt2 + basePt3;
1764        basePt1 = min( min( oldBasePt1, oldBasePt2 ), oldBasePt3 );
1765        basePt3 = max( max( oldBasePt1, oldBasePt2 ), oldBasePt3 );
1766        basePt2 = sum - basePt1 - basePt3;
1767
1768        oldBasePt1 = basePt1;
1769        oldBasePt2 = basePt2;
1770        oldBasePt3 = basePt3;
1771        cout << "newBase: " << basePt1 << '\t' << basePt2 << '\t'
1772                            << basePt3 << endl;
1773
1774        // basePt1 = 0; basePt2 = 1; basePt3 = atoms.size() - 1;
1775        // basePt1 = 1; basePt2 = 2; basePt3 = 3;
1776        newOx = atoms[ basePt1 ].x;
1777        newOy = atoms[ basePt1 ].y;
1778        newOz = atoms[ basePt1 ].z;
1779
1780        translate( -newOx, -newOy, -newOz );
1781        // cout << "Translate to new origin" << endl;
1782        // print();
1783        // getchar();
1784
1785        Point pt2;
1786        pt2.x = 1; pt2.y = 0; pt2.z = 0;
1787        long double angle = getAngle( atoms[ basePt2 ], pt2 );
1788        Point axis = getAxis( atoms[ basePt2 ], pt2 );
1789
1790        // cout << atoms[ basePt2 ] << endl;
1791        // cout << "angle, axis: " << angle << ',' << axis.x << ','
1792        //                         << axis.y << ',' << axis.z << endl;
1793        rotate( angle, axis.x, axis.y, axis.z );
1794
1795        // long double angle2 = getAngle( atoms[ basePt2 ], pt2 );
1796        // Point axis2 = getAxis( atoms[ basePt2 ], pt2 );
1797        //
1798        // cout << atoms[ basePt2 ] << endl;
1799        // cout << "angle, axis: " << angle2 << ',' << axis2.x << ','
1800        //                         << axis2.y << ',' << axis2.z << endl;
1801        // rotate( angle2, axis2.x, axis2.y, axis2.z );
```

```
1802
1803        // cout << "Rotate to make new base bond along the X axis" <<
               endl;
1804        // print();
1805        // getchar();
1806        long double angle3 = atan2( atoms[ basePt3 ].z, atoms[
               basePt3 ].y );
1807        cout << "-angle3, axis: " << -angle3 << ',' << "1" << ','
1808                                  << "0" << ',' << "0" << endl;
1809        rotate( -angle3, 1, 0, 0 );
1810        // print();
1811        // cout << "angle3, axis: " << angle3 << ',' << "1" << ','
1812        //                           << "0" << ',' << "0" << endl;
1813        // rotate( angle3, 1, 0, 0 );
1814        // print();
1815        // getchar();
1816        // return 1;
1817
1818        doBuildup3Dv2( basePt1, basePt2, basePt3 );
1819        atoms.insert( atoms.end(), pool.begin(), pool.end() );
1820        rotate( angle3, 1, 0, 0 );
1821        // print();
1822        rotate( -angle, axis.x, axis.y, axis.z );
1823        // print();
1824        translate( newOx, newOy, newOz );
1825        // print();
1826        // getchar();
1827
1828        cout << "Size of newPool: " << newPool.size() << endl;
1829        newPool.insert( newPool.end(), atoms.begin() +
               oldAtoms.size(),
1830                        atoms.end() );
1831        cout << "Size of newPool: " << newPool.size() << endl;
1832        pool.clear();
1833
1834        atoms.clear();
1835        atoms = oldAtoms;
1836    }
1837
1838    for( int j = 0; j < newPool.size(); ++j )
1839    {
1840        cout << "update, ";
1841        updatePool( newPool[ j ] );
1842        // getchar();
1843    }
```

```
1844
1845   return 0;
1846 }
1847
1848
1849 int print2structures( vector<Point>& stru1, vector<Point>& ↪
       ↩ stru2 )
1850 {
1851   // For convenience print stru1, stru2 format
1852   assert( stru1.size() >= stru2.size() );
1853   cout << "**** " << "Printing 2 structures" << " ****" << endl;
1854   Point zero;
1855   long double r1, r2, diff;
1856   int i = 0, j = 0;
1857   long double distToler = 0.5;
1858
1859   while( i < stru1.size() and j < stru2.size() )
1860   {
1861     r1 = getDistance( stru1[i], zero );
1862     r2 = getDistance( stru2[j], zero );
1863     diff = getDistance( stru1[i], stru2[j] );
1864
1865     if( fabs( r1 -r2 ) < distToler and diff < distToler )
1866     {
1867       cout << i << '\t' << "( " << stru1[i] << " )"
1868            << '\t' << "( " << stru2[j] << " )"
1869            << '\t' << getDistance( stru1[i], stru2[j] );
1870       ++i;
1871       ++j;
1872     }
1873     else if( r1 < r2 )
1874     {
1875       cout << i << '\t' << "( " << stru1[i] << " )"
1876            << '\t' << "( " << zero << " )"
1877            << '\t' << getDistance( stru1[i], zero );
1878       ++i;
1879     }
1880     else
1881     {
1882       cout << "-1" << '\t' << "( " << zero << " )"
1883            << '\t' << "( " << stru2[j] << " )"
1884            << '\t' << getDistance( zero, stru2[j] );
1885       ++j;
1886     }
1887
```

```
1888      cout << endl;
1889    }
1890
1891    while( i != stru1.size() )
1892    {
1893        cout << i << '\t' << "( " << stru1[i] << " )" << endl;
1894        ++i;
1895    }
1896
1897    while( j != stru2.size() )
1898    {
1899        cout << "-1" << '\t' << "( " << zero << " )"
1900             << '\t' << "( " << stru2[j] << " )"
1901             << '\t' << getDistance( zero, stru2[j] ) << endl;
1902        ++j;
1903    }
1904    cout << endl;
1905
1906    // cout << "i, j: " << i << ',' << j << endl;
1907    return 0;
1908  }
1909
1910
1911  bool Structure::reconstruct2()
1912  {
1913    bool growth = false;
1914    int resetNum = 0;
1915    vector<Point> givenCore = atoms;
1916    updateCurrDL();
1917    cout << "size of given core: " << givenCore.size() << endl;
1918
1919    while( atoms.size() < targetSize )
1920    {
1921      // updateUsedDists();
1922      // updateCurrDL();
1923      updateFreeDL();
1924      cout << "size of freeDL: " << freeDL.size() << endl;
1925
1926      if( false and growth and atoms.size() >= 20 and
1927          targetSize > 20 and ( atoms.size()/ 2 ) % 2 == 0 )
1928      {
1929      }
1930      else
1931      {
1932        pool.clear();
```

```
1933        // getPools();
1934        getPools2();
1935      }
1936      cout << "pool.size: " << pool.size() << endl;
1937      cout << "Pool pt0, cost: " << pool[0] << ',' << getPtCost( ↪
             ↩ pool[0] ) << endl;
1938
1939      growth = growStru();
1940      sort( atoms.begin(), atoms.end() );
1941      // cout << "solution size: " << atoms.size() << endl;
1942      if( resetNum >= 2 )
1943      {
1944        break;
1945      }
1946      if( growth == false )
1947      {
1948        print();
1949        atoms = givenCore;
1950        updateCurrDL();
1951        ++resetNum;
1952      }
1953      // getchar();
1954    }
1955
1956    return ( atoms.size() == targetSize );
1957    // return growth;
1958  }
1959
1960
1961  bool Structure::findCore3D()
1962  {
1963    bool successFlag = false;
1964
1965    // 10 bonds in the tetrahedron
1966    int idxA = 0, idxB, idxC, idxD, idxE, idxF, idxG, idxH, idxI, ↪
         ↩ idxJ, idxM;
1967    int bridgeIdx;
1968    long double counter = 0; // number of tetrahedra
1969    long double bridgeDist = 0;
1970    long double fmin, fmax, imin, imax, fracError;
1971
1972    int invC = 0; // # invalid cores
1973    // bond window
1974    int winStart = 0, inc = 10, winStop = winStart + inc; // ↪
         ↩ windowing on
```

```
1975    vector<long double> dlist = targetDL;
1976    cout << "targetDL size: " << targetDL.size() << endl;
1977
1978    Point basePt1, basePt2, basePt3,
1979          apexPt1, apexPt2;
1980    basePt1.x = 0; basePt1.y = 0; basePt1.z = 0;
1981    basePt2.x = dlist[ idxA ]; basePt2.y = 0; basePt2.z = 0;
1982    cout << "basePt1: " << basePt1.x << " " << basePt1.y << " " ↪
          ↩ << basePt1.z
1983         << endl;
1984    cout << "basePt2: " << basePt2.x << " " << basePt2.y << " " ↪
          ↩ << basePt2.z
1985         << endl;
1986    cout << "Bond window: ";
1987
1988    int countC = 0; // count number of cores
1989    vector<int> idxArr( 6, 0 );
1990    while( true ) // window loop
1991    {
1992      cerr << "->" << winStop;
1993
1994      // base triangle
1995      for( idxB = 1; idxB < winStop; ++idxB )
1996      {
1997        // cout << endl << "idxB:" << idxB << ", "; // endl;
1998        // cout << "idxC: ";
1999        for( idxC = idxB + 1; idxC < winStop; ++idxC )
2000        {
2001          // cout << "_" << idxC << "_"; // << endl;
2002          // NOTE: c>b so we have to check only 1 triangle ↪
                ↩ inequality
2003          if( dlist[ idxA ] + dlist[ idxB ] + toler < dlist[ idxC ↪
                ↩ ] ) break;
2004
2005          placeApex( basePt3, idxA, idxB, idxC, dlist );
2006
2007          for( idxD = 1; idxD < winStop; ++idxD )
2008          {
2009            if( idxD == idxB  or idxD == idxC ) continue;
2010            if( idxD < idxB ) continue;
2011            for( idxE = 1; idxE < winStop; ++idxE )
2012            {
2013              if( idxE < idxB ) continue;
2014              if( idxE == idxB or idxE == idxC or idxE == idxD ) ↪
                    ↩ continue;
```

