UCSF
UC San Francisco Electronic Theses and Dissertations

Title
Automated molecular docking

Permalink
https://escholarship.org/uc/item/0zm6t0tq

Author
Ewing, Todd Jonathan August

Publication Date
1997

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

AUTOMATED MOLECULAR DOCKING:
DEVELOPMENT AND EVALUATION OF NEW SEARCH METHODS

by

Todd Jonathan August Ewing

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

PHARMACEUTICAL CHEMISTRY

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

San Francisco

Copyright 1997
by
Todd J. A. Ewing

Preface

The material in chapter 2 originally appeared in the Journal of Computational Chemistry, authored by Todd J. A. Ewing and Irwin D. Kuntz, entitled "Critical Evaluation of Search Algorithms for Automated Molecular Docking and Database Screening," (1997), Volume 18, pp. 1175-1189. The material was originally copyrighted in 1997 by John Wiley & Sons, Inc.

I would like to thank my advisor, Tack Kuntz, for providing a warm, stimulating and rewarding research environment for graduate study.

I would like to thank present and past members of my research group for their partnership in research: Keith Burdick, Barbara Chapman, Cindy Corwin, Andy Good, Dan Gschwend, Donna Hendrix, Luke Hoffman, Ron Knegtel, J. Y. Liang, Paul McCloskey, Elaine Meng, Shingo Makino, John Olson, Connie Oshiro, Diana Roe, Geoff Skillman, Yax Sun, Malin Young, and Xiaoqin Zou.

I would like to thank the members of my oral examination committee for their participation: Fred Cohen, Ken Dill, Tom Ferrin, Teri Klein, and Wolfgang Sadee.

Most of all, I would like to thank my wife, Tara Ewing, for her unwavering support.

Abstract

The DOCK program constructs possible binding modes of a small molecule within a macromolecular site. It can be used to screen a database of molecules and identify the candidates most likely to bind. Success depends on the thoroughness of the search algorithm and the accuracy of the scoring function.

This work focuses on the development and validation of efficient search algorithms for incorporation into DOCK. A rigid-body orientation search is developed to dock rigid molecules. The rigid-body search is coupled with a highly pruned conformation search to dock flexible molecules. An assessment protocol is developed to monitor the performance of the new search algorithms in comparison with reasonable control algorithms. The protocol is based on reliably screening a test database.

Using the assessment protocol, the new search algorithms perform significantly better than the control algorithms. The rigid molecule docking method converges in 8 seconds per molecule, which is 60 fold faster than random and 3 fold faster than existing methods. The flexible molecule docking method converges in 100 seconds per molecule, whereas the control methods do not converge.

The new search algorithms have been incorporated into version 4.0 of the DOCK software package. As of the date of this publication, DOCK 4.0 has been distributed to approximately 12-24 industrial sites and 50-75 academic sites.

Table of Contents

Chapter 1: Introduction 1
References 3

Chapter 2: Search Methods for Rigid Molecule Docking 7
Summary 7
Introduction 7
Methods 9
Bipartite Docking Graph 9
Single Docking Graph 11
Matching Parameters 14
Minimum Match Size 16
Additional Search Constraints 17
Control Methods 18
Test System 20
Results and Discussion 24
Docking conditions 24
Resource Usage 25
Test Systems 26
Score Convergence 29
Rank Convergence 31
BGM versus SGM 31
BRM versus SGM 32
BRM versus BGM 35
Future Directions 35
Conclusion 37
References 38

Chapter 3: Search Methods for Flexible Molecule Docking 43
Summary 43
Introduction 43
Methods 48
Molecule Segmentation 48
Docking Algorithm 50
Scoring 53
Evaluation Techniques 55
Results and Discussion 61
Pruning Methods 61
Rebuild X-ray Complexes 63
Database Screening 70
Conclusion 72
References 72

Chapter 4: Conclusion 78

Appendix 1: Users Guide....81
Appendix 2: Reference Manual....113
Appendix 3: Dock Source Code....177

List of Tables

Table 1: Search Methods....19
Table 2: Sampling Parameters 21
Table 3: Scoring and Optimization Parameters 23
Table 4: Comparison Methods....24
Table 5: Receptor Structures from PDB Used for Test Systems....25
Table 6: Convergence Properties of Search Methods. 29
Table 7: Ligand Segmentation Algorithm....49
Table 8: X-ray Crystal Complexes used to Evaluate Docking Algorithm....55
Table 9: Accuracy of Dockings....63
Table 10: Speed of Dockings....70

List of Figures

Figure 1. Bipartite Graph Matching Algorithm 10
Figure 2. Adjacency Matrix for Single Docking Graph Algorithm. 12
Figure 3. Single Docking Graph Matching Algorithm. 13
Figure 4. Relative Chirality of Match Points 17
Figure 5. Weighted Average Score for Molecules Docked to 3DFR using SPHGEN site points 26
Figure 6. Weighted Rank Correlation for Molecules Docked to 3DFR using SPHGEN site points 27
Figure 7. Weighted Average RMSD for Molecules Docked to 3DFR using SPHGEN site points 28
Figure 8. How Molecule Size affects Force Field Score 33
Figure 9. How Molecule Size Affects Bump Filtering 34
Figure 10. Flowchart of Flexible Docking Algorithm 46
Figure 11. Pruning Methods for Incremental Construction 62
Figure 12. Scatter Plot of Docked Positions of Biotin with Streptavidin (1stp) 66
Figure 13. Top-scoring binding mode of biotin with streptavidin 67
Figure 14. Scatter Plot of Docked Positions of Glycyl-L-Tyrosine with Carboxypeptidase A (3cpa) 68
Figure 15. Top-scoring binding modes of glycyl-l-tyrosine with carboxypeptidase A (3cpa) 69
Figure 16. Accuracy versus time of database processing 71

Chapter 1: Introduction

With the advent of high resolution x-ray crystallography and NMR, structural chemists and biologists can study bio-macromolecular interactions in atomic detail. This information, combined with computational and visualization tools, has helped spawn the field of structure-based ligand design. A common step in the design cycle is the process of molecular docking, in which possible binding geometries of a molecule with a macromolecule are studied.

The docking process can be divided into two parts: a search algorithm and a scoring algorithm. The search algorithm should sample the degrees of freedom of the ligand:macromolecule system sufficiently to include the true binding mode(s). The scoring algorithm should represent the thermodynamics of interaction sufficiently to distinguish the true binding mode(s) from all others explored.

Because of the computationally expensive nature of the search problem, many different solutions have been proposed. Docking with molecular dynamics and Monte Carlo algorithms has been explored (Caflisch, Niederer, & Anliker, 1992; Di Nola, Roccatano, & Berendsen, 1994), including simulated annealing (Abagyan, Totrov, & Kuznetsov, 1994; Goodsell & Olson, 1990; Moon & Howe, 1991) and MCSS (Miranker & Karplus, 1991) methods. Other docking protocols consider molecular flexibility, including rotamer search (Leach, 1994; Leach & Kuntz, 1992; Mizutani, Tomioka, & Itai, 1994), distance geometry (Smellie, Crippen, & Richards, 1991), and genetic algorithm (Jones, Willett, & Glen, 1995; Judson, Jaeger, & Treasurywala, 1994; Oshiro, Kuntz, & Dixon, 1995) methods. To make the search tractable for processing a large set of molecules, the molecular components are often treated as rigid objects. With this approximation, researchers have used

systematic searching (Pang & Kozikowski, 1994), pattern recognition (Fischer, Lin, Wolfson, & Nussinov, 1995; Katchalski-Katzir, et al., 1992; Rarey, Wefing, & Lenguauer, 1996), graph theoretical (Kuhl, Crippen, & Friesen, 1984; Kuntz, Blaney, Oatley, Langridge, & Ferrin, 1982; Lawrence & Davis, 1992; Miller, Kearsley, Underwood, & Sheridan, 1994), and other superposition (Bohm, 1994) techniques to dock molecules.

The UCSF DOCK program belongs to the group of methods employing the rigid body assumption and uses graph theoretical techniques. Because of its speed, the program is often used to screen a large database of molecules, selecting potential ligands of a receptor target (Kuntz, 1992). In this paper, the term "ligand" is used loosely; it refers to any small molecule whose binding is under study. The term "receptor" refers to the macromolecule whose binding pocket is being explored.

In Chapter 1, the orientation search portion of the docking problem is studied. A new matching algorithm is used to superimpose a rigid ligand within a receptor site. To address the relative performance of matching algorithms, new testing protocols are also developed. The testing protocols are based on the processing of a database of molecules. Performance is measured by the time convergence of scores and rankings of the docked molecules.

In Chapter 2, the conformation search portion of the docking problem is studied, extending consideration to flexible ligands. The rigid-body orientation search developed in Chapter 1 is used to dock a rigid portion of each molecule. A heavily pruned conformation search is used while reattaching the flexible portions. The accuracy of the method will be verified by reconstructing known binding complexes. The feasibility of applying the method to database processing will be assessed.

UCSF LIBRARY

REFERENCES

Abagyan R., Totrov M., & Kuznetsov D. (1994). ICM - A new method for protein modeling and design - applications to docking and structure prediction from the distorted native conformation. Journal of Computational Chemistry, 15 (5), 488-506.

Bohm, H. J. (1994). On the use of LUDI to search the fine chemicals directory for ligands of proteins of known three-dimensional structure. Journal of Computer-Aided Molecular Design, 8(5), 623-632.

Caflisch A., Niederer P., & Anliker M. (1992). Monte-carlo docking of oligopeptides to proteins. Proteins-Structure Function and Genetics, 13 (3), 223-230.

Dinola A., Roccatano D., & Berendsen H.J.C. (1994). Molecular dynamics simulation of the docking of substrates to proteins. Proteins-Structure Function and Genetics, 19 (3), 174-182.

Fischer, D., Lin, S. L., Wolfson H. L., & Nussinov, R. (1991). A geometry-based suite of molecular docking processes. Journal of Molecular Biology, 248(2), 459-477.

Goodsell D.S. & Olson A.J. (1990). Automated docking of substrates to proteins by simulated annealing. Proteins-Structure Function and Genetics, 8 (3), 195-202.

Jones, G., Willett, P., & Glen R.,C. (1995). Molecular recognition of receptor sites using a genetic algorithm with a description of desolvation. Journal of Molecular Biology, 245(1), 43-53.

Judson, R. S., Jaeger, E. P., & Treasurywala, A. M. (1994). A genetic algorithm based method for docking flexible molecules. Theochem-Journal of Molecular Structure, 114, 191-206.

Katchalski-katzir, E., Shariv, I., Eisenstein, M., Friesem, A. A., Aflalo, C., & Vakser, I. A. (1992). Molecular surface recognition - determination of geometric fit between proteins and their ligands by correlation techniques. Proceedings of the National Academy of Sciences of the United States of America, 89(6), 2195-2199.

Kuhl, F. S., Crippen, G. M., & Friesen, D. K. (1984). Journal of Computational Chemistry, 5, 24.

Kuntz, I. D. (1992). Structure-based strategies for drug design and discovery. Science, 257 (5073), 1078-1082.

Kuntz, I. D., Blaney, J. M., Oatley, S. J., Langridge, R., & Ferrin, T. E. (1982). Journal of Molecular Biology, 161, 269.

Lawrence, M. C. & Davis, P. C. (1992). Clix - a search algorithm for finding novel ligands capable of binding proteins of known 3-dimensional structure. Proteins-Structure Function and Genetics, 12(1), 31-41.

Leach A.R. & Kuntz I.D. (1992). Conformational analysis of flexible ligands in macromolecular receptor sites. Journal of Computational Chemistry, 13 (6), 730-748.

Leach A.R. (1994). Ligand docking to proteins with discrete side-chain flexibility. Journal of Molecular Biology, 235 (1), 345-356.

Miller, M. D., Kearsley, S. K., Underwood, D. J., & Sheridan, R. P. (1994). Flog - a system to select quasi-flexible ligands complementary to a receptor of known three-dimensional structure. Journal of Computer-Aided Molecular Design, 8(2), 153-174.

Miranker A. & Karplus M. (1991). Functionality maps of binding sites - a multiple copy simultaneous search method. Proteins-Structure Function and Genetics, 11 (1), 29-34.

Mizutani M.Y., Tomioka N., & Itai A. (1994). Rational automatic search method for stable docking models of protein and ligand. Journal of Molecular Biology, 1994 oct 21, v243 n2:310-326.

Moon J.B. & Howe W.J. (1991). Computer design of bioactive molecules - a

UCSF LIBRARY

method for receptor-based denovo ligand design. Proteins-Structure Function and Genetics, 11 (4), 314-328.

Oshiro, C. M., Kuntz, I. D., & Dixon, J. S. (1995). Flexible ligand docking using a genetic algorithm. Journal of Computer-Aided Molecular Design, 9(2), 113-130.