```
2015            // triangle inequalities
2016            if( dlist[ idxA ] + dlist[ idxE ] + toler < dlist[ ↪
                   ↩ idxD ] ) continue;
2017            if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ ↪
                   ↩ idxE ] ) break;
2018
2019            placeApex( apexPt1, idxA, idxD, idxE, dlist );
2020
2021            fmin = getDistance( basePt3, apexPt1 );
2022            apexPt1.y = - apexPt1.y;
2023            fmax = getDistance( basePt3, apexPt1 );
2024
2025            for( idxF = 1; idxF < dlist.size(); ++idxF )
2026            {
2027              if( dlist[idxF] < ( fmin - toler ) ) continue;
2028              if( dlist[idxF] > ( fmax + toler ) ) break;
2029              if( idxF == idxB or idxF == idxC or idxF == idxD ↪
                     ↩ or idxF == idxE ) continue;
2030
2031              idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxF;
2032              idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxE;
2033              idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxD;
2034              placeTop( basePt1, basePt2, basePt3, apexPt1, ↪
                     ↩ idxArr, dlist );
2035
2036              for( idxG = 1; idxG < winStop; ++idxG )
2037              {
2038                if( idxG < idxB ) continue;
2039                if( idxG == idxB or idxG == idxC or idxG == ↪
                       ↩ idxD or
2040                    idxG == idxE or idxG == idxF ) continue;
2041                if( idxG < idxD ) continue;
2042                for( idxH = 1; idxH < winStop; ++idxH )
2043                {
2044                  if( idxH < idxB ) continue;
2045                  if( idxH ==idxB or idxH == idxC or idxH == ↪
                         ↩ idxD or
2046                      idxH == idxE or idxH == idxF or idxH == ↪
                           ↩ idxG ) continue;
2047                  if( dlist[ idxA ] + dlist[ idxH ] + toler < ↪
                         ↩ dlist[ idxG ] ) continue; // triangle ↪
                         ↩ inequality
2048                  if( dlist[ idxA ] + dlist[ idxG ] + toler < ↪
                         ↩ dlist[ idxH ] ) break; // triangle ↪
                         ↩ inequality
```

```
2049
2050                placeApex( apexPt2, idxA, idxG, idxH, dlist );
2051
2052                imin = getDistance( basePt3, apexPt2 );
2053                apexPt2.y = - apexPt2.y;
2054                imax = getDistance( basePt3, apexPt2 );
2055                // cout << "bp3, ap2: " << basePt3 << ',' << ↪
                       ↩ apexPt2 << endl;
2056
2057                for( idxI = 1; idxI < dlist.size(); ++idxI )
2058                {
2059                  if( idxI == idxB or idxI == idxC or idxI == ↪
                         ↩ idxD or
2060                      idxI == idxE or idxI == idxF or idxI == ↪
                             ↩ idxG or
2061                      idxI == idxH ) continue;
2062                  // correct windowing
2063                  if( idxH < winStart and idxG < winStart and
2064                      idxE < winStart and idxD < winStart and
2065                      idxC < winStart and idxB < winStart) ↪
                             ↩ continue;
2066
2067                  if( dlist[ idxI ] < ( imin - toler ) ) ↪
                         ↩ continue;
2068                  if( dlist[ idxI ] > ( imax + toler ) ) break;
2069
2070                  // cout << "abcdefghi: " << idxA << ',' << ↪
                         ↩ idxB << ','
2071                  //                      << idxC << ',' << ↪
                         ↩ idxD << ','
2072                  //                      << idxE << ',' << ↪
                         ↩ idxF << ','
2073                  //                      << idxG << ',' << ↪
                         ↩ idxH << ','
2074                  //                      << idxI << endl;
2075                  // cout << "dlist-I, imin, imax: " << ↪
                         ↩ dlist[ idxI ] << ',' << imin << ',' << ↪
                         ↩ imax << endl << endl;
2076
2077                  idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxI;
2078                  idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxH;
2079                  idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxG;
2080                  placeTop( basePt1, basePt2, basePt3, ↪
                         ↩ apexPt2, idxArr, dlist );
2081
```

```
2082                    for( idxM = 0; idxM < 2; ++idxM )
2083                    {
2084                      apexPt2.z = pow( -1.0L, idxM )* ↪
                            ↩ apexPt2.z; // reflect about XY plane
2085
2086                      bridgeDist = getDistance( apexPt1, ↪
                            ↩ apexPt2 );
2087                      counter += 1;
2088
2089                      if( bridgeDist != bridgeDist ) getchar(); ↪
                            ↩ // cout << p4 << '\t' << p5 << endl;
2090                      if( ( bridgeDist < ( dlist[ 0 ] - ↪
                            ↩ bridgeDist* toler ) ) or
2091                        ( bridgeDist > ( dlist[ dlist.size() ↪
                              ↩ - 1 ] + bridgeDist* toler ) ) ) ↪
                              ↩ continue;
2092                      idxJ = closestDist( bridgeDist, dlist );
2093
2094                      if( idxJ == idxA and idxJ == idxB and ↪
                            ↩ idxJ == idxC and
2095                        idxJ == idxD and idxJ == idxE and ↪
                              ↩ idxJ == idxF and
2096                        idxJ == idxG and idxJ == idxH and ↪
                              ↩ idxJ == idxI )
2097                      {
2098                        continue;
2099                      }
2100
2101                      fracError = fabs( dlist[ idxJ ] - ↪
                            ↩ bridgeDist )/ bridgeDist;
2102                      if( fracError < toler )
2103                      {
2104                        atoms.push_back( basePt1 );
2105                        atoms.push_back( basePt2 );
2106                        atoms.push_back( basePt3 );
2107
2108                        atoms.push_back( apexPt1 );
2109                        atoms.push_back( apexPt2 );
2110
2111                        cout << endl << "3D core found!!!" << ↪
                              ↩ endl;
2112                        cout << bridgeDist << '\t' << dlist[ ↪
                              ↩ idxJ ] << '\t'
2113                          << fabs( bridgeDist - dlist[ idxJ ↪
                                ↩ ] ) << endl;
```

```
2114                    cout << "bridgeDist: " << bridgeDist << endl;
2115                    cout << "core finder counter: " << counter << endl;
2116
2117                    for( int i = 0; i < atoms.size(); ++i )
2118                    {
2119                      cout << "Point: " << i + 1 << '\t' << atoms[ i ] << endl;
2120                    }
2121
2122                    // Adding the buildup inside core finder to deal
2123                    // with bad cores
2124                    doBuildup3D( idxArr );
2125                    if( atoms.size() >= min( 8, targetSize ) )
2126                    {
2127                      // If buildup was able to add 4 more points then with
2128                      // high probability, we have the right structure.
2129                      cout << " atoms size: " << atoms.size() << endl;
2130                      return true;
2131                    }
2132                    else
2133                    {
2134                      // If buildup could not even 4 points then with a very
2135                      // high probability, we have the wrong structure. Start
2136                      // over and find the next core.
2137                      cout << "No buildup, bad core. "
2138                              "Finding the next core ... " << endl;
2139                      atoms.clear();
2140                      updateCurrDL();
2141                      // cout << "Bond window: ";
2142                    }
2143                    // return true;
2144                  }
2145                } // m loop
2146              } // i loop
2147            } // h loop
```

```
2148                 } // g loop
2149               } // f loop
2150             } // e loop
2151           } // d loop
2152         } // c loop
2153       } // b loop
2154
2155       cout << "counter: "<< counter << endl;
2156       winStart = winStop; winStop += inc;
2157       if( winStart == dlist.size() ) break;
2158
2159       if( winStop > dlist.size() )
2160       {
2161         winStop = dlist.size();
2162       }
2163
2164       cout << "searching in a "<< winStop << " bond window" << endl;
2165     } // window while loop
2166
2167     return successFlag;
2168 }
2169
2170
2171 bool Structure::doBuildup3D( vector<int> idxArr )
2172 {
2173     bool successFlag = false;
2174
2175     // 10 bonds in the tetrahedron
2176     int idxA = idxArr[ 0 ], idxB = idxArr[ 1 ], idxC = idxArr[ 2 ],
2177         idxD = idxArr[ 3 ], idxE = idxArr[ 4 ], idxF = idxArr[ 5 ],
2178         idxG, idxH, idxI, idxJ, idxM;
2179
2180     int bridgeIdx;
2181     long double counter = 0; // number of tetrahedra
2182     long double bridgeDist = 0;
2183     long double fmin, fmax, imin, imax, fracError;
2184
2185     int invC = 0; // # invalid cores
2186     // bond window
2187     vector<long double> dlist = targetDL;
2188     int winStart = 0, winStop = dlist.size(); // windowing off
2189
2190     Point basePt1 = atoms[ 0 ], basePt2 = atoms[ 1 ], basePt3 = atoms[ 2 ],
2191         apexPt1 = atoms[ 3 ], apexPt2;
```

```
2192
2193    int countC = 0; // count number of cores
2194
2195    for( idxG = 0; idxG < winStop; ++idxG )
2196    {
2197      if( idxG == idxB or idxG == idxC or idxG == idxD or
2198          idxG == idxE or idxG == idxF ) continue;
2199      if( usedDist[ idxG ] ) continue;
2200
2201      for( idxH = 0; idxH < winStop; ++idxH )
2202      {
2203        if( idxH ==idxB or idxH == idxC or idxH == idxD or
2204            idxH == idxE or idxH == idxF or idxH == idxG ) continue;
2205        if( usedDist[ idxH ] ) continue;
2206
2207        if( dlist[ idxA ] + dlist[ idxH ] + toler < dlist[ idxG ] ↪
              ↩ ) continue; // triangle inequality
2208        if( dlist[ idxA ] + dlist[ idxG ] + toler < dlist[ idxH ] ↪
              ↩ ) break; // triangle inequality
2209        if( dlist[ idxH ] + dlist[ idxG ] + toler < dlist[ idxA ] ↪
              ↩ ) continue; // triangle inequality
2210
2211        placeApex( apexPt2, idxA, idxG, idxH, dlist );
2212
2213        imin = getDistance( basePt3, apexPt2 );
2214        apexPt2.y = - apexPt2.y;
2215        imax = getDistance( basePt3, apexPt2 );
2216        // cout << "bp3, ap2: " << basePt3 << ',' << apexPt2 << ↪
              ↩ endl;
2217
2218        for( idxI = 0; idxI < dlist.size(); ++idxI )
2219        {
2220          if( idxI == idxB or idxI == idxC or idxI == idxD or
2221              idxI == idxE or idxI == idxF or idxI == idxG or
2222              idxI == idxH ) continue;
2223          if( usedDist[ idxI ] ) continue;
2224
2225          if( dlist[ idxI ] < ( imin - toler ) ) continue;
2226          if( dlist[ idxI ] > ( imax + toler ) ) break;
2227
2228          idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxI;
2229          idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxH;
2230          idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxG;
2231          placeTop( basePt1, basePt2, basePt3, apexPt2, idxArr, ↪
                ↩ dlist );
```

```
2232
2233          for( idxM = 0; idxM < 2; ++idxM )
2234          {
2235            apexPt2.z = pow( -1.0L, idxM )* apexPt2.z; // reflect about XY plane
2236            counter += 1;
2237
2238            bridgeDist = getDistance( apexPt1, apexPt2 );
2239            if( bridgeDist != bridgeDist ) getchar(); // cout << p4 << '\t' << p5 << endl;
2240            if( ( bridgeDist < ( dlist[ 0 ] - bridgeDist* toler ) ) or
2241              ( bridgeDist > ( dlist[ dlist.size() - 1 ] + bridgeDist* toler ) ) ) continue;
2242            idxJ = closestDist( bridgeDist, dlist );
2243
2244            if( idxJ == idxA and idxJ == idxB and idxJ == idxC and
2245                idxJ == idxD and idxJ == idxE and idxJ == idxF and
2246                idxJ == idxG and idxJ == idxH and idxJ == idxI )
2247            {
2248              continue;
2249            }
2250            if( usedDist[ idxJ ] ) continue;
2251
2252            fracError = fabs( dlist[ idxJ ] - bridgeDist )/ bridgeDist;
2253            if( fracError < toler )
2254            {
2255              atoms.push_back( apexPt2 );
2256
2257              cout << endl << "Point found!!!" << endl;
2258              cout << bridgeDist << '\t' << dlist[ idxJ ] << '\t'
2259                   << fabs( bridgeDist - dlist[ idxJ ] ) << endl;
2260              cout << "bridgeDist: " << bridgeDist << endl;
2261              cout << "counter: " << counter << endl;
2262
2263              cout << "Point: " << atoms.size() << '\t'
2264                   << atoms[ atoms.size() - 1 ] << endl;
2265
2266              usedDist[ idxG ] = usedDist[ idxH ] = usedDist[ idxI ]
2267                               = usedDist[ idxJ ] = true;
2268              if( atoms.size() == targetSize )
2269              {
2270                cout << "buildup counter: "<< counter << endl;
```

```
2271                 return true;
2272               }
2273
2274             }
2275           } // m loop
2276         } // i loop
2277       } // h loop
2278     } // g loop
2279
2280     cout << "counter: " << counter << endl;
2281     return false;
2282 }
2283
2284
2285 bool Structure::home3D( int distIdx )
2286 {
2287     // Orient the structure in a unique manner so that it becomes ↪
        ↩ easy to check
2288     // if two or more structure are identical to one another or not.
2289
2290     long double minDist = 1e6;
2291     int idx1 = 0, idx2 = 1;
2292     long double dist = 1e6, err, minErr;
2293     minErr = fabs( targetDL[ distIdx ] - getDistance( atoms[ idx1 ↪
        ↩ ], atoms[ idx2 ] ) );
2294
2295     // Find the correct bond in the structure
2296     for( int i = 0; i < atoms.size(); ++i )
2297     {
2298       for( int j = i + 1; j < atoms.size(); ++j )
2299       {
2300         dist = getDistance( atoms[ i ], atoms[ j ] );
2301         err = fabs( targetDL[ distIdx ] - dist );
2302         if( err < minErr )
2303         {
2304           minErr = err;
2305           idx1 = i;
2306           idx2 = j;
2307         }
2308       }
2309     }
2310     // cout << "idx1, idx2: " << idx1 << '\t' << idx2 << endl;
2311
2312     // Locate the apex point of the base triangle
2313     long double minDist1 = 1e6, minDist2 = 1e6;
```

```
2314    long double dist1, dist2;
2315    int minIdx1, minIdx2;
2316
2317    for( int i = 0; i < atoms.size(); ++i )
2318    {
2319      if( i == idx1 or i == idx2 ) continue;
2320
2321      dist1 = getDistance( atoms[ i ], atoms[ idx1 ] );
2322      if( dist1 < minDist1 )
2323      {
2324        minDist1 = dist1;
2325        minIdx1 = i;
2326      }
2327
2328      dist2 = getDistance( atoms[ i ], atoms[ idx2 ] );
2329      if( dist2 < minDist2 )
2330      {
2331        minDist2 = dist2;
2332        minIdx2 = i;
2333      }
2334    }
2335
2336    int idx3 = minIdx1;
2337    if( minDist1 > minDist2 )
2338    {
2339      idx3 = minIdx2;
2340      swap( idx1, idx2 );
2341    }
2342
2343    // Correctly place the base triangle
2344    translate( -atoms[ idx1 ].x, -atoms[ idx1 ].y, -atoms[ idx1 ↪
       ↩ ].z  );
2345    // find angle and rotate
2346    Point pt2;
2347    pt2.x = 1; pt2.y = 0; pt2.z = 0;
2348    long double angle = getAngle( atoms[ idx2 ], pt2 );
2349    Point axis = getAxis( atoms[ idx2 ], pt2 );
2350    // cout << "angle, axis: " << angle << ',' << axis.x << ',' ↪
       ↩ << axis.y << ','
2351    //      << axis.z << endl;
2352    rotate( angle, axis.x, axis.y, axis.z );
2353
2354    if( atoms[ idx3 ].y < 0 )
2355    {
2356      reflect( "X" );
```

```
2357    }
2358
2359    // Sort the points so that there is some unique order
2360    sort( atoms.begin(), atoms.end() );
2361
2362    int idxApex = 2;
2363    long double angle3 = atan2( atoms[ idxApex ].z, atoms[ ↪
           ↩ idxApex ].y );
2364    // cout << "-angle3, axis: " << -angle3 << ',' << "1" << ','
2365    //                          << "0" << ',' << "0" << endl;
2366    rotate( -angle3, 1, 0, 0 );
2367
2368    if( atoms[ 3 ].z < 0 )
2369    {
2370      reflect( "Z" );
2371    }
2372
2373    return true;
2374  }
2375
2376
2377  bool Structure::doBuildup3Dv2( int pt1, int pt2, int pt3 )
2378  {
2379    bool successFlag = false;
2380    freeDL = targetDL;
2381    // updateUsedDists();
2382    // vector<long double> freeDL;
2383
2384    // for( int i = 0; i < targetDL.size(); ++i )
2385    // {
2386    //   if( usedDist[ i ] == false )
2387    //   {
2388    //     freeDL.push_back( targetDL[ i ] );
2389    //   }
2390    // }
2391
2392    vector<int> idxArr( 6, 0 );
2393    idxArr[ 0 ] = closestDist( getDistance( atoms[ pt1 ], atoms[ ↪
           ↩ pt2 ] ),
2394                               targetDL );
2395    idxArr[ 1 ] = closestDist( getDistance( atoms[ pt1 ], atoms[ ↪
           ↩ pt3 ] ),
2396                               targetDL );
2397    idxArr[ 2 ] = closestDist( getDistance( atoms[ pt2 ], atoms[ ↪
           ↩ pt3 ] ),
```

```
2398                                    targetDL );
2399    // idxArr[ 0 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
        ↩ 1 ] ),
2400    //                                 freeDL );
2401    // idxArr[ 1 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
        ↩ 2 ] ),
2402    //                                 freeDL );
2403    // idxArr[ 2 ] = closestDist( getDistance( atoms[ 1 ], atoms[ ↪
        ↩ 2 ] ),
2404    //                                 freeDL );
2405    // idxArr[ 3 ] = closestDist( getDistance( atoms[ 0 ], atoms[ ↪
        ↩ 3 ] ),
2406    //                                 freeDL );
2407    // idxArr[ 4 ] = closestDist( getDistance( atoms[ 1 ], atoms[ ↪
        ↩ 3 ] ),
2408    //                                 freeDL );
2409    // idxArr[ 5 ] = closestDist( getDistance( atoms[ 2 ], atoms[ ↪
        ↩ 3 ] ),
2410    //                                 freeDL );
2411    cout << "idxArr: " << idxArr[ 0 ] << ',' << idxArr[ 1 ] << ','
2412                       << idxArr[ 2 ] << ',' << idxArr[ 3 ] << ','
2413                       << idxArr[ 4 ] << ',' << idxArr[ 5 ] << endl;
2414
2415    // 10 bonds in the tetrahedron
2416    int idxA = idxArr[ 0 ], idxB = idxArr[ 1 ], idxC = idxArr[ 2 ],
2417        idxD = idxArr[ 3 ], idxE = idxArr[ 4 ], idxF = idxArr[ 5 ],
2418        idxG, idxH, idxI, idxJ, idxM;
2419    int numTestPts = 0; // number of tetrahedra
2420
2421    // Point basePt1 = atoms[ 0 ], basePt2 = atoms[ 1 ], basePt3 ↪
        ↩ = atoms[ 2 ],
2422    //       apexPt1 = atoms[ 3 ], apexPt2;
2423    Point basePt1 = atoms[ pt1 ], basePt2 = atoms[ pt2 ], basePt3 ↪
        ↩ = atoms[ pt3 ],
2424          apexPt1, apexPt2;
2425
2426    long double imin, imax;
2427    long double triToler = 0.1;
2428    long double distA = getDistance( basePt1, basePt2 ),
2429                distG, distH, distI, error;
2430
2431    for( idxG = 0; idxG < freeDL.size(); ++idxG )
2432    {
2433      distG = freeDL[ idxG ];
2434
```

```
2435      for( idxH = 0; idxH < freeDL.size(); ++idxH )
2436      {
2437        distH = freeDL[ idxH ];
2438        if( distA + distH + triToler < distG ) continue; // triangle inequality
2439        if( distA + distG + triToler < distH ) break; // triangle inequality
2440
2441        placeApex( apexPt2, idxA, idxG, idxH, freeDL );
2442
2443        imin = getDistance( basePt3, apexPt2 );
2444        apexPt2.y = - apexPt2.y;
2445        imax = getDistance( basePt3, apexPt2 );
2446        // cout << "bp3, ap2: " << basePt3 << ',' << apexPt2 << endl;
2447
2448        for( idxI = 0; idxI < freeDL.size(); ++idxI )
2449        {
2450          distI = freeDL[ idxI ];
2451          if( distI < ( imin - triToler ) ) continue;
2452          if( distI > ( imax + triToler ) ) break;
2453
2454          idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxI;
2455          idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxH;
2456          idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxG;
2457          placeTop( basePt1, basePt2, basePt3, apexPt2, idxArr, freeDL );
2458
2459          for( idxM = 0; idxM < 2; ++idxM )
2460          {
2461            apexPt2.z = pow( -1.0L, idxM )* apexPt2.z; // reflect about XY plane
2462
2463            apexPt2.cost = error = getPtCost( apexPt2 );
2464
2465            if( error < 0.2 )
2466            {
2467              // cout << "adding pt,cost: " << apexPt2 << ',' << error << endl;
2468              updatePool( apexPt2 );
2469              ++numTestPts;
2470              // FIX ME
2471              // return false;
2472            }
2473          } // m loop
```