Pang, Y. P. & Kozikowski, A. P. (1994). Prediction of the binding site of 1-benzyl-4-[(5,6-dimethoxy-1-indanon-2-yl)methyl]piperidine in acetylcholinesterase by docking studies with the sysdoc program. Journal of Computer-Aided Molecular Design, 8(6), 683-693.

Rarey, M., Wefing, S., & Lengauer, T. (1991). Placement of medium-sized molecular fragments into active sites of proteins. Journal of Computer-Aided Molecular Design, 10(1) 41-54.

Smellie, A. S., Crippen, G. M., & Richards, W. G. (1991). Fast drug-receptor mapping by site-directed distances - a novel method of predicting new pharmacological leads. Journal of Chemical Information and Computer Sciences, 31(3), 386-392.

UCSF LIBRARY

Chapter 2: Search Methods for Rigid Molecule Docking

SUMMARY

The DOCK program explores possible orientations of a molecule within a macromolecular active site by superimposing atoms onto pre-computed site points. Here we compare a number of different search methods, including an exhaustive matching algorithm based on a single docking graph. We evaluate the performance of each method by screening a small database of molecules to a variety of macromolecular targets. By varying the amount of sampling, we can monitor the time convergence of scores and rankings. We not only show that the site point-directed search is tenfold faster than a random search, but that the single graph matching algorithm boosts the speed of database screening up to sixty-fold. The new algorithm, in fact, outperforms the bipartite graph matching algorithm currently used in DOCK. The results indicate that a critical issue for rapid database screening is the extent to which a search method biases run time toward the highest-ranking molecules. The single docking graph matching algorithm will be incorporated into DOCK version 4.0.

INTRODUCTION

The core of the DOCK search algorithm is the superimposition of ligand atoms onto predefined site points (Ferro & Hermans, 1977) that map out the negative image of the binding site. A matching process is used to determine which ligand atoms and site points are to be superimposed (Shoichet, Bodian, & Kuntz, 1992). Multiple orientations are generated this way, with each receiving a score assessing the intermolecular interac-

tions. This score is based on the intermolecular terms of a molecular mechanics force field (Meng, Shoichet, & Kuntz). Recently, an optimization procedure has been added that adjusts each orientation to improve the intermolecular interactions (Gschwend & Kuntz, 1996; Meng, Gschwend, Blaney, & Kuntz, 1993).

In this work, we critically evaluate several matching algorithms for the docking process, including an exhaustive matching algorithm. The exhaustive algorithm was presented by Bron and Kerbosch as a method to detect cliques in an undirected graph (Bron & Kerbosch, 1973). It was later incorporated into a docking algorithm by Crippen and coworkers (Kuhl et al., 1984; Smellie et al., 1991). It has many attractive features, so we wish to evaluate it in the context of rigid molecular docking, score optimization, and database screening. As an exhaustive search, it avoids some of the artifacts encountered by the current matching method. For example with a non-exhaustive algorithm, adjusting parameters to increase the total amount of sampling can reduce the amount of sampling of certain binding modes (Meng et al., 1993). Increased sampling, with the new algorithm, will always retain binding orientations found with less sampling, leading to a proper superset of binding modes. An exhaustive algorithm also does not require the additional parameters controlling the heuristics of the non-exhaustive search (Shoichet et al., 1992).

Though the matching algorithms formally treat the ligand and receptor as rigid objects, they can readily be incorporated into a flexible docking schemes (Bohm, 1992; DesJarlais, Sheridan, Dixon, Kuntz, & Venkataraghaven, 1986; Eisen, Wiley, Karplus, & Hubbard, 1994; Miller et al., 1994; Smellie et al., 1991). In future work, we will investigate how best to divide a flexible docking problem up into smaller rigid parts.

To evaluate the performance of the new matching algorithm, we propose a new

assessment protocol based on screening a small database of molecules. Since we specifically wish to minimize any artifacts due to the quality of the scoring function in this work, we will not use experimental measurements as the standard, but instead, the global minimum of our current scoring function. We will also evaluate random, and partially random, search algorithms as controls with which to put the current DOCK performance in perspective. These control algorithms let us investigate fundamental issues of orientational sampling, such as the effects of using site points to guide the search.

METHODS

Bipartite Docking Graph

Since the first release of DOCK, the search process has been driven by a matching procedure in which subsets of ligand atoms and receptor site points are identified that have equivalent internal distances (Kuntz et al., 1982). Matching is formulated as a graph theoretical problem in which the ligand atoms and receptor site points are separate sets of nodes in a bipartite graph (Shoichet et al., 1992). A match is defined as a set of compatible edges which connect a subset of ligand nodes with an equal number of receptor nodes. For the edges to be compatible, the distances among ligand nodes must map to equivalent distances among receptor nodes. An example of match formation is depicted in Figure 1. As this figure illustrates, to extend a match, all possible edges (including bad edges) must be considered; distance comparisons are used to identify and discard bad edges. Matches are extended until there are a sufficient number of nodes in the match to define a unique orientation of the ligand.

Since version 2.0, DOCK avoided considering some bad edges with a pruning

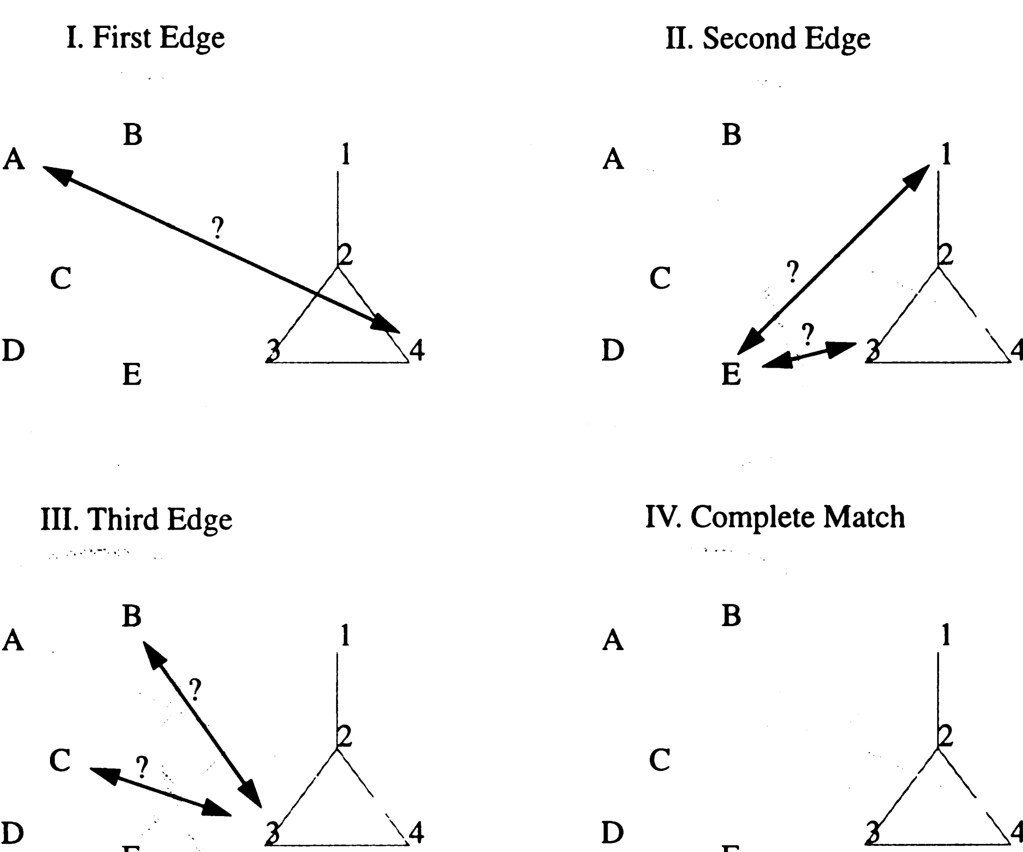


Figure 1. Bipartite Graph Matching Algorithm. The receptor site points (A-E) and ligand atoms (1-4) are separate sets of nodes in a bipartite graph. I. A first (seed) edge is considered. Of the 20 seed edges to be tried (5 points * 4 atoms), we first consider A4 for this example. In three dimensions, such a match would superimpose atom 4 onto site point A. This match would fix three out of six orientational degrees of freedom. II. Second edges are considered. Of the 12 edges to be tried (4 points left * 3 atoms left), we consider E3. Since AE > 43, we discard E3 as a second edge. Then we consider E1. Since AE = 41, we retain E1 as a second edge. In three dimensions, this match would superimpose atoms 4 and 1 onto points A and E, respectively. This match would fix two more orientational degrees of freedom. III. Third edges are considered. Of the six edges to be tried (3 points left * 2 atoms left), we consider C3. Though AC = 43, EC < 13 so we must discard C3 as a third edge. Then we consider B3. Since AB = 43 and EB = 13, we retain B3 as a third edge. This match fixes the last of six orientational degrees of freedom. IV. The match is large enough to define a unique orientation which superimposes atoms 4, 1 and 3 onto site points A, E and B, respectively.

UCSF LIBRARY

method involving distance binning (Shoichet et al., 1992). Nodes were pre-organized in distance bins, such that for each seed node, sets of nodes in discrete distance intervals from the seed were identified. These bins guide match extension from a seed edge (connecting seed nodes), ensuring that candidate edges are compatible with the seed edge. They do not, however, ensure compatibility with other non-seed edges already included in the match. For example, the binning algorithm would avoid considering the bad edge in step II of Figure 1, but not the bad edge in step III. The storage requirements for this algorithm grow as $N_n N_b N_{n/b} \le N_n^3$, where N_n is the number of nodes (ligand atoms or receptor site points). N_b is the number of distance bins and grows with the longest distance and the inverse of the bin width. $N_{n/b}$ is the number of nodes in each distance bin which grows with N_n and the bin width.

Single Docking Graph

Kuhl et. al. (1984) proposed merging the bipartite docking graphs into a single docking graph, which is then amenable to clique detection techniques developed by Bron and Kerbosch (1973). In a single docking graph, each node represents a pairing of an atom with a site point. Each edge identifies adjacent nodes, or two nodes for which both atom components and site point components are separated by equivalent distances. The docking graph is represented by an adjacency matrix in which each nonzero element identifies adjacent nodes. An adjacency matrix for the example depicted in Figure 1 is presented in Figure 2. The chief advantage with this representation is that all necessary distance comparisons are made during the construction of the adjacency matrix. Consequently, during matching the adjacency matrix is used as a rapid filter to ensure that no bad edges are ever considered. This type of matching is presented in Figure 3. Though the

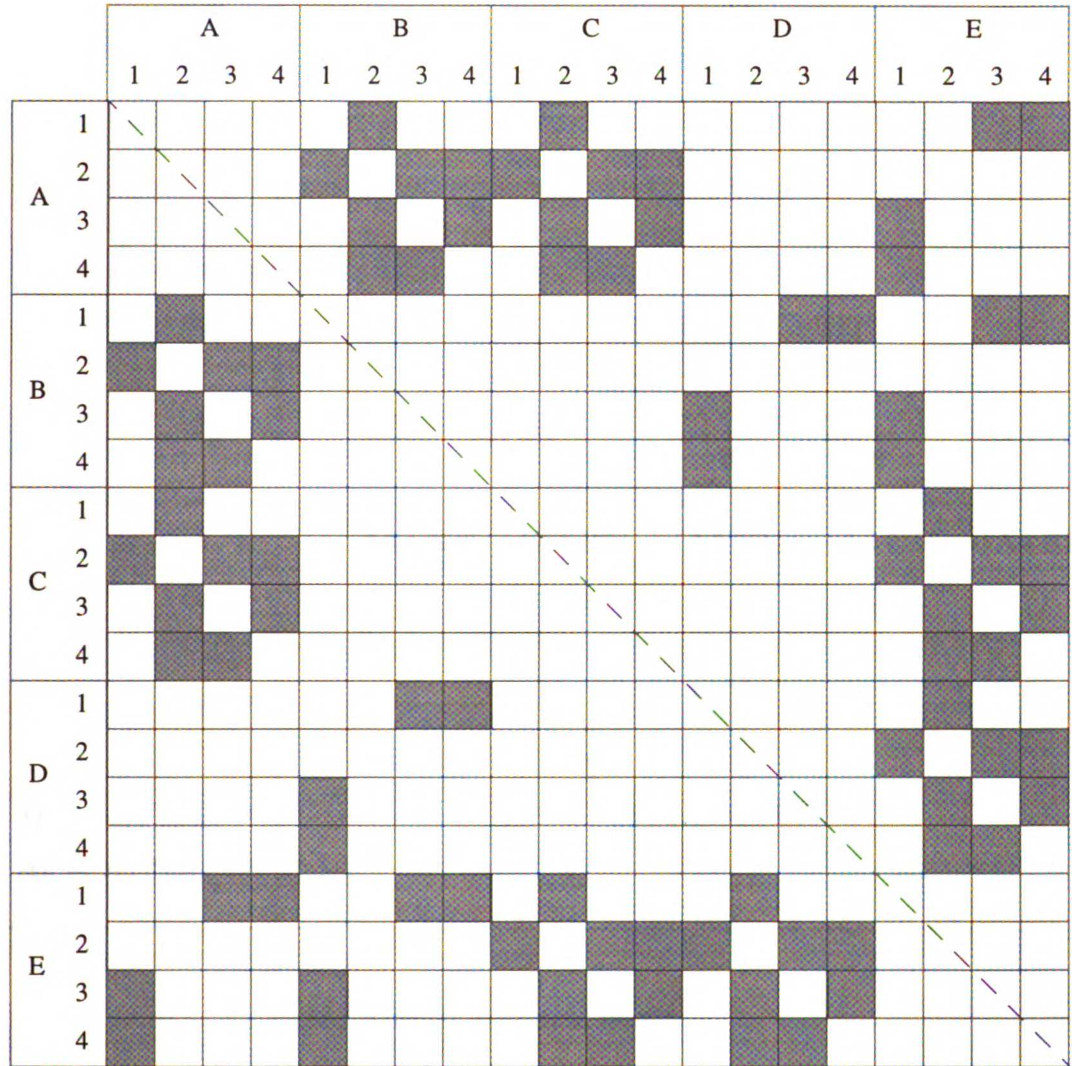


Figure 2. Adjacency Matrix for Single Docking Graph Algorithm. This matrix identifies all adjacent nodes for the example given in Figure 1. Each node is defined as a site point-ligand atom pair, e.g. A4. For two nodes to be adjacent, the intra-atom distance must be equal to the intra-site point distance. For example, matrix element (A4,E1) is turned on because $\overline{AE} = \overline{41}$. The matrix is symmetric.

single docking graph is one step removed from the intuitive appeal of the bipartite docking graph, it enables a more efficient solution to the docking problem.

The single docking graph representation has also been implemented in the FLOG docking program (Miller et al., 1994). This program heavily prunes the matching search

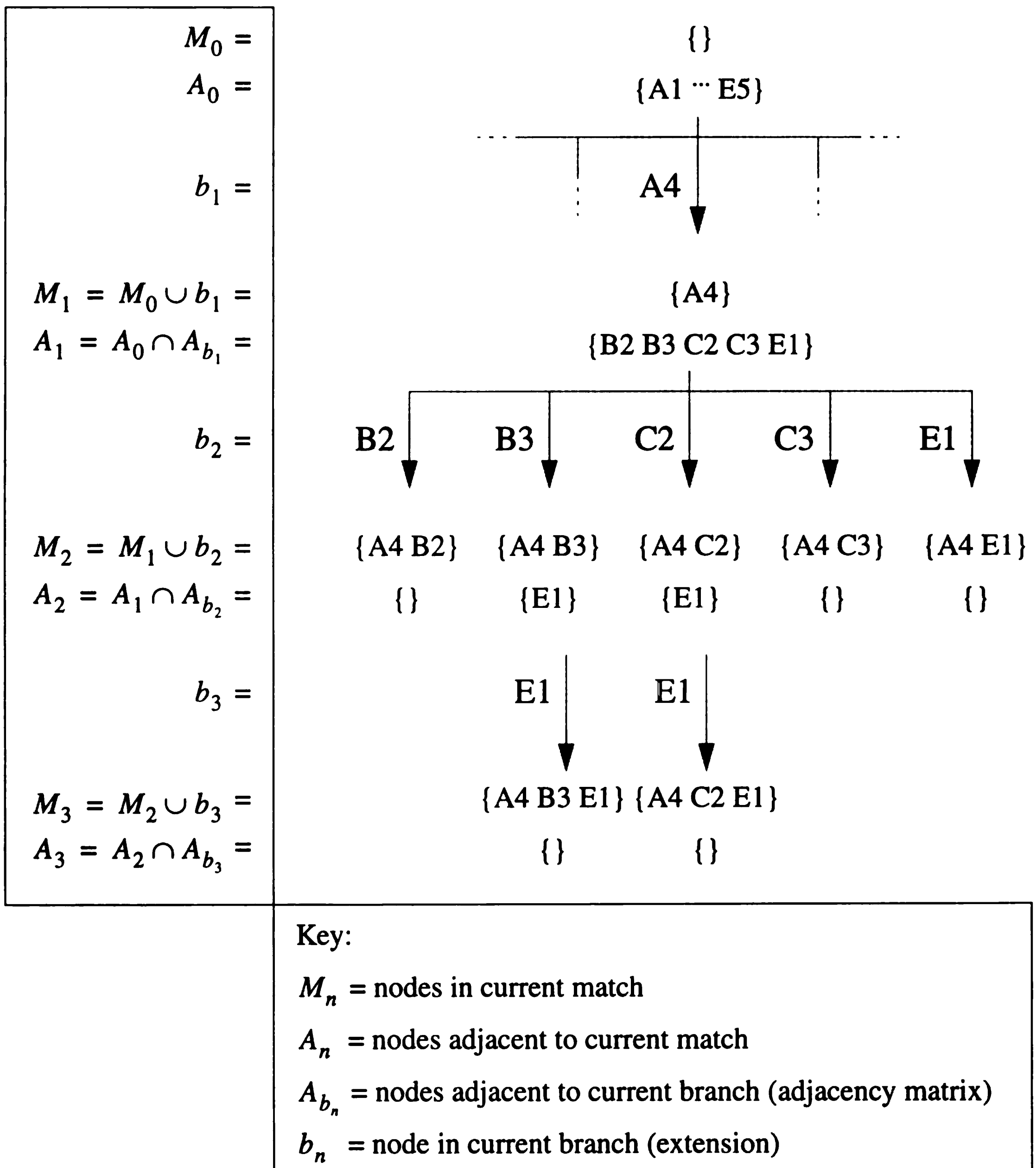


Figure 3. Single Docking Graph Matching Algorithm. This figure depicts the entire search path starting from the same seed used in Figure 1. Matching begins with the match set, M_0, empty and the adjacency set, A_0, maximally filled. A match is extended by finding the union of the previous match, M_{n-1}, and a branch node, b_n, which is selected from A_{n-1}. The new adjacency set, A_n, is the intersection of A_{n-1} with the set of all nodes adjacent to b_n, which is A_{b_n} and is taken from the adjacency matrix presented in Figure 2. Match extension continues until A_n is empty. Matches with three or more nodes define a unique ligand orientation (see text and Figure 4). The match set, {A4 B3 E1}, corresponds to the solution presented in Figure 1.

tree using a minimum-residual search heuristic. Though it examines all possible nodes at each branch point, it only pursues the node with the smallest difference in ligand and receptor distances with respect to the most recently added node in the match. Backtracking is only allowed at the seed level, where all possible nodes are pursued to initiate matching. As a result, the total number of matches can never exceed the number of nodes.

Here we propose implementing the single docking graph representation combined with a variation of the exhaustive clique detection method discussed by Bron and Kerbosch (1973). A clique is defined as a set of fully adjacent nodes (i.e. a completely connected subgraph) which cannot be further enlarged without adding a nonadjacent node. Much attention is given to the intractable nature of the maximum clique problem. It is classified as NP-complete because the solution time grows faster than any polynomial expression of the problem size (Kuhl et al., 1984). For the application of molecular docking, however, we are not trying to find the single, largest clique. The process of matching is in fact a process of finding completely connected subgraphs within an undirected graph: a less restrictive, and therefore more tractable problem than finding cliques and maximum cliques. Although Bron and Kerbosch actually present two methods and recommend a bounding technique for clique detection, we find their original, brute-force method sufficient for finding fully connected subgraphs in a manner efficient for molecular docking.

Matching Parameters

In our molecular docking implementation, we use two parameters to determine node adjacency: a distance tolerance and a distance minimum. The distance tolerance parameter addresses experimental uncertainty in the ligand and target structures. The distance minimum parameter enables the search to focus on the longer, more relevant internal

distances. If adjacency information is stored in a matrix, then, for most docking situations, the matrix can be very large. The matrix size grows as $(N_{lig}N_{rec})^2$, where N_{lig} is the number of non-hydrogen ligand atoms and N_{rec} is the number of receptor site points. Since these matrices are sparse (ca. 1% elements typically occupied), we store only the nonzero elements of each row of the matrix as an integer list of nodes. The probability, p_{on}, of an element being nonzero is a function of the distance tolerance and distance minimum. The memory requirement for the adjacency lists is $p_{on}(N_{lig}N_{rec})^2$. We presort each adjacency list so that the process of finding the common elements of two lists (the $A_n = A_{n-1} \cap A_{b_n}$ steps in Figure 3) can be performed on a once-through basis at a speed comparable to the use of the complete matrix.

The advantage of an exhaustive algorithm is that when sampling is increased, the search is guaranteed to include the search space explored at the lower sampling level. This property helps to avoid sampling artifacts encountered with the bipartite matching algorithm (Meng et al., 1993). Despite its exhaustive nature, it does not undergo a combinatorial explosion for larger systems because of user control over the sampling parameters. Typical sampling parameters for a docking scenario having fewer than 30 ligand non-hydrogen atoms and fewer than 50 target site points are: four nodes minimum for a match, 0.5 Å distance tolerance and 2.0 Å distance minimum for node adjacency. Since the search never explores invalid branches, search time grows as a function of the number of distance-constrained solutions rather than the number of possible unconstrained solutions. Therefore, docking larger molecules into larger sites can be made nearly as rapid if a smaller distance tolerance or larger distance minimum are chosen. Since memory reserved for the adjacency lists is dynamically allocated, the memory burden can be

adjusted as well.

Minimum Match Size

Some confusion exists in the literature over how many atoms and site points must be in a match to define a unique orientation. The orientation is generated by a ligand transformation which has a translation and a rotation component. The translation vector has three degrees of freedom. The rotation matrix has four degrees of freedom. Three of them are represented by the Euler angles. The fourth is represented by the sign of the determinant. A rotation with a positive determinant retains the handedness of the object it transforms, while one with a negative determinant will reverse the handedness of the object. If one knows in advance whether to reverse the handedness of the object, then only the three euler angles need to be determined. For example when docking a chiral ligand where only one stereoisomer is relevant (e.g. protein or peptide ligands), only the positive-determinant rotation matrix would be of interest. When docking a ligand available as a racemate, then both transformations would be of interest. The FLOG program (Miller et al., 1994), for instance, routinely samples both mirror images a ligand, even when the ligand is achiral. When the sign of the determinant is known in advance, the six degrees of freedom of the rotation and translation are uniquely determined by a match set containing three nonlinear atoms and site points.