```
2474        } // i loop
2475      } // h loop
2476    } // g loop
2477
2478    cout << "pool.size: " << pool.size() << endl;
2479    cout << "Number of points tested: " << numTestPts << endl;
2480    cout << "End of doBuildup3Dv2" << endl;
2481    return false;
2482  }
2483
2484
2485  bool Structure::findCoreMPI( int windowStart )
2486  {
2487    // Find a core made of 4 points by iterating over all triangle
2488    // combinations. Also, the function to do the buildup is ↪
         ↩ called after
2489    // we find a core because it is more convenient this way.
2490
2491    int idxA = 0, idxB, idxC, idxD, idxE, idxF; // indices for ↪
         ↩ the bonds
2492    int idxM, idxN; // indices for the orientation of the triangle
2493    vector<int> idxArr( 6, -1 );
2494    int inc = 6; // width of the bond window
2495    int winStart = windowStart, winStop = windowStart + inc; // ↪
         ↩ indices for the window
2496    vector<long double> dlist = targetDL;
2497
2498    long double bridgeDist = 0.0;
2499    int bridgeIdx = 0;
2500    int bridgeCount = 0; // count the number of bridge bond checks
2501    long double fracError = 1e6;
2502
2503    Point basePt1, basePt2,
2504          apexPt1, apexPt2;
2505    basePt1.x = 0; basePt1.y = 0;
2506    basePt2.x = dlist[ idxA ]; basePt2.y = 0;
2507    cout << "basePt1: " << basePt1.x << " " << basePt1.y << endl;
2508    cout << "basePt2: " << basePt2.x << " " << basePt2.y << endl;
2509    cout << "Bond window: ";
2510
2511    while( true )
2512    {
2513      cerr << " -> " << winStop;
2514
2515      for( idxB = idxA + 1; idxB < winStop; ++idxB )
```

```
2516        {
2517          for( idxC = idxB + 1; idxC < winStop; ++idxC )
2518          {
2519            if( dlist[ idxA ] + dlist[ idxB ] + toler < dlist[ idxC ↪
                  ↩ ] )
2520            {
2521              break;
2522            }
2523            placeApex( apexPt1, idxA, idxB, idxC, dlist );
2524
2525            for( idxD = idxA + 1; idxD < winStop; ++idxD )
2526            {
2527              if ( idxD == idxB or idxD == idxC )
2528              {
2529                continue;
2530              }
2531              for( idxE = idxD + 1; idxE < winStop; ++idxE )
2532              {
2533                if( idxB < winStart and idxC < winStart and
2534                    idxD < winStart and idxE < winStart )
2535                {
2536                  continue;
2537                }
2538
2539                if( ( idxD < idxB and idxE < idxC ) or
2540                    ( idxE > idxC and idxD < idxB ) )
2541                {
2542                  continue;
2543                }
2544
2545                if( idxE == idxB or idxE == idxC )
2546                {
2547                  continue;
2548                }
2549
2550                if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ ↪
                      ↩ idxE ] )
2551                {
2552                  break;
2553                }
2554
2555                placeApex( apexPt2, idxA, idxD, idxE, dlist );
2556
2557                for( idxM = 0; idxM < 2; ++idxM )
2558                {
```

```
2559            // cout << "idxM: " << idxM << endl;
2560            if( idxM == 1 )
2561            {
2562              apexPt2.x = dlist[ idxA ] - apexPt2.x;
2563            }
2564
2565            for( idxN = 0; idxN < 2; ++idxN )
2566            {
2567              // cout << "idxN: " << idxN << endl;
2568              if( idxN == 1 )
2569              {
2570                apexPt2.y = - apexPt2.y;
2571              }
2572
2573              bridgeDist = getDistance( apexPt1, apexPt2 );
2574              bridgeCount += 1; // count number of bridge checks
2575
2576              bridgeIdx = closestDist( bridgeDist, dlist );
2577              if( bridgeIdx == idxA or bridgeIdx == idxB or
2578                  bridgeIdx == idxC or bridgeIdx == idxD or
2579                  bridgeIdx == idxE )
2580              {
2581                // Make sure that the bridge bond is not the ↪
                     ↩ same as any of
2582                // the distances in use.
2583                continue;
2584              }
2585
2586              fracError = fabs( dlist[ bridgeIdx ] - ↪
                   ↩ bridgeDist )/ bridgeDist;
2587              // cout << "fracError: " << fracError << endl;
2588
2589              if( fabs( apexPt2.y ) < 0.5 )
2590              {
2591                // Skinny triangles have been found to have a ↪
                     ↩ large error,
2592                // hence reducing their error "by hand" so ↪
                     ↩ that we don't
2593                // miss out on them.
2594                fracError /= 1000;
2595              }
2596
2597              if( fracError < toler )
2598              {
2599                idxArr[ 0 ] = idxA; idxArr[ 1 ] = idxB;
```

```
2600                idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxD;
2601                idxArr[ 4 ] = idxE; idxArr[ 5 ] = bridgeIdx;
2602
2603                cout << endl;
2604                if( testCore( idxArr, idxM, idxN ) )
2605                {
2606                  // atoms.push_back( basePt1 );
2607                  // atoms.push_back( basePt2 );
2608                  // atoms.push_back( apexPt1 );
2609                  // atoms.push_back( apexPt2 );
2610
2611                  cout << basePt1 << endl;
2612                  cout << basePt2 << endl;
2613                  cout << apexPt1 << endl;
2614                  cout << apexPt2 << endl;
2615
2616                  // for( int i = 0; i < atoms.size(); ++i )
2617                  // {
2618                  //   cout << "Point: " << i + 1 << '\t' << ↪
                        ↩ atoms[ i ] << endl;
2619                  // }
2620
2621                  // usedDist[ idxA ] = usedDist[ idxB ] = true;
2622                  // usedDist[ idxC ] = usedDist[ idxD ] = true;
2623                  // usedDist[ idxE ] = usedDist[ bridgeIdx ] ↪
                        ↩ = true;
2624
2625                  // updateCurrDL();
2626                  // return true;
2627
2628                  // Attempt buildup to get the remaining ↪
                        ↩ points.
2629                  // doBuildup();
2630
2631                  // if( atoms.size() >= min( 8, targetSize ) )
2632                  // {
2633                  //   // If buildup was able to add 4 more ↪
                        ↩ points then with
2634                  //   // high probability, we have the right ↪
                        ↩ structure.
2635                  //   return true;
2636                  // }
2637                  // else
2638                  // {
2639                  //   // If buildup could not even 4 points ↪
```

```
                        ↩ then with a very
2640                    //    // high probability, we have the wrong ↪
                        ↩ structure. Start
2641                    //    // over and find the next core.
2642                    //    cout << "No buildup, bad core. "
2643                    //             "Finding the next core ... " ↪
                        ↩ << endl;
2644                    //    atoms.clear();
2645                    //    updateCurrDL();
2646                    //    cout << "Bond window: ";
2647                    // }
2648                  }
2649                }
2650
2651              } // n loop
2652            } // m loop
2653          } // idxE loop
2654        } // idxD loop
2655      } // idxC loop
2656    } // idxB loop
2657
2658    // When idxB hits the window edge increment it.
2659    // if ( idxB == winStop )
2660    // {
2661    //   winStart = winStop;
2662    //   winStop += inc;
2663    //
2664    //   if( winStart == dlist.size() )
2665    //   {
2666    //     cout << endl;
2667    //     break;
2668    //   }
2669    //
2670    //   if( winStop > dlist.size() )
2671    //   {
2672    //     winStop = dlist.size();
2673    //   }
2674    // }
2675
2676  } // while( true ) loop
2677
2678  return false;
2679 }
2680
2681
```

```
2682 bool Structure::findCore3D( int baseIdx )
2683 {
2684   // Function to get timings runs for different choices of base bond
2685
2686   bool successFlag = false;
2687
2688   // 10 bonds in the tetrahedron
2689   int idxA = baseIdx, idxB, idxC, idxD, idxE, idxF, idxG, idxH,
         idxI, idxJ, idxM;
2690
2691   int invC = 0; // # invalid cores
2692   vector<long double> dlist = targetDL;
2693   cout << "targetDL size: " << targetDL.size() << endl;
2694
2695   int inc = 10,
2696       winStart = max( baseIdx - inc/2, 0),
2697       winStop = min( winStart + inc/2, int( dlist.size() ) - 1
           ); // windowing on
2698
2699   int bridgeIdx;
2700   long double counter = 0; // number of tetrahedra
2701   long double bridgeDist = 0;
2702   long double fmin, fmax, imin, imax, fracError;
2703
2704   Point basePt1, basePt2, basePt3,
2705         apexPt1, apexPt2;
2706   basePt1.x = 0; basePt1.y = 0; basePt1.z = 0;
2707   basePt2.x = dlist[ idxA ]; basePt2.y = 0; basePt2.z = 0;
2708   cout << "basePt1: " << basePt1.x << " " << basePt1.y << " "
         << basePt1.z
2709        << endl;
2710   cout << "basePt2: " << basePt2.x << " " << basePt2.y << " "
         << basePt2.z
2711        << endl;
2712   cout << "Bond window: ";
2713
2714   int countC = 0; // count number of cores
2715   vector<int> idxArr( 6, 0 );
2716   while( true ) // window loop
2717   {
2718     // Break after the entire distance list is exhausted
2719     if( winStart <= 0 and winStop >= dlist.size() - 1 )
2720     {
2721       break;
```