Processing a larger match set causes only one transformation to be allowed. As illustrated in Figure 4, when a one-to-one mapping has been made between two sets of four non-planar points, then each set can be assigned a relative chirality. This is true even if the points come from an achiral molecule or from a set of site points where chirality is ambiguous. If the relative chiralities are the same, then the ligand can be oriented nor-

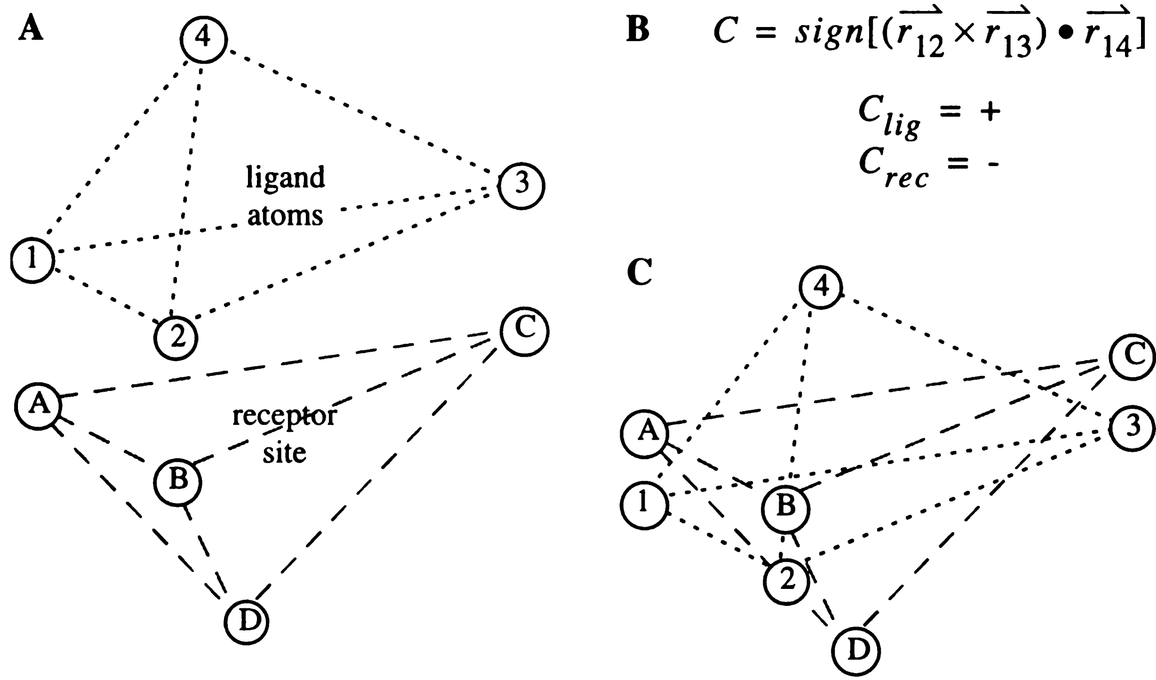


Figure 4. Relative Chirality of Match Points. A. Distance matching might identify a set of atoms {1 2 3 4} to match with a set of site points {A B C D}, such that A is with 1, B is with 2, and so on. Though the internal distances within the atoms are equivalent to those within the site points, the handedness is opposite. B. We define the relative chirality, *C*, according to the sign of the triple product. For any given four-point match, the probability that the relative chiralities are the same is 50%. C. If the chirally opposed sets are superimposed without inverting the chirality of the ligand set, then the resulting least squares fit is poor.

mally. If the relative chiralities are opposite, then either the ligand is inverted when oriented, or if that is not desired, then the match is discarded. In fact, all larger matches that are supersets of the discarded match are also discarded, because they too yield inconsistent matches. If these steps are not taken, then the resulting orientations will poorly superimpose the ligand atoms and receptor site points in the match set, even though all the distance tolerances are met (Figure 4C).

Additional Search Constraints

The systematic design of the matching algorithm makes it well suited to incorpo-

rate specialized search constraints. Some examples, though not assessed in this study, are mentioned because they have been shown to be useful elsewhere. To avoid oversampling particular binding modes, orientational degeneracy checking has been studied (Gschwend & Kuntz, 1996; Rarey et al., 1996). In the new matching algorithm, a degenerate orientation is detected as a degenerate match whose nodes are a subset of nodes in a larger match. In other words, only subgraphs that are true cliques need be processed. As another example, chemical information can be used to guide the matching process using labeled atoms and site points. Only nodes composed of a chemically compatible atom and site point are used to seed or extend a match. Much like the repellent node implementation of Kuhl et al. (1984), matches adjacent to chemically incompatible nodes are discarded. In addition, sampling can be focused on particular regions of the active site by defining critical site point clusters. This technique is similar to the approach used in targeted-DOCK (DesJarlais & Dixon, 1994) and FLOG (Miller et al., 1994), except that clusters can be of arbitrary size and number. The matching process automatically restricts itself to make sure all matches include members from each cluster. The new matching algorithm lends itself so well to these constraints that when activated, they contribute a negligible computational overhead, and can lead to considerable speed improvements for database searches (DesJarlais & Dixon, 1994; Miller et al., 1994).

Control Methods

We will test a total of five methods (Table 1) to isolate specific aspects of the search process. These methods range in complexity from a completely random search to the bipartite and single graph procedures described above. We begin with the Uniform Random Transformation (URT) method which explores a predefined rectangular volume

Table 1: Search Methods

Abbr	Method	Description	Hypothesis Tested
URT	Uniform Random Transformation	• Construct rectangular volume enclosing site. • Randomly move molecule center of mass within volume. • Randomly rotate. • Each molecule in database sampled uniformly.	• Random method used as reference. • Rectangular enclosure is sufficient. Site point description is unnecessary.
URM	Uniform Random Matching	• Match random subsets of atoms with random subsets of site points. • Superimpose match atoms onto match site points with a least squares fit. • Each molecule in database sampled uniformly.	• An irregular volume to describe the site is more efficient.
BRM	Biased Random Matching	• Match randomly (like URM). • SGM controls amount of sampling for each molecule in database.	• An irregular volume is more efficient. • Spending more time on molecules which match better is more efficient.
SGM	Single Graph Matching	• Using single graph, exhaustively match subsets of atoms with subsets of site points with equivalent internal distances (DOCK 4.0).	• Fitting some atoms precisely onto some site points is more efficient.
BGM	Bipartite Graph Matching	• Using bipartite graph, non-exhaustively match using binning algorithm (DOCK 3.5).	• Fitting some atoms precisely onto some site points is more efficient. • Binning algorithm more efficient.

enclosing the active site. It is the most simple and "hypothesis free" of the methods tested here. URT will indicate the minimum level of performance that we expect from any docking algorithm. The Uniform Random Matching (URM) method explores the irregularly shaped volume described by the collection of site points. It will show the performance gains, if any, of using a "negative image" approach to map out the binding site. The Biased Random Matching (BRM) method is identical to URM, except it uses the new matching algorithm to determine the number of random matches to try for each molecule.

UCSF LIBRARY

Once the number of matches has been determined, BRM uses completely random selections of nodes to form the actual matches used to generate molecule orientations. Since BRM is a hybrid approach, it is meant to help isolate the source of any differences between URM and the new matching algorithm. The Single Graph Matching (SGM) algorithm uses the new matching algorithm to determine both the number of matches and the actual orientations to try for each molecule. SGM will reveal the performance gains, if any, of using site points to not only map out the most interesting binding site volume, but to also direct the positioning of individual ligand atoms. The Bipartite Graph Matching (BGM) method is the existing DOCK 3.5 matching algorithm. It will reveal the advantage, if any, of using a non-exhaustive search method with a longest distance first heuristic. BGM is described last because within the spectrum of different search methods, its algorithm is the most elaborate.

Test System

We assess the performance of the search methods in the following way. We dock a set of 100 molecules, chosen randomly from the set of uncharged, medium-sized and generally rigid molecules in the Available Chemicals Database (ACD, distributed by Molecular Design Ltd., San Leandro, CA). In our study, a medium-sized molecule is one with 15 to 35 non-hydrogen atoms. A generally rigid molecule is one with no single bonds except those attaching hydrogen atoms, attaching terminal non-hydrogens (i.e. methyl or hydroxyl groups) or participating in ring structures. Molecules meeting these three criteria compose 40% of the ACD. For each molecule in the test set, a single CONCORD-generated conformation is used (Rusinko, Sheridan, Nilikantan, & Haraki, 1989).

We see several advantages to using such a data set of molecules to test the search

methods. First, the docking conditions represent a close approximation to the typical application of DOCK to database screening (DesJarlais et al., 1988). Not only can we study the convergence of score for each molecule, we can study the convergence of relative scores, or rankings, of the set of molecules. Second, the docking conditions allow us to explore a multitude of diverse molecular shapes so that our results are less subject to potential artifacts of a particular ligand:receptor system. Though some may argue that studying a set of known, potent ligands would be more relevant, we counter that the databases DOCK searches often do not contain potent binders, and that DOCK frequently finds micromolar inhibitors to serve as lead compounds (Kuntz, 1992). By choosing a random subset of molecules, we, in fact, will arrive at a set that best represents the typical array of molecules tested. By biasing the subset to include medium-sized, generally rigid molecules, we also focus on that portion of the database which is best treated by rigid molecular docking.

For each search method, we perform multiple docking runs, and vary the amount of sampling from zero to a value at which the docking results converge. The key sampling parameters for each method are listed in Table 2. Scoring and optimization parameters are

Table 2: Sampling Parameters

URT	
total orientations	0 - 900,000[a]
URM	
total orientations	0 - 100,000[a]
nodes min/max	3/3[b]
BRM	
distance tolerance	0 - 0.9 Å[a]
distance minimum	2.0 Å[c]
nodes min/max	3/3[b]

Table 2: Sampling Parameters (Continued)

SGM	
distance tolerance	0 - 0.6 Å[a]
distance minimum	2.0 Å[c]
nodes min/max	4/10[d]
BGM	
distance tolerance	1.5 Å[e]
ligand bin width	0.1 - 0.9 Å[a,f]
receptor bin width	0.1 - 0.9 Å[a,f]
ligand bin overlap	0.1 - 0.5 Å[a,f]
receptor bin overlap	0.1 - 0.5 Å[a,f]
nodes min/max	4/4[e]

a. For some test systems, the upper limit was not reached if docking results converged early.
b. Set to three, because as the size of a random match increases, the least-squares superposition procedure increasingly biases the orientation toward the centroid of the site points.
c. Set large enough to exclude atoms sharing a covalent bond.
d. Minimum of four chosen so that chirality could be used to filter matches. Maximum of ten is somewhat inconsistent with value chosen for BGM, but we presume any effects of this would be small.
e. Chosen as historical default.
f. Minimum value not zero because of a numerical instability of the algorithm.

listed in Table 3. The range of values in these tables correspond to timings of less than 0.1 seconds/molecule to more than 100 seconds/molecule on a modern workstation (see below).

For each docking run, several properties are computed which compare the results from any specific run to the best results from all runs combined (assumed to contain the

Table 3: Scoring and Optimization Parameters

type	force field
bump maximum	3[a]
dielectric	4r
grid spacing	0.3 Å
interpolation	trilinear
convergence criteria	0.1 kcal/mol[b]
maximum iterations	500[c]

a. Non-zero maximum allows some orientations with limited Van der Waals clashes with the receptor to be recovered by the minimizer.
b. A relatively tight convergence criteria was selected to reduce noise in the score evaluation, so that differences between methods were more directly attributable to differences in sampling. The rank correlation would be especially vulnerable to such noise.
c. A large iteration limit was also selected to reduce noise in the score evaluation by preventing the minimizer from terminating prematurely.

global minimum). These properties are summarized in Table 4. When these values are plotted versus time, the convergence of each property can be monitored. We assume that a better search algorithm will lead to more rapid convergence.

When considering the average behavior of each property, we compute both the usual mean and also a rank-weighted mean. The rank-weighted mean is more sensitive to the behavior of the top scoring molecules, which are of most interest in database screening runs. Though many kinds of weighting functions could be chosen for this purpose, we chose to use the reciprocal of the rank for convenience.

To make sure that our conclusions are generalizable, we analyzed the methods using five different receptor sites listed in Table 5. These sites were chosen from the list of

Table 4: Comparison Methods

Method	Definition	Equation	Rank Weighted Equation	Range
Average Relative Score	• For each molecule in the docking run, normalize its score by the best score it ever received in the site. • Then compute the average over the molecules in the set.	$\frac{\sum_{i=1}^{N} S_i/S_i^{min}}{N}$	$\frac{\sum_{i=1}^{N} \frac{S_i/S_i^{min}}{i}}{\sum_{i=1}^{N} \frac{1}{i}}$	[0,1]; unitless
Rank Correlation	• Assign a rank, y_i, to each molecule based on its best score in the docking run. • Then correlate y_i with the rank of each molecule based the best score it ever received in the site, x_i.	$\frac{\sum_{i=1}^{N} (x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{N} (x_i-\bar{x})^2}$	$\frac{\sum_{i=1}^{N} \frac{1}{i}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{N} \frac{1}{i}(x_i-\bar{x})^2}$	[-1,1]; unitless
Average RMS Error	• For each molecule in the docking run, compute the RMS error of its predicted orientation compared to that which received the best score for that site. • Then, compute the average over the molecules in the set	$\frac{\sum_{i=1}^{N} R_i}{N}$	$\frac{\sum_{i=1}^{N} \frac{R_i}{i}}{\sum_{i=1}^{N} \frac{1}{i}}$	[0,∞]; Å

complexes of high resolution, well-refined structure with a ligand having a well-defined binding position. They were also chosen to have very different-shaped binding sites. The chief features of each site are discussed in Table 5.

RESULTS AND DISCUSSION

Docking conditions

The test cases were prepared for docking in the standard way. Site points were constructed using the sphere generation accessory program of DOCK with default parameters (Kuntz et al., 1982). We selected the cluster of site points which occupied the binding site of the actual ligand in the crystal complex. Within this cluster, we merged the

UCSF LIBRARY

Table 5: Receptor Structures from PDB[1] Used for Test Systems

Code	Structure	Resolut ion	R factor	Site Description
121D	DNA dodecamer with Netropsin[2]	2.2 Å	0.198	Site is broad, presenting two continuous binding sites in the major and minor grooves of the DNA dodecamer. Highly polar.
1ULB	Purine nucleoside phosphorylase with Guanine[3]	2.75 Å	0.204	Site has two pockets; one is broad and centrally located, other where actual ligand binds is peripheral and solvent-excluded.
3DFR	Dihydrofolate reductase with NADPH and Methotrexate[4]	1.7 Å	0.152	Site has a deep, centrally-located binding pocket. Mixed polar and non-polar regions.
4FAB	Fab fragment with Fluorescein[5]	2.7 Å	0.215	Site is shallow with three pockets formed by the six hypervariable loops. Generally non-polar.
9HVP	HIV-1 protease with A-74704[6]	2.8 Å	0.182	Site is a long, narrow tube which completely penetrates protein. Mixed polar and non-polar regions.

1. Abola, Bernstein, Bryant, Koetzle, & Weng, 1987; Bernstein et al., 1977
2. Tabernero et al., 1993
3. Ealick et al., 1991
4. Bolin, Filman, Matthews, Hamlin, & Kraut, 1982
5. Herron, He, Mason, Voss, & Edmundson, 1989
6. Erikson et al., 1990

positions of tightly grouped site points using a 2 Å cutoff. The final number of site points used for each receptor ranged from 30 to 60.

Resource Usage

All docking calculations were performed on Silicon Graphics Indigo2 workstations equipped with 200 MHz R4400 processors and 128 Mb RAM, so timings are consistent among the different methods. Several weeks of computer time were required to complete all runs. All methods required approximately 13 Mb of RAM to store the scoring grids. The URT and URM methods required negligible additional memory for matching and orienting. BRM and SGM required up to 0.1 Mb of RAM for matching arrays.

BGM required 1 Mb of RAM for matching arrays.

Test Systems

Selected results for the 3DFR test system are presented in Figures 5, 6 and 7 to illustrate the type of data we collected. As shown in Figure 5, the weighted average score

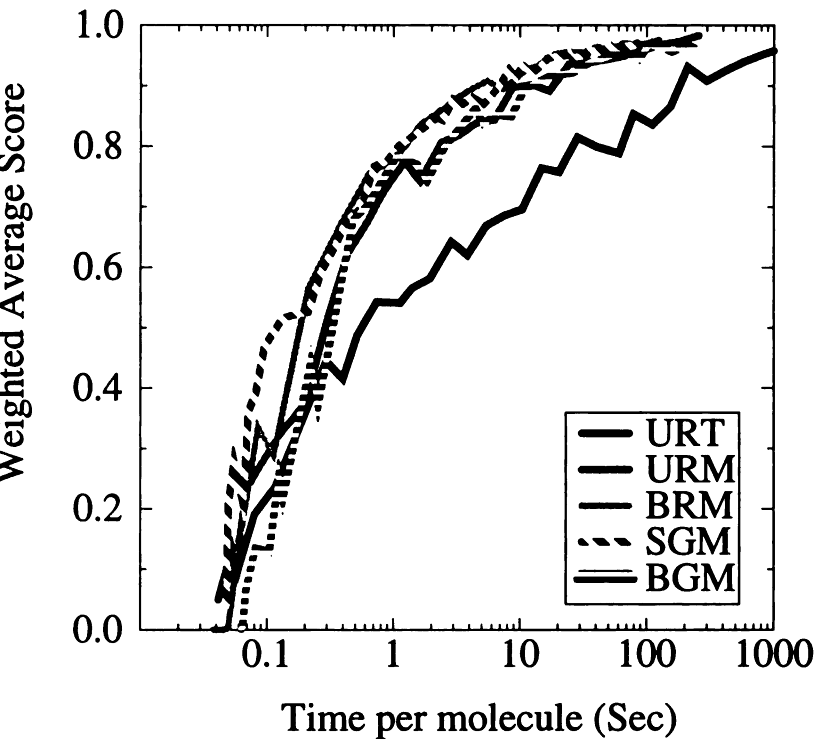


Figure 5. Weighted Average Score for Molecules Docked to 3DFR using SPHGEN site points. Each curve represents a search algorithm in Table 1. Each data point is a weighted average of the score for all molecules in a particular run using the equation in Table 4.

generally converges asymptotically to an optimum as sampling increases. The scores from all matching methods converge to within 90% of the optimum in about 10 seconds per molecule, whereas the URT method requires about 100 seconds per molecule. The weighted rank correlation in Figure 6 also shows convergent behavior, but with some interesting differences. It goes through much wider fluctuations, indicating that small

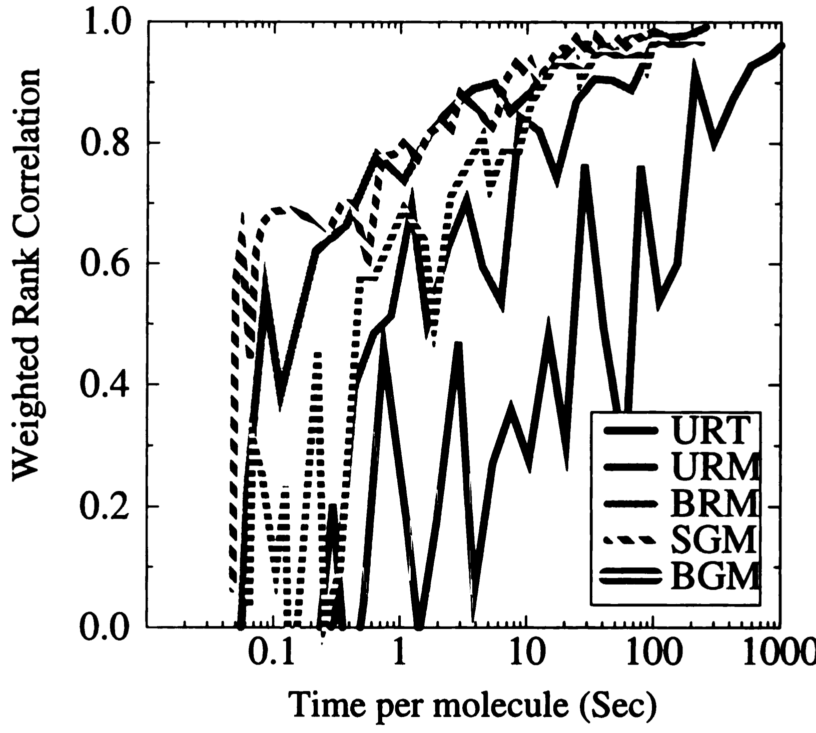


Figure 6. Weighted Rank Correlation for Molecules Docked to 3DFR using SPHGEN site points. Each data point is an average of the rank correlation for all molecules in a particular run using the equations in Table 4.

changes in score have large effects on the rankings of the top scoring molecules. It appears to discriminate among the different methods, selecting BRM and SGM as superior, BGM and URM as next best and URT as worst again. In particular, BRM and SGM both show a rapid initial rise, indicating that with very little sampling, these methods come closest to predicting the rankings of the top scoring molecules. The convergence of weighted RMSD in Figure 7 indicates how long it takes the different methods to reproducibly predict the same binding mode of the top scoring molecules. Though the two top performing methods converge in predicted score and ranking in about 10 seconds, they require about 500 seconds before they consistently predict the same binding mode. This result indicates that for these molecules in this site, several good scoring orientations must

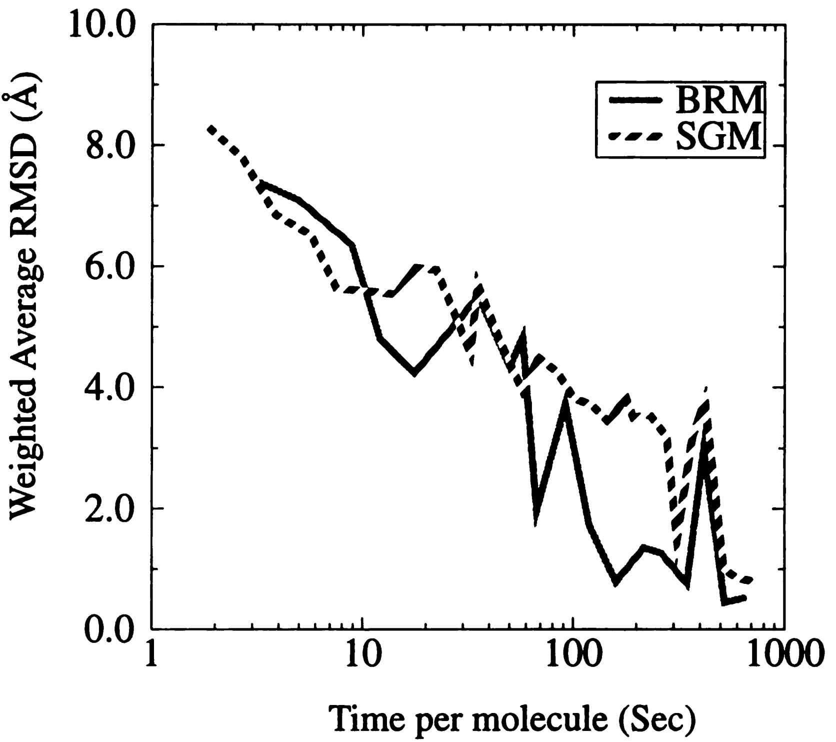


Figure 7. Weighted Average RMSD for Molecules Docked to 3DFR using SPHGEN site points. Only the two best search algorithms from Table 1 are presented. Each data point is an average of the RMSD for all molecules in a particular run using the equations in Table 4.

exist that are close in score but far in space. We found similar results for the other sites as well.

We found the weighted and unweighted forms of the average score and rank correlation to be the most relevant in assessing database screening performance. This gives us four measurements of five methods over five sites. Instead of presenting 100 different curves, we have condensed each curve into a single value: the convergence time, which represents the time at which the 90% threshold value is passed (and not recrossed). In Table 6, we present the convergence times along with the speed improvement factor of each method compared to the URT method.

Table 6: Convergence Properties of Search Methods.
For each sampling method and receptor site, we find the amount of sampling beyond which all values are within 90% of the maximum. With the time of URT as the reference, the relative speed factor of each method is reported in parenthesis. The geometric mean over all receptor site values is reported at bottom. Because of the large uncertainty in these values, all values are rounded to one significant digit.

6A. Unweighted Average Score

Site	Search Methods									
	URT		URM		BRM		SGM		BGM	
121D	200	(1x)	10	(20x)	20	(10x)	10	(20x)	8	(30x)
1ULB	200	(1x)	10	(20x)	20	(8x)	20	(10x)	8	(20x)
3DFR	100	(1x)	9	(10x)	10	(10x)	10	(10x)	20	(6x)
4FAB	30	(1x)	7	(4x)	8	(4x)	7	(5x)	7	(4x)
9HVP	200	(1x)	7	(30x)	5	(40x)	5	(40x)	6	(30x)
Mean	100	(1x)	9	(10x)	10	(10x)	9	(10x)	9	(10x)

6B. Rank-weighted Average Score

Site	Search Methods									
	URT		URM		BRM		SGM		BGM	
121D	400	(1x)	10	(40x)	6	(80x)	3	(100x)	8	(50x)
1ULB	1000	(1x)	60	(20x)	30	(40x)	20	(70x)	30	(40x)
3DFR	200	(1x)	20	(9x)	10	(20x)	8	(30x)	20	(10x)
4FAB	90	(1x)	7	(10x)	4	(20x)	4	(20x)	7	(10x)
9HVP	200	(1x)	7	(30x)	2	(100x)	4	(40x)	5	(40x)
Mean	300	(1x)	20	(20x)	7	(40x)	6	(50x)	10	(30x)

Score Convergence

With respect to the unweighted average score in Table 6A, all matching methods show roughly equivalent convergence, and outperform URT by a factor of ten. Therefore, on average, site points provide a much more succinct description of the active site than the smallest enclosing box, especially when searching a site that is large or difficult to define a

UCSF LIBRARY

6C. Unweighted Rank Correlation

Site	Search Methods									
	URT		URM		BRM		SGM		BGM	
121D	400	(1x)	20	(20x)	10	(30x)	7	(60x)	10	(40x)
1ULB	1000	(1x)	60	(20x)	40	(20x)	30	(30x)	80	(10x)
3DFR	400	(1x)	50	(9x)	20	(20x)	10	(30x)	40	(10x)
4FAB	200	(1x)	30	(7x)	10	(20x)	20	(7x)	50	(3x)
9HVP	700	(1x)	20	(40x)	8	(90x)	10	(60x)	20	(50x)
Mean	500	(1x)	30	(10x)	20	(30x)	20	(30x)	30	(20x)

6D. Rank-weighted Rank Correlation

Site	Search Methods									
	URT		URM		BRM		SGM		BGM	
121D	400	(1x)	20	(30x)	10	(40x)	7	(60x)	20	(30x)
1ULB	1000	(1x)	60	(20x)	20	(60x)	20	(70x)	80	(20x)
3DFR	600	(1x)	100	(6x)	10	(40x)	10	(40x)	20	(30x)
4FAB	400	(1x)	10	(30x)	3	(100x)	4	(90x)	10	(30x)
9HVP	300	(1x)	7	(40x)	2	(100x)	4	(60x)	3	(80x)
Mean	500	(1x)	20	(20x)	7	(70x)	8	(60x)	20	(30x)

priori. Since the site points were generated based on general considerations of shape, this result should be generalizable to other "negative image" techniques, like the shape-based critical point methods of Lin, Nussinov, Fischer, and Wolfsen (1994), and the energetic probe methods of Reynolds, Wade, and Goodford (1989). The rank-weighted average score in Table 6B shows more discrimination among the matching methods. While the two uniform random methods (URT and URM) had more difficulty converging with the top-scoring molecules, BRM and SGM actually converged more quickly. This implies that trying a uniform number of orientations for the molecules in the database is inefficient with respect to processing the top-scoring molecules. BGM performed better than URM,

but not as well as either BRM or SGM.