```
2722        }
2723
2724        // Make sure to check if window edges are legit
2725        if( winStart < 0 )
2726        {
2727          winStop += -winStart;
2728          winStart = 0;
2729        }
2730
2731        if( winStop >= dlist.size() )
2732        {
2733          winStart -= winStop - dlist.size() + 1;
2734          winStop = dlist.size() -1;
2735
2736          if( winStart < 0 )
2737          {
2738            winStop += -winStart;
2739            winStart = 0;
2740          }
2741        }
2742
2743        cerr << "->" << winStop;
2744
2745        // base triangle
2746        for( idxB = 0; idxB < winStop; ++idxB )
2747        {
2748          // cout << endl << "idxB:" << idxB << ", "; // endl;
2749          // cout << "idxC: ";
2750          for( idxC = idxB + 1; idxC < winStop; ++idxC )
2751          {
2752            // cout << "_" << idxC << "_"; // << endl;
2753            // NOTE: c>b so we have to check only 1 triangle ↪
                  ↩ inequality
2754            if( dlist[ idxA ] + dlist[ idxB ] + toler < dlist[ idxC ↪
                  ↩ ] ) break;
2755            if( dlist[ idxC ] + dlist[ idxB ] + toler < dlist[ idxA ↪
                  ↩ ] ) continue;
2756
2757            placeApex( basePt3, idxA, idxB, idxC, dlist );
2758
2759            for( idxD = 0; idxD < winStop; ++idxD )
2760            {
2761              if( idxD == idxB  or idxD == idxC ) continue;
2762              if( idxD < idxB ) continue;
2763              for( idxE = 0; idxE < winStop; ++idxE )
```

```
2764            {
2765              if( idxE < idxB ) continue;
2766              if( idxE == idxB or idxE == idxC or idxE == idxD ) ↪
                    ↩ continue;
2767              // triangle inequalities
2768              if( dlist[ idxA ] + dlist[ idxE ] + toler < dlist[ ↪
                    ↩ idxD ] ) continue;
2769              if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ ↪
                    ↩ idxE ] ) break;
2770              if( dlist[ idxE ] + dlist[ idxD ] + toler < dlist[ ↪
                    ↩ idxA ] ) continue;
2771
2772              placeApex( apexPt1, idxA, idxD, idxE, dlist );
2773
2774              fmin = getDistance( basePt3, apexPt1 );
2775              apexPt1.y = - apexPt1.y;
2776              fmax = getDistance( basePt3, apexPt1 );
2777
2778              for( idxF = 0; idxF < dlist.size(); ++idxF )
2779              {
2780                if( dlist[idxF] < ( fmin - toler ) ) continue;
2781                if( dlist[idxF] > ( fmax + toler ) ) break;
2782                if( idxF == idxB or idxF == idxC or idxF == idxD ↪
                      ↩ or idxF == idxE ) continue;
2783
2784                idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxF;
2785                idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxE;
2786                idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxD;
2787                placeTop( basePt1, basePt2, basePt3, apexPt1, ↪
                      ↩ idxArr, dlist );
2788
2789                for( idxG = 0; idxG < winStop; ++idxG )
2790                {
2791                  if( idxG < idxB ) continue;
2792                  if( idxG == idxB or idxG == idxC or idxG == ↪
                        ↩ idxD or
2793                    idxG == idxE or idxG == idxF ) continue;
2794                  if( idxG < idxD ) continue;
2795                  for( idxH = 0; idxH < winStop; ++idxH )
2796                  {
2797                    if( idxH < idxB ) continue;
2798                    if( idxH ==idxB or idxH == idxC or idxH == ↪
                          ↩ idxD or
2799                      idxH == idxE or idxH == idxF or idxH == ↪
                            ↩ idxG ) continue;
```

```
2800            if( dlist[ idxA ] + dlist[ idxH ] + toler < dlist[ idxG ] ) continue; // triangle inequality
2801            if( dlist[ idxA ] + dlist[ idxG ] + toler < dlist[ idxH ] ) break; // triangle inequality
2802            if( dlist[ idxH ] + dlist[ idxG ] + toler < dlist[ idxA ] ) continue; // triangle inequality
2803
2804            placeApex( apexPt2, idxA, idxG, idxH, dlist );
2805
2806            imin = getDistance( basePt3, apexPt2 );
2807            apexPt2.y = - apexPt2.y;
2808            imax = getDistance( basePt3, apexPt2 );
2809            // cout << "bp3, ap2: " << basePt3 << ',' << apexPt2 << endl;
2810
2811            for( idxI = 0; idxI < dlist.size(); ++idxI )
2812            {
2813              if( idxI == idxB or idxI == idxC or idxI == idxD or
2814                  idxI == idxE or idxI == idxF or idxI == idxG or
2815                  idxI == idxH ) continue;
2816              // correct windowing
2817              if( idxH < winStart and idxG < winStart and
2818                  idxE < winStart and idxD < winStart and
2819                  idxC < winStart and idxB < winStart) continue;
2820
2821              if( dlist[ idxI ] < ( imin - toler ) ) continue;
2822              if( dlist[ idxI ] > ( imax + toler ) ) break;
2823
2824              // cout << "abcdefghi: " << idxA << ',' << idxB << ','
2825              //                                 << idxC << ',' << idxD << ','
2826              //                                 << idxE << ',' << idxF << ','
2827              //                                 << idxG << ',' << idxH << ','
2828              //                                 << idxI << endl;
2829              // cout << "dlist-I, imin, imax: " <<
```

```
      ↩ dlist[ idxI ] << ',' << imin << ',' << ↪
      ↩ imax << endl << endl;
2830
2831                idxArr[ 0 ] = idxA; idxArr[ 5 ] = idxI;
2832                idxArr[ 1 ] = idxB; idxArr[ 4 ] = idxH;
2833                idxArr[ 2 ] = idxC; idxArr[ 3 ] = idxG;
2834                placeTop( basePt1, basePt2, basePt3, ↪
                      ↩ apexPt2, idxArr, dlist );
2835
2836                for( idxM = 0; idxM < 2; ++idxM )
2837                {
2838                  apexPt2.z = pow( -1.0L, idxM )* ↪
                        ↩ apexPt2.z; // reflect about XY plane
2839
2840                  bridgeDist = getDistance( apexPt1, ↪
                        ↩ apexPt2 );
2841                  counter += 1;
2842
2843                  if( bridgeDist != bridgeDist ) getchar(); ↪
                        ↩ // cout << p4 << '\t' << p5 << endl;
2844                  if( ( bridgeDist < ( dlist[ 0 ] - ↪
                        ↩ bridgeDist* toler ) ) or
2845                    ( bridgeDist > ( dlist[ dlist.size() ↪
                          ↩ - 1 ] + bridgeDist* toler ) ) ) ↪
                          ↩ continue;
2846                  idxJ = closestDist( bridgeDist, dlist );
2847
2848                  if( idxJ == idxA and idxJ == idxB and ↪
                        ↩ idxJ == idxC and
2849                    idxJ == idxD and idxJ == idxE and ↪
                          ↩ idxJ == idxF and
2850                    idxJ == idxG and idxJ == idxH and ↪
                          ↩ idxJ == idxI )
2851                  {
2852                    continue;
2853                  }
2854
2855                  fracError = fabs( dlist[ idxJ ] - ↪
                        ↩ bridgeDist )/ bridgeDist;
2856                  if( fracError < toler )
2857                  {
2858                    atoms.push_back( basePt1 );
2859                    atoms.push_back( basePt2 );
2860                    atoms.push_back( basePt3 );
2861
```

```
2862                atoms.push_back( apexPt1 );
2863                atoms.push_back( apexPt2 );
2864
2865                cout << endl << "3D core found!!!" << endl;
2866                cout << bridgeDist << '\t' << dlist[ idxJ ] << '\t'
2867                     << fabs( bridgeDist - dlist[ idxJ ] ) << endl;
2868                cout << "bridgeDist: " << bridgeDist << endl;
2869                cout << "core finder counter: " << counter << endl;
2870
2871                for( int i = 0; i < atoms.size(); ++i )
2872                {
2873                  cout << "Point: " << i + 1 << '\t' << atoms[ i ] << endl;
2874                }
2875
2876                doBuildup3D( idxArr );
2877
2878                if( atoms.size() >= min( 8, targetSize ) )
2879                {
2880                  // If buildup was able to add 4 more points then with
2881                  // high probability, we have the right structure.
2882                  cout << " atoms size: " << atoms.size() << endl;
2883                  return true;
2884                }
2885                else
2886                {
2887                  // If buildup could not even 4 points then with a very
2888                  // high probability, we have the wrong structure. Start
2889                  // over and find the next core.
2890                  cout << "No buildup, bad core. "
2891                          "Finding the next core ... " << endl;
2892                  atoms.clear();
2893                  updateCurrDL();
```

```
2894                                    // cout << "Bond window: ";
2895                                  }
2896
2897                                  // return true;
2898                                }
2899                              } // m loop
2900                            } // i loop
2901                          } // h loop
2902                        } // g loop
2903                      } // f loop
2904                    } // e loop
2905                  } // d loop
2906                } // c loop
2907              } // b loop
2908
2909              cout << "counter: "<< counter << endl;
2910              winStart = winStop; winStop += inc;
2911              if( winStart == dlist.size() ) break;
2912
2913              if( winStop > dlist.size() )
2914              {
2915                winStop = dlist.size();
2916              }
2917
2918              cout << "searching in a "<< winStop << " bond window" << endl;
2919            } // window while loop
2920
2921            return successFlag;
2922          }
2923
2924
2925          long double Structure::feasibleTetra( int baseIdx )
2926          {
2927            // Function to get number of feasible tetrahedra's for a
                 given base bond
2928            long double numTet = 0;
2929            long double numTri = 0;
2930            bool successFlag = false;
2931
2932            // 10 bonds in the tetrahedron
2933            int idxA = baseIdx, idxB, idxC, idxD, idxE, idxF, idxG, idxH,
                 idxI, idxJ, idxM;
2934
2935            int invC = 0; // # invalid cores
2936            vector<long double> dlist = targetDL;
```

```
2937    cout << "targetDL size: " << targetDL.size() << endl;
2938
2939    int inc = 10,
2940        // winStart = max( baseIdx - inc/2, 0),
2941        // winStop = min( winStart + inc, int( dlist.size() ) - 1 ↪
                ↩ ); // windowing on
2942        winStart = 0,
2943        winStop = dlist.size() - 1 ; // windowing on
2944
2945    int bridgeIdx;
2946    int counter = 0; // number of tetrahedra
2947    long double bridgeDist = 0;
2948    long double fmin, fmax, imin, imax, fracError;
2949
2950    Point basePt1, basePt2, basePt3,
2951          apexPt1, apexPt2;
2952    basePt1.x = 0; basePt1.y = 0; basePt1.z = 0;
2953    basePt2.x = dlist[ idxA ]; basePt2.y = 0; basePt2.z = 0;
2954    cout << "basePt1: " << basePt1.x << " " << basePt1.y << " " ↪
           ↩ << basePt1.z
2955         << endl;
2956    cout << "basePt2: " << basePt2.x << " " << basePt2.y << " " ↪
           ↩ << basePt2.z
2957         << endl;
2958
2959    int countC = 0; // count number of cores
2960    vector<int> idxArr( 6, 0 );
2961    while( true ) // window loop
2962    {
2963      for( idxB = 0; idxB < winStop; ++idxB )
2964      {
2965        // cout << endl << "idxB:" << idxB << ", "; // endl;
2966        // cout << "idxC: ";
2967        for( idxC = idxB + 1; idxC < winStop; ++idxC )
2968        {
2969          // cout << "_" << idxC << "_"; // << endl;
2970          // NOTE: c>b so we have to check only 1 triangle ↪
                  ↩ inequality
2971          if( dlist[ idxA ] + dlist[ idxB ] + toler < dlist[ idxC ↪
                  ↩ ] ) break;
2972
2973          placeApex( basePt3, idxA, idxB, idxC, dlist );
2974          numTri += 1;
2975
2976          for( idxD = 1; idxD < winStop; ++idxD )
```

```
2977            {
2978              if( idxD == idxB  or idxD == idxC ) continue;
2979              if( idxD < idxB ) continue;
2980              for( idxE = 1; idxE < winStop; ++idxE )
2981              {
2982                if( idxE < idxB ) continue;
2983                if( idxE == idxB or idxE == idxC or idxE == idxD ) ↪
                      ↩ continue;
2984                // triangle inequalities
2985                if( dlist[ idxA ] + dlist[ idxE ] + toler < dlist[ ↪
                      ↩ idxD ] ) continue;
2986                if( dlist[ idxA ] + dlist[ idxD ] + toler < dlist[ ↪
                      ↩ idxE ] ) break;
2987
2988                placeApex( apexPt1, idxA, idxD, idxE, dlist );
2989                numTri += 1;
2990
2991                fmin = getDistance( basePt3, apexPt1 );
2992                apexPt1.y = - apexPt1.y;
2993                fmax = getDistance( basePt3, apexPt1 );
2994
2995                for( idxF = 1; idxF < dlist.size(); ++idxF )
2996                {
2997                  if( dlist[idxF] < ( fmin - toler ) ) continue;
2998                  if( dlist[idxF] > ( fmax + toler ) ) break;
2999                  if( idxF == idxB or idxF == idxC or idxF == idxD ↪
                        ↩ or idxF == idxE ) continue;
3000
3001                  // placeTop( basePt1, basePt2, basePt3, apexPt1, ↪
                        ↩ idxArr, dlist );
3002
3003                  numTet += 1;
3004                  continue;
3005                } // f loop
3006              } // e loop
3007            } // d loop
3008          } // c loop
3009        } // b loop
3010
3011        cout << "numTri: " << numTri << endl;
3012        return numTet;
3013      } // window while loop
3014    }
3015
3016
```

```
3017 bool Structure::reconstruct3( int baseIdx )
3018 {
3019   // Reconstruct the structure by first finding the core and ↪
       ↩ then doing
3020   // buildup(if needed).
3021
3022   if( atoms.size() == 0 )
3023   {
3024     if( dim == 2 )
3025     {
3026       bool findCoreFlag = findCore();
3027       cout << "Core found? " << boolalpha << findCoreFlag << endl;
3028     }
3029     else if( dim == 3 )
3030     {
3031       cout << "findCore3D, " << baseIdx << endl;
3032       findCore3D( baseIdx );
3033     }
3034   }
3035   else if( dim == 3 )
3036   {
3037
3038   }
3039
3040   if( atoms.size() == targetSize )
3041   {
3042     return true;
3043   }
3044   else
3045   {
3046     return false;
3047   }
3048 }
```

```
1  // Test2D.cpp: File with all the test functions for Tribond 2D.
2  //
3  #include <ctime>
4  #include <string>
5  #include "Tribond.h"
6  using namespace std;
7
8
9  int processArgs( int argc, char** argv, int& N, string& file, ↪
       ↩ int& rngSeed )
10 {
```

```
11    // Process input from user to get problem parameters.
12    if( argc == 4 )
13    {
14      N = atoi( argv[1] );
15      file = argv[2];
16      rngSeed = atoi( argv[3] );
17    }
18    else
19    {
20      cout << "Usage: " << "./Test2d N(>4) rand2 rngSeed" << endl;
21      cout << "  N: number of sites in the random point set" << endl;
22      cout << "  rand2: command to use a random 2D point set" << endl;
23      cout << "  rngSeed: seed for the random number generator" << endl;
24      exit(0);
25    }
26  }
27
28
29  int test1( int DIM, int N, string file, int rngSeed )
30  {
31    // Initialize the random number generator.
32    if (rngSeed <= -1)
33    {
34      rngSeed = time( NULL );
35    }
36    srand( rngSeed );
37
38    // Setup target structure and attempt reconstruction.
39    Structure targetStru( DIM, N, file );
40    Structure testStru( DIM, N, targetStru.currDL );
41
42    testStru.reconstruct();
43
44    // Compare reconstructed structure with the target.
45    targetStru.home();
46    testStru.home();
47    compareStru( testStru, targetStru );
48    // testStru.printDLtoFile( "distanceList.txt" );
49
50    return 0;
51  }
52
```

```
53
54 int main( int argc, char** argv )
55 {
56   int DIM = 2, N, rngSeed;
57   string file;
58   processArgs( argc, argv, N, file, rngSeed );
59
60   test1( DIM, N, file, rngSeed );
61   return 0;
62 }
```

```
1  // Test3D.cpp: File with all the test functions for Tribond 3D.
2  //
3  #include <ctime>
4  #include <string>
5  #include "Tribond.h"
6  using namespace std;
7
8
9  int processArgs( int argc, char** argv, int& N, string& file, ↪
       ↩ int& rngSeed )
10 {
11   // Process input from user to get problem parameters.
12   if( argc == 4 )
13   {
14     N = atoi( argv[1] );
15     file = argv[2];
16     rngSeed = atoi( argv[3] );
17   }
18   else
19   {
20     cout << "Usage: " << "./Test2d N(>4) rand2 rngSeed" << endl;
21     cout << "  N: number of sites in the random point set" << ↪
         ↩ endl;
22     cout << "  rand2: command to use a random 2D point set" << ↪
         ↩ endl;
23     cout << "  rngSeed: seed for the random number generator" ↪
         ↩ << endl;
24     exit(0);
25   }
26 }
27
28
29 int test1( int DIM, int N, string file, int rngSeed )
30 {
```

```
31    // Initialize the random number generator.
32
33    // FIXME hard coding the rnd seed
34    int idx = rngSeed;
35    // rngSeed = 1;
36
37    if (rngSeed <= -1)
38    {
39      rngSeed = time( NULL );
40    }
41    srand( rngSeed );
42
43    for( int i = 0; i <= 0; ++i )
44    {
45      // Setup target structure and attempt reconstruction.
46      // Structure targetStru( DIM, N, file );
47      Structure targetStru( N, file );
48      targetStru.home3D( 0 );
49      targetStru.print();
50
51      Structure testStru( DIM, N, targetStru.currDL );
52      testStru.atoms = targetStru.getCore( 5 );
53
54      int baseIdx = idx* (testStru.targetDL.size() - 1)/ 10;
55      cout << "baseIdx: " << baseIdx << endl;
56      // testStru.reconstruct3( baseIdx );
57      testStru.reconstruct();
58
59      // Compare reconstructed structure with the target.
60      testStru.home3D( 0 );
61      compareStru( testStru, targetStru );
62      print2structures( testStru.atoms, targetStru.atoms );
63      compareStru( testStru, targetStru );
64    }
65
66    // testStru.printDLtoFile( "distanceList.txt" );
67    return 0;
68  }
69
70
71  int test2( int DIM, int N, string file, int rngSeed )
72  {
73    // Initialize the random number generator.
74    if (rngSeed <= -1)
75    {
```

```
76      rngSeed = time( NULL );
77    }
78    srand( rngSeed );
79
80    // Setup target structure and attempt reconstruction.
81    Structure targetStru( DIM, N, file );
82    // Structure targetStru( N, file );
83    // targetStru.home3D();
84
85    Structure testStru( DIM, N, targetStru.currDL );
86    // testStru.atoms = targetStru.getCore( 5 );
87    long double numTetra = 0;
88    for( int i = 0; i <= 10; ++i )
89    {
90      numTetra = testStru.feasibleTetra( i* ↪
          ↩ (testStru.targetDL.size() - 1) /10 );
91      cout << "i: " << i << ", " << "numTetra: " << numTetra << ↪
          ↩ endl;
92
93    }
94
95    return 0;
96  }
97
98
99  int main( int argc, char** argv )
100 {
101   int DIM = 3, N, rngSeed;
102   string file;
103   processArgs( argc, argv, N, file, rngSeed );
104
105   test1( DIM, N, file, rngSeed );
106   return 0;
107 }
```

```
1  // Test2D-v2.cpp: File with all the functions to test the
2  // imprecise version of the Tribond 2D algorithm.
3  //
4
5  #include <ctime>
6  #include <string>
7  #include "Tribond.h"
8  using namespace std;
9
10
```

```
11 int iniRandGen( int rngSeed )
12 {
13   // Initialize the random number generator.
14   if (rngSeed <= -1)
15   {
16     rngSeed = time( NULL );
17   }
18   srand( rngSeed );
19
20   return 0;
21 }
22
23
24 int processArgs( int argc, char** argv, int& N, string& file, ↪
     ↩ int& rngSeed,
25                  int& precision, int& coreSize )
26 {
27   // Process input from user to get problem parameters.
28   if( argc == 6 )
29   {
30     N = atoi( argv[1] );
31     file = argv[2];
32     rngSeed = atoi( argv[3] );
33     precision = atoi( argv[4] );
34     coreSize = atoi( argv[5] );
35     cout << "N, file, rngSeed, precision, coreSize: " << N << ↪
        ↩ ", " << file
36       << ", " << rngSeed << ", " << precision << ", " << ↪
          ↩ coreSize << endl;
37   }
38   else
39   {
40     cout << "Usage: " << "./test2d N(>4) \"rand2\" rngSeed ↪
        ↩ precision coreSize" << endl;
41     exit(0);
42   }
43
44   return 0;
45 }
46
47
48 int test1( int DIM, int N, string file, int rngSeed )
49 {
50   // // Initialize the random number generator.
51   // if (rngSeed <= -1)
```