Rank Convergence

The convergence of the rank correlation further confirms the differences between methods. The unweighted rank correlations in Table 6C again show a general 10 to 30 fold advantage of using site points to dock the molecules. Interestingly, the time required to get the rank correct is 2 to five fold greater than the time to get the average score (Table 6A) correct. The weighted rank correlations in Table 6D also further discriminate among the methods. URM and BGM still show the 20 to 30 fold advantage over URT. BRM and SGM again outperform by 60 to 70 fold over URT. Of all measures, this last one is arguably the most relevant to database screening, since absolute scores are generally not used as strict cutoffs, but instead the rankings are used to select some subgroup of molecules whose number is amenable to further processing. Most often, it is the top-scoring subgroup of most interest, so using a rank-weighted correlation should focus our attention on how the methods are treating this particular subgroup of molecules. Therefore, it appears that of the methods investigated here, BRM and SGM are the best suited for database screening.

UCSF LIBRARY

BGM versus SGM

The single graph matching method clearly outperforms the existing bipartite graph matching method by up to two-fold in speed. This result may appear counter-intuitive, because SGM is exhaustive whereas BGM uses heuristics to speed the search. However, the precomputing of the adjacency matrix and the rapid processing of the adjacency lists show that reformulating the problem into an efficient form can be just as effective as using heuristics. SGM has the additional advantage of requiring fewer fundamental matching

parameters than BGM (Table 2).

BRM versus SGM

Why does a simplistic random matching algorithm, BRM, perform so competitively with the distance matching algorithm, SGM, and even outperform BGM? Does this suggest that distance matching is an unnecessarily complicated solution to docking? We seek to resolve this question by breaking the problem into three parts.

7. What is the disadvantage of sampling each molecule uniformly?
8. Why does distance matching sample molecules non-uniformly?
9. Is an orientation from random matching just as good as one from distance matching?

First, spending the same time on each molecule may result in spending too little time on the better scoring molecules. One feature of the force field scoring function is that it tends to favor larger molecules (Miller et al., 1994). For the set of molecules we used in this study, we have plotted the best score for each molecule against its size in Figure 8. Though there is some trend, the correlation coefficient is not large. A stronger trend exists in Figure 9, relating molecule size to bump filtering. Large molecules have a greater propensity to bump into receptor atoms when oriented in the site. Since we use a bump filter in DOCK to discard poor orientations before the more computationally expensive scoring and optimization steps, we are more likely to discard an orientation of a large molecule than that of a small molecule. Forcing a uniform number of matches per molecule would then result in a size-biased attrition through the bump filter and, over-all, spending less time on the potentially better scoring, larger molecules.

Second, nonuniform sampling arises in distance matching because the number of matches is related to the number of internal distances that the ligand has in common with the site points. Larger molecules have more internal distances, and so will tend to have more in common. Therefore, larger molecules tend to generate more matches than smaller

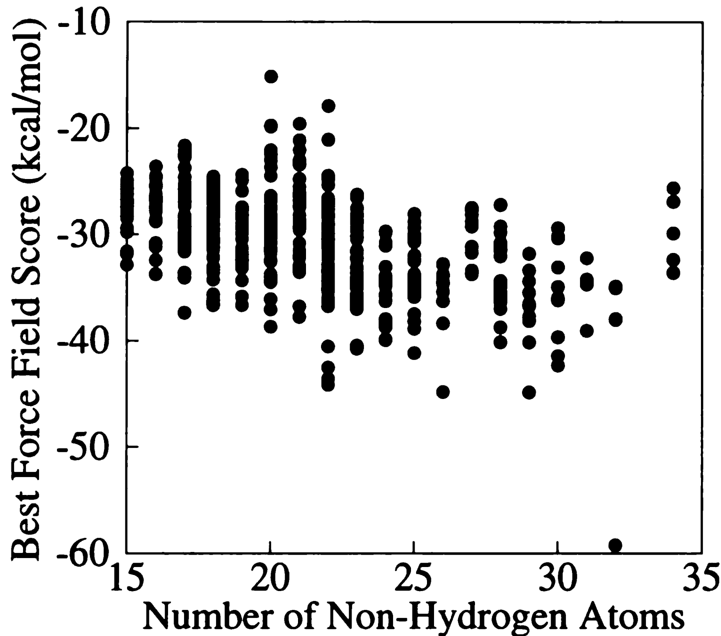


Figure 8. How Molecule Size affects Force Field Score. The best force field score for each molecule is plotted versus the number of non-hydrogen atoms in the molecule. The plots for each receptor are pooled into this single plot to show the overall trend. Fitting a line to these data yields an R^2 value of 0.24.

molecules. However, a molecule which is not larger, but is more similar in shape to the binding site will also have more internal distances in common. Therefore, distance matching will spend more time on molecules that are complementary in shape to the receptor. Thus, the BRM method benefits from using distance matching to determine the number of orientations to try simply because it will bias its efforts toward the larger and/or more complementary molecules.

Third, an orientation from random matching is on average not as good as one from distance matching. We can check the quality of these orientations by examining how they survive the bump filter as depicted in Figure 9. The orientations from distance matching are at least ten times more likely to make it past this filter, so they are indeed superior. To understand why this superiority does not translate directly into faster docking, we must

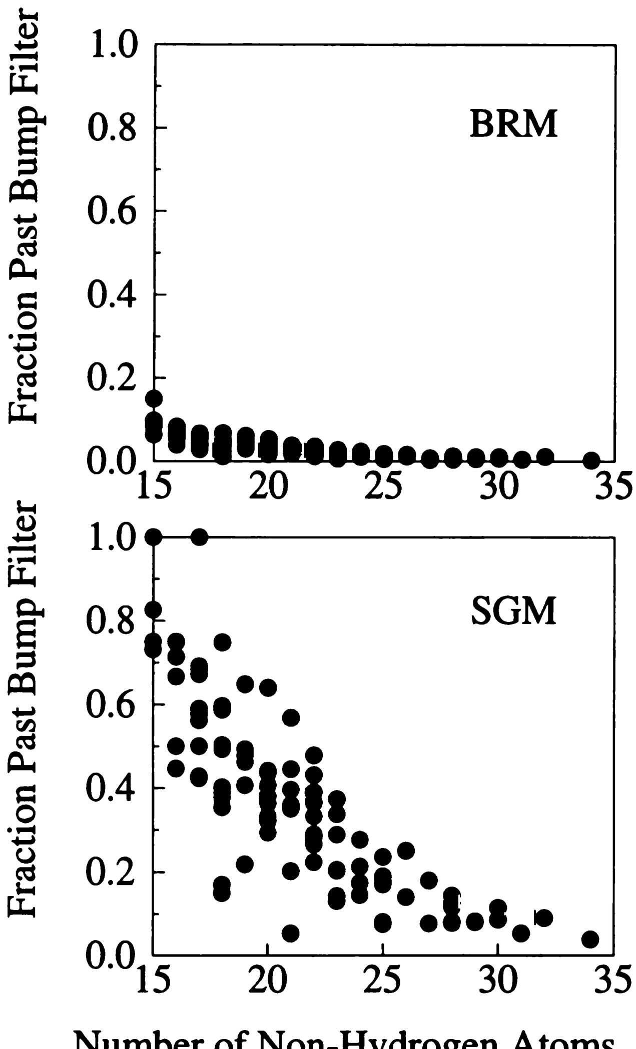


Figure 9. How Molecule Size Affects Bump Filtering. In a given docking run, we compute the fraction of orientations that pass the bump filter for each molecule, then plot this value against the size of the molecule. (Top) BRM method. Average filter rate is 0.031 . (Bottom) SGM method. Average filter fate is 0.36.

consider the computational bottle-neck of the current implementation of DOCK. If the early docking steps -- matching, orienting and bump filtering -- were to consume a dominant portion of the total cpu time, then SGM would be up to ten times faster than BRM. On the other hand, if the later docking steps -- scoring and optimization -- were to consume a dominant portion of the total cpu time, then BRM would be equivalent in speed to

SGM. With the current implementation of the minimizer, the scoring and optimization steps indeed consume an overwhelming portion of cpu time (>90%), so the ability of distance matching to form high quality orientations is for now unrewarded. BRM is kept competitive because, in concert with heavy pruning by the bump filter, it forms orientations that are suitable starting points for optimization.

BRM versus BGM

The fact that BRM outperforms BGM by up to a factor of two, points out the critical importance of how processing time is allocated among the different molecules in the database. For maximum efficiency, a search algorithm must bias its efforts toward the most highly-ranked molecules. BGM indeed has such a bias built in, but the heuristics have the effect of reducing the magnitude of bias inherent to the unrestrained matching found in SGM and BRM.

Future Directions

It appears sensible to incorporate the single graph matching technique into version 4.0 of DOCK since its results are of high quality and its potential for speed improvement is high. The speed gains might arise from additional orientation filtering or by fundamental improvements in the optimization technique. The new matching algorithm will provide a solid algorithmic foundation on which to base further development of molecular docking, including the addition of sophisticated search constraints like chemical labels and critical clusters, as well as the explicit treatment of ligand flexibility.

Until a faster scoring and optimization method is implemented, it may be useful to preprocess the set of orientations generated by matching. For instance, an implementation of degeneracy checking has been tried in which similar orientations are removed at the

matching stage (Gschwend & Kuntz, 1996). Since orienting is also relatively facile, it implies that the degeneracy checking could instead be performed as an RMS deviation calculation between real orientations. Rarey, et. al. present a rapid RMSD evaluation technique based directly on the rotation matrices (Rarey et al., 1996) which would be applicable. It might also be possible identify a unique property of orientations that lead to the best scores upon minimization (e.g. degree of surface overlap with receptor). We intend to further study the nature of orientations generated by BRM that pass the bump filter, to see why they manage to score so well upon minimization. Based upon this knowledge, a filter could be constructed that enriches the set of orientations generated by regular matching prior to minimization.

An additional avenue for improvement would be to try alternative methods to generate site points. The current shape-based site points may not be entirely consistent with a force-field-based scoring function (Miller et all., 1994). Including force field considerations during site point construction, might make the site points more relevant as points on which to position ligand atoms, thereby increasing the quality of docked orientations.

We are working to extend the current docking protocol to include ligand and receptor flexibility. Ligand flexibility can be incorporated in several ways. The most straightforward approach is to dock multiple conformations of the ligand separately (Miller et al., 1994). A potentially more efficient method is to use distance geometry to build a ligand conformation that fits a subset of receptor site points (Smellie et al., 1991). Another viable option is the "divide and conquer" strategy, in which a flexible molecule is broken into rigid fragments, the fragments are docked independently and the molecule is rebuilt from adjacent fragment orientations (Bohm, 1992; DesJarlais et al., 1986; Eisen et al., 1994).

UCSF LIBRARY

Limited receptor flexibility is being investigated during the step of score evaluation by superimposing multiple receptor conformers on a single score potential grid (Knegtel, Kuntz, & Oshiro, 1997).

CONCLUSION

We evaluated various search algorithms for automated molecular docking that range in complexity from purely random to site point-directed, graph-theoretical matching methods. Our basis of comparison was how quickly each docking method could correctly score and rank a database of molecules. Over a broad range of active site environments, it is at least tenfold more efficient to use a collection of site points to describe the active site search volume than to use the smallest enclosing box. Using graph theoretical matching techniques boosts this relative efficiency higher. The bipartite graph matching used in the current DOCK version 3.5 improves efficiency up to 30-fold. The single graph matching proposed for DOCK version 4.0 improves efficiency up to 60-fold. Since single graph matching is not only faster, but also less complicated than the bipartite graph matching used in version 3.5, we feel it will be an important advance in the docking technology.

REFERENCES

Abola, E. E., Bernstein, F. C., Bryant, S. H., Koetzle, T. F., & Weng, J. (1987). Crystallographic databases: information content, software systems, scientific applications. In F. H. Allen, G. Bergerhoff, & R. Seivers (Eds.), Data Commission of the International Union of Crystallography (pp. 107-132).

Bernstein, F. C., Koetzle, T. F., Williams, G. J. B., Meyer Jr., E. F., Brice, M. D., Rodgers, J. R., Kennard, O., Shimanouchi, T., & Tasumi, M. (1977). Journal of Molecular Biology, 112, 535.

Bohm, H. J. (1992). The computer program ludi - a new method for the denovo design of enzyme inhibitors. Journal of Computer-Aided Molecular Design, 6(1), 61-78.

Bolin, J. T., Filman, D. J., Matthews, D. A., Hamlin, R. C., & Kraut, J. (1982). Crystal structure of escherichia Coli and lactobacillus Casei dihydrofolate reductase refined at 1.7 angstroms resolution. I. General features and binding of methotrexate. Journal of Biological Chemistry, 257, 13650.

Bron, C. & Kerbosch J. (1973). Finding all cliques of an undirected graph. Communications of the ACM, 16(9), 575-577.

UCSF LIBRARY

DesJarlais, R. L. & Dixon, J. S. (1994). A shape- and chemistry-based docking method and its use in the design of hiv-1 protease inhibitors. Journal of Computer-Aided Molecular Design, 8 (3), 231-242.

DesJarlais, R. L., Sheridan, R. P., Dixon, J. S., Kuntz, I. D., & Venkataraghavan, R. (1986). Docking Flexible Ligands to Macromolecular Receptors by Molecular Shape. Journal of Medicinal Chemistry, 29, 2149.

DesJarlais, R. L., Sheridan, R. P., Seibel, G. L., Dixon, J. S., Kuntz, I. D., & Venkataraghavan, R. (1988). Using shape complementarity as an initial screen in designing ligands for a receptor binding site of known three-dimensional structure. Journal of Medicinal Chemistry, 31(4), 722-729.

Ealick, S. E., Babu, Y. S., Bugg, C. E., Erion, M. D., et al. (1991). Application of crystallographic and modeling methods in the design of purine nucleoside phosphorylase inhibitors. Proceedings of the National Academy of Sciences of the United States of America, 88(24), 11540-11544.

Eisen, M. B., Wiley, D. C., Karplus, M., & Hubbard, R. E. (1994). Hook - a program for finding novel molecular architectures that satisfy the chemical and steric requirements of a macromolecule binding site. Proteins-Structure Function and Genetics, 19(3), 199-221.

UCSF LIBRARY

Erickson, J., Neidhart, D. J., Van Drie, J., Kempf, D. J., Wang, X. C., Norbeck, D. W., Plattner, J. J., Rittenhouse, J. W., Turon, M., Wideburg, N., Kohlbrenner, W. E., Simmer, R., Helfrich, R., Paul, D. A., & Knigge, M. (1990). Design, activity, and 2.8 a crystal structure of a c2 symmetric inhibitor complexed to hiv-1 protease. Science, 249(4968), 527-533.

Ferro, D. R. & Hermans, J. (1977). Acta Crystallographica, A33, 345.

Gschwend, D. A. & Kuntz, I. D. (1996). Orientational sampling and rigid-body minimization in molecular docking revisited - on-the-fly optimization and degeneracy removal. Journal of Computer-Aided Molecular Design, 10(2), 123-132.

Herron, J. N., He, X. M., Mason, M. L., Voss, E. W., et al. (1989). 3-dimensional structure of a fluorescein fab complex crystallized in 2-methyl-2,4-pentanediol. Proteins-Structure Function and Genetics, 5(4), 271-280.

Knegtel, R. M. A, Kuntz, I. D., & Oshiro, C. M. (1997). Molecular docking to ensembles of protein structures. Journal of Molecular Biology, 266(2), 424-440.

Kuhl, F. S., Crippen, G. M., & Friesen, D. K. (1984). A combinatorial algorithm for calculating ligand binding. Journal of Computational Chemistry, 5, 24.

Kuntz, I. D. (1992). Structure-based strategies for drug design and discovery.

UCSF LIBRARY

Science, 257 (5073), 1078-1082.

Kuntz, I. D., Blaney, J. M., Oatley, S. J., Langridge, R., & Ferrin, T. E. (1982). A geometric approach to macromolecule-ligand interactions. Journal of Molecular Biology, 161, 269.

Lin, S. L., Nussinov, R., Fischer, D., & Wolfson, H. J. (1994). Molecular surface representations by sparse critical points. Proteins-Structure Function and Genetics, 18(1), 94-101.

Meng, E. C., Gschwend, D. A., Blaney, J. M., & Kuntz, I. D. (1993). Orientational sampling and rigid-body minimization in molecular docking. Proteins-Structure Function and Genetics, 17(3), 266-278.

Meng, E. C., Shoichet, B. K., & Kuntz, I. D. (1992). Automated docking with grid-based energy evaluation. Journal of Computational Chemistry, 13(4), 505-524.

Miller, M. D., Kearsley, S. K., Underwood, D. J., & Sheridan, R. P. (1994). Flog - a system to select quasi-flexible ligands complementary to a receptor of known three-dimensional structure. Journal of Computer-Aided Molecular Design, 8(2), 153-174.

Rarey, M., Wefing, S., & Lengauer, T. (1991). Placement of medium-sized molecular fragments into active sites of proteins. Journal of Computer-Aided Molecular Design,

UCSF LIBRARY

10(1) 41-54.

Reynolds, C. A., Wade, R. C., & Goodford, P. J. (1989). Identifying targets for bioreductive agents - using grid to predict selective binding regions of proteins. Journal of Molecular Graphics, 7(2), 103+.

Rusinko, A., Sheridan, R. P., Nilakantan, R., Haraki, K. S., Bauman, N., & Venkataraghavan, R. (1989). Using concord to construct a large database of 3-dimensional coordinates from connection tables. Journal of Chemical Information and Computer Sciences, 29(4), 251-255.

Shoichet, B. K, Bodian, D. L., & Kuntz, I. D. (1992). Molecular docking using shape descriptors. Journal of Computational Chemistry, 13(3), 380-397.

Smellie, A. S., Crippen, G. M., & Richards, W. G. (1991). Fast drug-receptor mapping by site-directed distances - a novel method of predicting new pharmacological leads. Journal of Chemical Information and Computer Sciences, 31(3), 386-392.

Tabernero, L., Verdaguer, N., Coll, M., Fita, I., Van der Marel, G. A., Van Boom, J. H., Rich, A., & Aymami, J. (1993). Molecular structure of the a-tract dna dodecamer d(cgcaaatttgcg) complexed with the minor groove binding drug netropsin. Biochemistry, 32(33), 8403-8410.

UCSF LIBRARY

Chapter 3: Search Methods for Flexible Molecule Docking

SUMMARY

We present an extension of the docking algorithm to accommodate flexible molecules. A rigid portion of the molecule is docked using the orientation search presented in Chapter 1. Flexible portions are reattached incrementally with a simultaneous torsion search and relaxation. The partial configurations are pruned at each cycle of reattachment according to diversity and affinity. The method successfully reconstructs the binding modes of X-ray crystal complexes. It is accurate (docked positions < 2 Angstroms from X-ray position) and fast (computation times <= 180 seconds). The method is practical for database processing. It can accurately rank a random set of molecules for binding to dihydrofolate reductase within 100 seconds per molecule. It is more effective than rigid docking alone, or in combination with a random conformation search.

INTRODUCTION

The binding of a small molecule to a macromolecule can involve many degrees of freedom. The rigid body degrees of freedom -- rotation and translation -- are fundamental to the docking process. Searching rigid body space is described in Chapter 1. The conformation space of a small molecule can also play an important in the docking of flexible ligands. Searching ligand conformation space is described in this chapter.

A similarly difficult problem is the consideration of the conformation space of the macromolecule as it accommodates the incoming ligand. Methods have been developed to allow discrete residues to move. A highly pruned systematic search of these limited

receptor conformations uses the A* and Dead End Elimination heuristic (Leach, 1994). As well, multiple receptor conformations can be treated in a "first pass" manner using ensemble grids (Knegtel, Kuntz, & Oshiro, 1997).

Many techniques have been used to search ligand conformation space. The most promising techniques are based on simulated annealing (Morris, Goodsell, Huey, & Olson, 1996), genetic algorithm (Clark & Ajay, 1995; Judson, Jaeger, & Treasurywala, 1994; Oshiro, Kuntz, & Dixon, 1995) and incremental construction. The incremental construction methods are based on docking a rigid portion of the molecule first, and reconstructing the flexible parts in fashion complementary to the local receptor environment. Leach and Kuntz (1992) were seminal in presenting this technique. Other workers have elaborated upon this method (Makino & Kuntz, 1997; Mizutani, Tomioka, & Itai, 1994; Rarey, Kramer, Lengauer, & Klebe, 1996; Welch, Ruppert, & Jain, 1996).

In the method of Leach and Kuntz (1992), the set of shape-based site points is supplemented with hydrogen-bond based site points. The site points are used to drive the orientation search and also to guide the flexible search. The flexible portions are reattached one torsion at a time in a depth-first fashion. The torsion positions are assigned from a rotamer library and are tweaked to allow ligand atoms to interact with nearby complementary site points. Relatively few constraints are placed on the conformation search, so it can grow in a combinatorial fashion.

Mizutani, et al. (1994) describe a flexible docking routine, ADAM, which uses hydrogen bonding patterns to form docking models. The smallest subset of the ligand which contains all hydrogen bonding groups is processed first. A systematic conformation search is performed. All conformers are assessed for compatibility with every docking

UCSF LIBRARY

model. A systematic search combined with energy evaluation is used to fit the remaining portion of the molecule. Because systematic searches are used, the user has little control over the amount of sampling.

Welch, et al. (1996) recently describe a flexible docking routine, Hammerhead, in which the conformation search is more tightly linked to site point positions. The molecules are divided into three-torsion segments. Terminal segments are somewhat arbitrarily chosen as the starting positions. They are initially placed by matching ligand and receptor surface points. A breadth-first, N-best search is used for the remaining segments, which are placed by matching additional ligand and receptor surface points. The N best configurations are used to seed each additional cycle of construction. This method was shown to redock crystal complexes accurately and to screen a database of ligands efficiently. It uses multiple levels of pruning in order to reduce computation time. Each type of pruning has user-adjustable parameters which might become difficult to set for a variety of docking sites.

Rarey, et al. (1996) recently present a flexible docking routine, FlexX, which is guided directly by precomputed site points. The site points are constructed based on receptor hydrophobic patches and hydrogen-bonding patterns. The starting segment of the ligand must be selected by the user and can include rotatable torsions. It is docked to the site based on site point positions. A breadth-first, N best search is used to reattach adjacent segments, which are attached and pruned based on interactions with nearby site points, and further filtered by RMSD clustering. This method requires not only the beginning segment, but also each added segment, must interact with complementary site points. This can be a problem when such well-defined interactions are missing from the original

UCSF LIBRARY

binding mode. In addition, the beginning segment must be chosen manually by the user. No general rules are presented by the authors for the user to follow to accomplish this task. This method was shown to successfully dock a selected set of crystal complexes.