```
52    // {
53    //   rngSeed = time( NULL );
54    // }
55    // srand( rngSeed );
56    iniRandGen( rngSeed );
57
58    // Setup target structure and attempt reconstruction.
59    Structure targetStru( DIM, N, file );
60    Structure testStru( DIM, N, targetStru.currDL );
61    testStru.reconstruct();
62
63    // Compare reconstructed structure with the target.
64    targetStru.home();
65    testStru.home();
66    compareStru( testStru, targetStru );
67    // testStru.printDLtoFile( "distanceList.txt" );
68
69    return 0;
70 }
71
72
73 int test2( int DIM, int N, string file, int rngSeed, int ↪
      ↩ precision,
74               int coreSize )
75 {
76    // cout << "check pars: " << DIM << ", " << N << ", " << file ↪
         ↩ << ", "
77    //      << rngSeed << ", " << precision << ", " << coreSize ↪
         ↩ << endl;
78    iniRandGen( rngSeed );
79
80    // Setup target structure and attempt reconstruction.
81    Structure targetStru( DIM, N, file );
82    targetStru.home();
83
84    vector<Point> core = targetStru.getCore( coreSize );
85    targetStru.reduceDLprecision( precision );
86    Structure testStru( DIM, N, targetStru.targetDL );
87    testStru.atoms = core;
88
89    // testStru.atoms = targetStru.getCore( coreSize );
90    testStru.currSize = testStru.atoms.size();
91    cout << "core size: " << testStru.atoms.size() << endl;
92    testStru.print();
93
```

```
94    // Point smPt;
95    // smPt.x = -2.04305609; smPt.y = 1.36504993; smPt.z = 0;
96    // cout << "Point, cost: " << smPt << '\t' << ↪
         ↩ testStru.getPtCost( smPt ) << endl;
97    // return 1;
98    // smPt.x = 1.17447201; smPt.y = -2.80072967; smPt.z = 0;
99    // cout << "Point, cost: " << smPt << '\t' << ↪
         ↩ testStru.getPtCost( smPt ) << endl;
100   // return 1;
101
102   bool successFlag = testStru.reconstruct2();
103   cout << "testStru size: " << testStru.atoms.size() << endl;
104   testStru.print();
105
106   print2structures( testStru.atoms, targetStru.atoms );
107   // Compare reconstructed structure with the target.
108   if( successFlag )
109   {
110     targetStru.home();
111     testStru.home();
112     compareStru( testStru, targetStru );
113   }
114   else
115   {
116     cout << "Reconstruction failed!" << endl;
117   }
118
119   return 0;
120 }
121
122
123 int main( int argc, char** argv )
124 {
125   int DIM = 2, N, rngSeed, precision, coreSize;
126   string file;
127   processArgs( argc, argv, N, file, rngSeed, precision, ↪
         ↩ coreSize );
128
129   // test1( DIM, N, file, rngSeed );
130   test2( DIM, N, file, rngSeed, precision, coreSize );
131   return 0;
132 }
```

```
1  // Test3D-v2.cpp: File with all the functions to test the
2  // imprecise version of the Tribond 3D algorithm.
```

```
3  //
4
5  #include <ctime>
6  #include <string>
7  #include "Tribond.h"
8  using namespace std;
9
10
11 int iniRandGen( int rngSeed )
12 {
13   // Initialize the random number generator.
14   if (rngSeed <= -1)
15   {
16     rngSeed = time( NULL );
17   }
18   srand( rngSeed );
19
20   return 0;
21 }
22
23
24 int processArgs( int argc, char** argv, int& N, string& file, int& rngSeed,
25                  int& precision, int& coreSize )
26 {
27   // Process input from user to get problem parameters.
28   if( argc == 6 )
29   {
30     N = atoi( argv[1] );
31     file = argv[2];
32     rngSeed = atoi( argv[3] );
33     precision = atoi( argv[4] );
34     coreSize = atoi( argv[5] );
35     cout << "N, file, rngSeed, precision, coreSize: " << N << ", " << file
36          << ", " << rngSeed << ", " << precision << ", " << coreSize << endl;
37   }
38   else
39   {
40     cout << "Usage: " << "./test2d N(>4) \"rand2\" rngSeed precision coreSize" << endl;
41     exit(0);
42   }
43
```

```
44    return 0;
45  }
46
47
48  int test1( int DIM, int N, string file, int rngSeed )
49  {
50    // // Initialize the random number generator.
51    // if (rngSeed <= -1)
52    // {
53    //   rngSeed = time( NULL );
54    // }
55    // srand( rngSeed );
56    iniRandGen( rngSeed );
57
58    // Setup target structure and attempt reconstruction.
59    Structure targetStru( DIM, N, file );
60    Structure testStru( DIM, N, targetStru.currDL );
61    testStru.reconstruct();
62
63    // Compare reconstructed structure with the target.
64    targetStru.home();
65    testStru.home();
66    compareStru( testStru, targetStru );
67    // testStru.printDLtoFile( "distanceList.txt" );
68
69    return 0;
70  }
71
72
73  int test2( int DIM, int N, string file, int rngSeed, int ↪
      ↩ precision,
74             int coreSize )
75  {
76    cout << "check␣pars:␣" << DIM << ",␣" << N << ",␣" << file << ↪
        ↩ ",␣"
77         << rngSeed << ",␣" << precision << ",␣" << coreSize << ↪
            ↩ endl;
78    iniRandGen( rngSeed );
79
80    // Setup target structure and attempt reconstruction.
81    Structure targetStru( N, file );
82    // targetStru.home();
83
84    vector<Point> core = targetStru.getCore( coreSize );
85    targetStru.reduceDLprecision( precision );
```

```
86    Structure testStru( DIM, N, targetStru.targetDL );
87    testStru.atoms = core;
88
89    // testStru.atoms = targetStru.getCore( coreSize );
90    testStru.currSize = testStru.atoms.size();
91    cout << "core size: " << testStru.atoms.size() << endl;
92    testStru.print();
93
94    // Point smPt;
95    // smPt.x = -2.04305609; smPt.y = 1.36504993; smPt.z = 0;
96    // cout << "Point, cost: " << smPt << '\t' << ↪
        ↩ testStru.getPtCost( smPt ) << endl;
97    // return 1;
98    // smPt.x = 1.17447201; smPt.y = -2.80072967; smPt.z = 0;
99    // cout << "Point, cost: " << smPt << '\t' << ↪
        ↩ testStru.getPtCost( smPt ) << endl;
100   // return 1;
101
102   bool successFlag = testStru.reconstruct2();
103   cout << "testStru size: " << testStru.atoms.size() << endl;
104   testStru.print();
105
106   print2structures( testStru.atoms, targetStru.atoms );
107   // Compare reconstructed structure with the target.
108   if( successFlag )
109   {
110     targetStru.home();
111     testStru.home();
112     compareStru( testStru, targetStru );
113   }
114   else
115   {
116     cout << "Reconstruction failed!" << endl;
117   }
118
119   return 0;
120 }
121
122
123 int main( int argc, char** argv )
124 {
125   int DIM = 3, N, rngSeed, precision, coreSize;
126   string file;
127   processArgs( argc, argv, N, file, rngSeed, precision, ↪
        ↩ coreSize );
```

```
128
129     // test1( DIM, N, file , rngSeed );
130     test2( DIM, N, file , rngSeed , precision , coreSize );
131     return 0;
132 }
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] G. M. Crippen and T. F. Havel, *Distance Geometry and Molecular Conformation*. Wiley and Sons, New York, 1988.

[2] G. Crippen, "Chemical distance geometry: current realization and future projection," *Journal of mathematical chemistry*, vol. 6, no. 1, pp. 307–324, 1991.

[3] K. Wuthrich, "The development of nuclear magnetic resonance spectroscopy as a technique for protein structure determination," *Accounts of Chemical Research*, vol. 22, pp. 36–44, Jan. 1989.

[4] K. Wuthrich, "Protein structure determination in solution by nuclear magnetic resonance spectroscopy," *Science*, 1989.

[5] M. Li, Y. Otachi, and T. Tokuyama, "Efficient algorithms for network localization using cores of underlying graphs," *Algorithms for Sensor Systems*, pp. 101–114, 2012.

[6] R. L. McGreevy and L. Pusztai, "Reverse Monte Carlo Simulation: A New Technique for the Determination of Disordered Structures," *Molecular Simulation*, vol. 1, pp. 359–367, Dec. 1988.

[7] A. L. Patterson, "Ambiguities in the X-Ray Analysis of Crystal Structures," *Phys. Rev.*, vol. 65, pp. 195–201, Mar. 1944.

[8] J. Yoon, Y. Gad, and Z. Wu, "Mathematical modeling of protein structure using distance geometry," tech. rep., 2000.

[9] J. C. Kendrew, Dickerson R. E., B. E. Strandberg, R. G. Hart, D. R. Davies, D. C. Phillips, and V. C. Shore, "Structure of Myoglobin," *Nature*, vol. 185, pp. 422–427, 1960.

[10] M. F. Perutz, M. Rossmann, A. Cullis, H. Muirhead, G. Will, and A. C. T. North, "Structure of Haemoglobin," *Nature*, vol. 185, pp. 416–422, 1960.

[11] J. Miao, H. N. Chapman, J. Kirz, D. Sayre, and K. O. Hodgson, "Taking X-ray diffraction to the limit: macromolecular structures from femtosecond X-ray pulses and diffrac-

tion microscopy of cells with synchrotron radiation.," *Annual review of biophysics and biomolecular structure*, vol. 33, pp. 157–76, Jan. 2004.

[12] J. Miao, J. Kirz, and D. Sayre, "The oversampling phasing method research papers," *Acta Crystallographica Section D*, pp. 1312–1315, 2000.

[13] J. Wu, K. Leinenweber, J. C. H. Spence, and M. O'Keeffe, "Ab initio phasing of X-ray powder diffraction patterns by charge flipping.," *Nature materials*, vol. 5, pp. 647–52, Aug. 2006.

[14] V. L. Shneerson, A. Ourmazd, and D. K. Saldin, "Crystallography without crystals. I. The common-line method for assembling a three-dimensional diffraction volume from single-particle scattering.," *Acta Crystallographica Section A*, vol. 64, pp. 303–15, Mar. 2008.

[15] Y. Jiao and S. Torquato, "Geometrical ambiguity of pair statistics: Point configurations," *Physical Review E*, vol. 81, pp. 1–11, Jan. 2010.

[16] Y. Jiao, F. Stillinger, and S. Torquato, "Geometrical ambiguity of pair statistics. II. Heterogeneous media," *Physical Review E*, vol. 82, pp. 1–11, July 2010.

[17] D. Cule and S. Torquato, "Generating random media from limited microstructural information via stochastic optimization," *Journal of applied physics*, vol. 86, no. 6, p. 3428, 1999.

[18] T. Egami and S. J. L. Billinge, *Underneath the Bragg Peaks: Structural Analysis of Complex Materials.* Oxford: Pergamon Press, Elsevier, 2003.

[19] M. Nilges and S. I. O'Donoghue, "Ambiguous NOEs and automated NOE assignment," *Progress in Nuclear Magnetic Resonance Spectroscopy*, vol. 32, pp. 107–139, Apr. 1998.

[20] B. Hendrickson, "The molecule problem: Exploiting structure in global optimization," *SIAM Journal on Optimization*, vol. 5, no. 4, pp. 835–857, 1995.

[21] B. Berger, J. Kleinberg, and T. Leighton, "Reconstructing a three-dimensional model with arbitrary errors," *Journal of the ACM (JACM)*, pp. 1–16, 1999.

[22] H. Lin, E. Božin, S. Billinge, E. Quarez, and M. Kanatzidis, "Nanoscale clusters in the high performance thermoelectric AgPbmSbTem+2," *Physical Review B*, vol. 72, pp. 1–7, Nov. 2005.

[23] L. Malavasi, G. a. Artioli, H. Kim, B. Maroni, B. Joseph, Y. Ren, T. Proffen, and S. J. L. Billinge, "Local structural investigation of SmFeAsOxF(x) high temperature superconductors.," *Journal of physics. Condensed matter*, vol. 23, p. 272201, July 2011.

[24] T. Proffen and S. Billinge, "Probing the local structure of doped manganites using the atomic pair distribution function," *Applied Physics A*, vol. 74, pp. 1770–1772, 2002.

[25] S. J. Billinge, "Nanoscale structural order from the atomic pair distribution function (PDF): There's plenty of room in the middle," *Journal of Solid State Chemistry*, vol. 181, pp. 1695–1700, July 2008.

[26] S. J. L. Billinge and M. G. Kanatzidis, "Beyond crystallography: the study of disorder, nanocrystallinity and crystallographically challenged materials with pair distribution functions.," *Chemical communications (Cambridge, England)*, pp. 749–60, Apr. 2004.

[27] S. J. L. Billinge and I. Levin, "The problem with determining atomic structure at the nanoscale.," *Science (New York, N.Y.)*, vol. 316, pp. 561–5, Apr. 2007.

[28] P. Juhás, D. M. Cherba, P. M. Duxbury, W. F. Punch, and S. J. L. Billinge, "Ab initio determination of solid-state nanostructure.," *Nature*, vol. 440, pp. 655–8, Mar. 2006.

[29] P. Juhás, L. Granlund, P. M. Duxbury, W. F. Punch, and S. J. L. Billinge, "The Liga algorithm for ab initio determination of nanostructure.," *Acta crystallographica. Section A, Foundations of crystallography*, vol. 64, pp. 631–40, Nov. 2008.

[30] P. Juhas, L. Granlund, S. R. Gujarathi, P. M. Duxbury, and S. J. L. Billinge, "Crystal structure solution from experimentally determined atomic pair distribution functions," *Journal of Applied Crystallography*, vol. 43, pp. 623–629, 2010.

[31] J. Moré and Z. Wu, "ε-Optimal Solutions To Distance Geometry Problems Via Global Continuation," Tech. Rep. May, 1995.

[32] J. Saxe, "Embeddability of weighted graphs in k-space is strongly NP-hard," *Proc. 17th Allerton Conference in Communications, Control and Computing*, vol. 480-489, 1979.

[33] D. Freeman, "Maximizing irregularity and the Golomb ruler problem," *Available through internet at http://citeseer. nj. nec. com/6709. html*, 1997.

[34] G. Bloom and S. Golomb, "Applications of numbered undirected graphs," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 562–570, 1977.

[35] J. N. Franklin, "Ambiguities in the X-ray analysis of crystal structures," *Acta Crystallographica Section A*, vol. 30, pp. 698–702, Nov. 1974.

[36] G. Bloom, "A counterexample to a theorem of S. Piccard," *Journal of Combinatorial Theory, Series A*, vol. 22, no. 3, pp. 378–379, 1977.

[37] W. Babcock, "Intermodulation interference in radio systems," *Bell Systems Technical Journal*, 1953.

[38] E. Blum, J. Ribes, and F. Biraud, "Some new possibilities of optimum synthetic linear arrays for radioastronomy," *Astronomy and Astrophysics*, vol. 41, no. 3-4, pp. 409–411, 1975.

[39] F. Biraud, E. Blum, and J. Ribes, "On optimum synthetic linear arrays with application to radioastronomy," *IEEE Transactions on Antennas and Propagation*, pp. 108–109, 1974.

[40] A. Moffet, "Minimum-redundancy linear arrays," *IEEE Transactions on Antennas and Propagation*, vol. 16, pp. 172–175, Mar. 1968.

[41] D. Robertson, "Geophysical applications of very-long-baseline interferometry," *Reviews of modern physics*, vol. 63, no. 4, pp. 899–918, 1991.

[42] A. K. Dewdney, "Computer recreations," *Scientific American*, pp. 16–26, Dec. 1985.

[43] A. K. Dewdney, "Computer recreations," *Scientific American*, pp. 14–21, Mar. 1986.

[44] M. Gardner, "Mathematical games," *Scientific American*, vol. 226, pp. 108–112, 1972.

[45] M. Gardner, "Mathematical games," *Scientific American*, vol. 226, pp. 114–118, June 1972.

[46] A. Eckler, "The construction of missile guidance codes resistant to random interference," *Bell Syst Technical J*, vol. 39, no. 3, pp. 973–994, 1960.

[47] A. Dimitromanolakis, *Analysis of the Golomb Ruler and the Sidon Set Problems, and Determination of Large, Near-Optimal Golomb Rulers*. PhD thesis, 2002.

[48] P. Erdös and P. Turán, "On a problem of Sidon in additive number theory, and on some related problems," *Journal of the London Mathematical Society*, vol. 16, pp. 212–215, 1941.

[49] S. Sidon, ""Ein Satz über trigonometrische Polynome und seine Anwendungen in der Theorie der Fourier-Reihen"," *Mathematische Annalen*, vol. 106, pp. 536–539, 1932.

[50] M. Ajtai, J. Kolmos, and E. Szemeredi, "A dense infinite Sidon sequence," *European Journal of Combinatorics*, vol. 2, pp. 1–11, 1981.

[51] R. Bose, "An affine analogue of Singers theorem," *J. Indian Math. Soc*, vol. 6, pp. 1–15, 1942.

[52] I. Z. Ruzsa, "Solving a linear equation in a set of integers I," *Acta Arithmetica*, vol. 3, no. LXV, pp. 259–282, 1993.

[53] B. Lindstrom, "Finding finite B2-sequences faster," *Mathematics of Computation*, vol. 67, no. 223, pp. 1173–1178, 1998.

[54] C. Meyer and P. a. Papakonstantinou, "On the complexity of constructing Golomb Rulers," *Discrete Applied Mathematics*, vol. 157, pp. 738–748, Feb. 2009.

[55] J. Robinson and A. Bernstein, "A class of binary recurrent codes with limited error propagation," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 106–113, 1967.

[56] J. Shearer, "Some new optimum Golomb rulers," *IEEE Transactions on Information Theory*, vol. 36, no. 1, pp. 183–184, 1990.

[57] W. Rankin, *Optimal golomb rulers: An exhaustive parallel search implementation*. PhD thesis, 1993.

[58] A. Dollas, W. Rankin, and D. McCracken, "A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler," *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 379–382, 1998.

[59] "http://distributed.net/ogr."

[60] A. K. Hartmann and M. Weigt, *Phase Transitions in Combinatorial Optimization Problems: Basics, Algorithms and Statistical Mechanics*. Wiley-VCH, Berlin, 2005.

[61] D. Achlioptas, A. Naor, and Y. Peres, "Rigorous location of phase transitions in hard optimization problems.," *Nature*, vol. 435, pp. 759–64, June 2005.

[62] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky, "Determining computational complexity from characteristic'phase transitions'," *Nature*, vol. 400, no. 6740, pp. 133–137, 1999.

[63] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, vol. AAAI Press, p. 440, 1992.

[64] R. Monasson and R. Zecchina, "Statistical mechanics of the random K-satisfiability model," *Physical Review E*, vol. 56, no. 2, p. 1357, 1997.

[65] M. Weigt and A. K. Hartmann, "Number of guards needed by a museum: a phase transition in vertex covering of random graphs.," *Physical review letters*, vol. 84, pp. 6118–21, June 2000.

[66] C. Fay, J. Liu, and P. Duxbury, "Maximum independent set on diluted triangular lattices," *Physical Review E*, vol. 73, pp. 1–14, May 2006.

[67] D. Johnson and M. Garey, "Computers and Intractability: A Guide to the Theory of NP-completeness," *Freeman & Co, San Francisco*, 1979.

[68] R. Moessner and A. P. Ramirez, "Geometrical Frustration," *Physics Today*, vol. 59, no. 2, p. 24, 2006.

[69] A. P. Ramirez, "Strongly Geometrically Frustrated Magnets," *Annual Review of Materials Science*, vol. 24, pp. 453–480, Aug. 1994.

[70] B. Roth, "Rigid and flexible frameworks," *The American Mathematical Monthly*, vol. 88, no. 1, pp. 6–21, 1981.

[71] H. Crapo, "Structural rigidity," *Structural Topology*, vol. 1, pp. 26–45, 1979.

[72] L. Asimow and B. Roth, "The rigidity of graphs, II," *Journal of Mathematical Analysis and Applications*, vol. 68, pp. 171–190, 1979.

[73] L. Asimow and B. Roth, "The rigidity of graphs," *Transactions of the American Mathematical Society*, vol. 245, no. November 1978, pp. 279–289, 1978.

[74] M. Thorpe and P. Duxbury, *Rigidity Theory and Applications.* Springer, 1999.

[75] G. Laman, "On graphs and rigidity of plane skeletal structures," *Journal of Engineering Mathematics*, vol. 4, pp. 331–340, Oct. 1970.

[76] R. Kenna, "Homotopy in statistical physics," *Condensed Matter Physics*, vol. 9, pp. 283–304, 2006.