We have developed a more distinctly two-step algorithm for flexible docking for use in DOCK version 4.0. A flowchart of the algorithm is presented in Figure 10. The

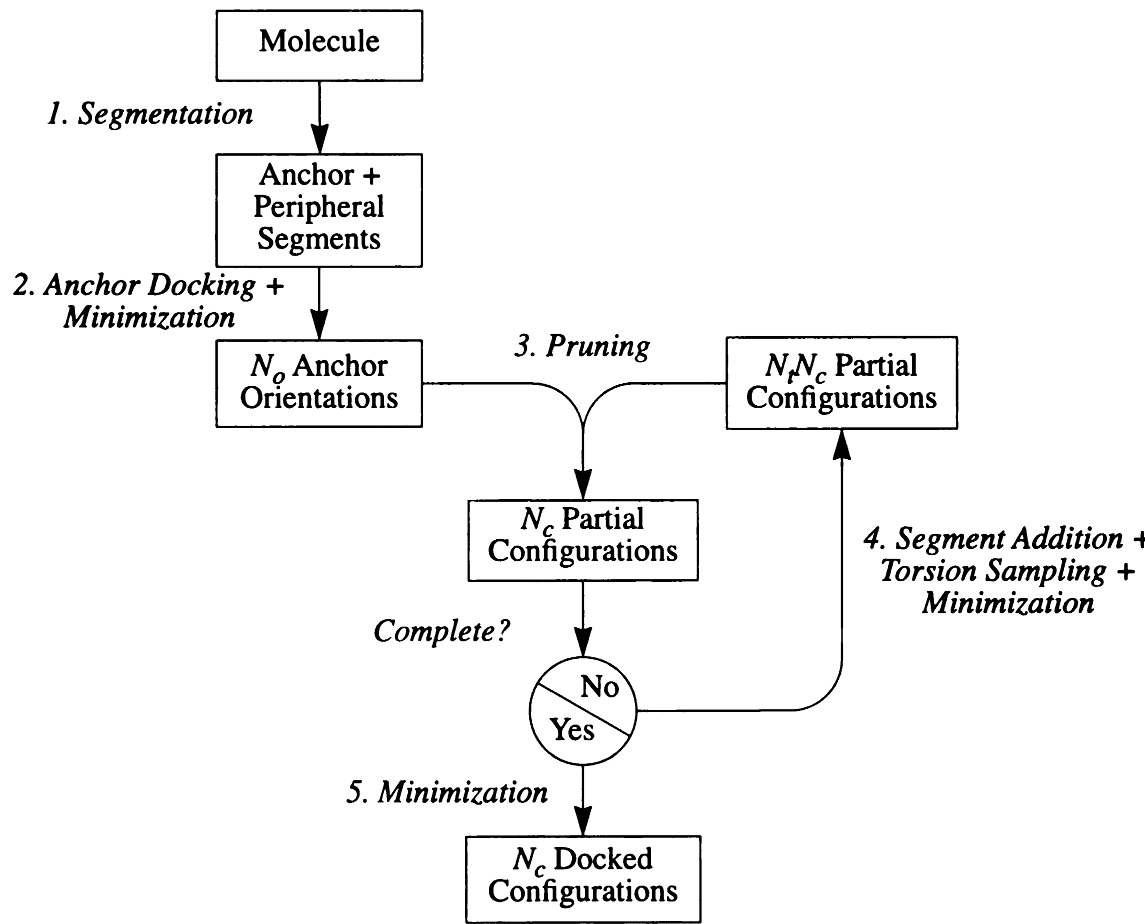


Figure 10. Flowchart of Flexible Docking Algorithm.

molecule is divided into rigid segments. The beginning segment may be selected manually, or automatically based on size -- multiple segments can be tried in this way. The beginning, or anchor, segment is docked as a rigid unit, using the regular matching algo-

rithm to precomputed site points. The remaining segments are reattached in a breadth-first manner and pruned according to score, rather than site point interaction. The breadth of the search is directly controlled by selecting the number of seed configurations to carry over to the next cycle. A wide variety of seed configurations is selected by clustering the partially built configurations. Score optimization helps resolve minor clashes and enrich the torsion angle search. Since the score alone is used during the conformation search, areas of low complementarity can be easily traversed where lack of site points might cripple the other algorithms. The amount of sampling is easily controlled by specifying the number of orientations in the orientation search and the number of conformations in the conformation search.

The incorporation of scoring function evaluation as part of the search procedure raises the importance of scoring function accuracy. In previous efforts, the rigid body search was considered sufficient to generate many binding orientations, including nearly correct binding modes. As an independent effort, a scoring function would be used to fish out the correct modes. With the use of very high sampling, the possibility of including the correct binding mode in the incremental construction strategy still exists, but an inaccurate scoring function will make this very unlikely. The work presented here must therefore be considered in the context of such hazards. The importance of continued development of more accurate scoring functions cannot be emphasized enough. The incremental construction strategy places a constraint on the nature of scoring functions that can be applied. Since the strategy must evaluate the interactions of partially-built molecules, the scoring function must be atom pairwise decomposable.

The flexible docking method described here will be evaluated in several ways. A

UCSF LIBRARY

test set of high resolution X-ray crystal complexes will be used to test docking accuracy and speed. Accuracy will be measured by the RMS distance of the nearest docked position for a dock run using a high level of sampling. Speed will be measured by how much time or sampling is required to generate a docked position within 2 Angstroms of the X-ray position.

The method will also be evaluated as a tool for database processing. A small randomly selected database of molecules will be docked to dihydrofolate reductase. Over a range of sampling conditions, the accuracy of the predicted rankings will be measured. Several benchmark algorithms will be used for reference.

METHODS

Molecule Segmentation

The flexible docking algorithm involves a significant amount of molecule preprocessing before actual docking, which is presented in Table 7. Since it not computationally intensive, ligand segmentation can be performed at run time and requires no modification of the ligand database.

The ring detection step is necessary to isolate ring bonds. Ring flexibility will not be considered directly in the algorithm presented here. For molecules in which ring flexibility plays an important role, different ring conformations can be generated by other modeling programs and treated as independent instances in an input molecule database by DOCK. Rings are detected using a depth-first algorithm which recursively traverses atom connectivity starting from any seed atom. During the expansion of the recursion tree, atoms are identified according to the numeric level of recursion at which they are encoun-

UCSF LIBRARY

Table 7: Ligand Segmentation Algorithm

Step	Description	Sample processing of Methotrexate
Ring Detection	A depth-first, single pass search of atom connectivity is performed to detect rings. Ring bonds are excluded from rotation.	
Rotatable Bond Detection	From the remaining set of bonds, rotatable bonds are identified according to definitions contained in user-editable text files.	
Rigid Segment Identification	The location of each rotatable bond is used to partition the molecule into rigid segments.	
Anchor Segment Identification	The anchor segment is selected either manually or automatically. The anchor defines the molecule center. All other segments are arranged into concentric layers.	
Segment Repartitioning	Each rotatable bond is assigned to the segment of the outer atom. The outer atom is transferred to the inner segment since it is fixed during bond rotation.	

tered. If the depth-first search encounters an atom with a lower recursion identifier, the

UCSF LIBRARY

intervening bond is identified as a ring bond. The lowest recursion identifier encountered is stored. During contraction of the recursion tree, if the lowest recursion identifier encountered from traversal of a particular bond is lower than the current level of recursion, then the bond is identified as a ring bond. This method grows linearly with molecule size and any size or number of rings is detected.

Once ring bonds have been isolated, all other bonds are compared to a library of flexible bond definitions stored in an editable text file. Each rotatable bond is assigned its allowed torsion positions which are also contained in an editable text file. The molecule is then divided into rigid segments, which can include ring systems or be as small as a methylene group. The anchor segment can be selected manually or automatically based on segment size. In automatic mode, either the largest segment is used, or all segments which have a required number of heavy atoms.

The last step in preprocessing is repartitioning of the segments based on the location of the anchor segment. This process is repeated if multiple anchor segments are used. Each rotatable bond is assigned to a segment. Atoms are transferred among adjacent segments so for each rotatable bond, the associated segment includes all atoms that are directly rotated. Careful inspection of the color figure for this step in Table 7 will reveal how atoms and bonds are grouped after repartitioning. As a final step, segments are organized into concentric layers about the anchor.

Docking Algorithm

Anchor Docking

The docking process illustrated in Figure 10 is divided cleanly into two steps: the rigid anchor orientation search (step 2) and the peripheral segment conformation search

UCSF LIBRARY

(steps 3 and 4). The orientation search of the anchor segment provides a rapid, preliminary search of binding modes. The matching algorithm has been modified slightly from what was presented in Chapter 1. In the original matching algorithm, the user specifies geometric parameters, and the algorithm exhaustively enumerate all orientations which fit the parameters. Now the user can specify the number of orientations, N_o in Figure 10, and the algorithm will perform a series of matching operations in which the distance tolerance is steadily increased until N_o matches have been formed. Since matching is completely exhaustive within the bounds of the distance tolerance, the matches produced by a preceding cycle of matching are a complete subset of the matches in the current cycle, and are removed after matching but before generating molecule orientation orientations. This change was made because for a given distance tolerance, the number of matches is strongly dependent on the number of molecule atoms included in matching. Since anchor segments can vary greatly in size, the updated matching method ensures that smaller anchor segments sample orientation space sufficiently. The computational demand of anchor docking depends linearly on N_o.

UCSF LIBRARY

PRUNING

The peripheral segment conformation search (steps 3 and 4 in Figure 10) is seeded by the anchor orientations generated in the rigid docking step. The orientations are pruned down to a number that is under user control (N_c in step 3). Pruning is also performed on the partially built configurations for every cycle of flexible segment addition.

SEGMENT ADDITION

The next step is segment addition (step 4). Segments are added to the pruned partial configuration in an orderly fashion, which starts with the innermost layer and proceeds

outward. A single segment is added multiple times, forming a new partial configuration for each torsion position allowed by the adjoining bond. If the bond can adopt N_t torsion positions, then $N_c N_t$ new configurations will be built in a cycle. Each new configuration is minimized. If the newly added bond is flagged in the definition file as minimizable (bonds with double bond character are excluded), then the torsion position is adjusted to minimize the score. Additional bonds may be included from inner layers of segments to help prevent partial configurations from getting trapped in dead-end constructions. The position of the anchor may also be adjusted to relieve strain during the construction process. These minimization options are under user control. The cycles of segment addition and pruning continue until all configurations are complete molecules. The computational time required by segment addition grows linearly with N_c, N_t, and N_s (the number of peripheral segments).

Minimization

After all cycles of segment addition are complete, the pruned set of complete configurations is minimized (step 5) to relieve any strain developed during the construction process. All minimizable torsions are minimized along with the anchor position using the minimization protocol discussed below.

Pruning Method

The pruning performed at each cycle of segment addition plays an important role in overall performance. Several methods of pruning were explored. The most straightforward protocol tested was the selection of the top N_c scoring configurations as seeds for the next cycle of addition. A second protocol tested was the selection of the most diverse configurations. This method involves complete-linkage, hierarchical clustering based on the

UCSF LIBRARY

RMS distance between configurations. When two configurations are clustered, the better scoring one is retained to represent the cluster. Clustering proceeds until N_c clusters have been formed. The third protocol tested was the selection of the best scoring and the most diverse configurations. This clustering method is based on the RMS distance between configurations divided by the rank of the poorer scoring configuration. The computational demand of cluster-based pruning depends on the square of N_o and linearly on the difference of N_o and N_c.

Scoring

INTERMOLECULAR TERMS

The intermolecular interaction of the small molecule with the macromolecule is described with the non-bonded terms of the AMBER molecular mechanic potential (Pearlman, et al., 1995) precomputed on a three-dimensional grid (Meng, Shoichet, & Kuntz, 1992). In the current version of DOCK, the user can select the following options: whether a united atom or all atom model is used, the exponents of the attractive and repulsive Lennard-Jones terms, and the dielectric constant. During the construction of flexible molecules, the interactions of the partially built portions is modeled to a first level of approximation by leaving the VDW terms and partial charges unperturbed. Since functional groups are often split into different segments, this approximation may cause short-term inaccuracies in the modeling of the interactions with these partially-built functional groups. Some further study in how to treat these interactions is warranted.

INTRAMOLECULAR TERMS

Because of the conformational freedom given to flexible molecules during construction, some treatment of intramolecular energy is necessary. The chief concern is to

UCSF LIBRARY

prevent such gross violations of chemical modeling as internal clashes. To this end, the non-bonded Lennard-Jones and Coulombic terms are computed for atoms in different rigid segments. Atoms within a segment can be excluded, since their contribution is a constant. Since only torsional degrees of freedom are explored, no bond length or angle terms are included. The molecular geometry is assumed to accurate in the input. For reasons of expediency, torsion terms are also excluded. As a first level of approximation to treat rotatable bonds with some double bond character, such bonds are excluded from free rotation during minimization. Speed is enhanced by maintaining a record of the interactions of each segment with all internal segments to avoid recalculation when no relative movement has occurred.

Optimization

The thoroughness of sampling is enhanced with on-the-fly score optimization (Gschwend & Kuntz, 1996). The optimization technique used in DOCK is simplex minimization (Nelder & Mead, 1965; Press & Vetterling, 1989). During rigid docking, rigid body minimization is applied to each generated orientation. Six simplex variables are used for rotation and translation. The quaternion is used to represent the rotation angles because of enhanced numerical stability (Allen & Tildesley, 1987). Additional simplex variables are used to represent rotatable bonds. When torsions are included from multiple layers, movement of the inner torsions causes greater perturbation than movement of outer torsions. To compensate for this distortion, the step size of an inner torsion is scaled by the number of layers it is away from the last segment added.

An important part of minimization is deciding when to stop. The progress of minimization is monitored by checking the difference in score of the best scoring and worst

scoring simplex vertex. When the variation is within some tolerance, minimization terminates. Simplex minimization can be vulnerable to premature convergence. As a consequence, additional cycles of minimization are launched if the current minimization has moved a specified distance in simplex space, which is measured as the vector distance which is normalized with respect to the step sizes of each simplex variable.

An important consequence of using a simplex minimizer is that the results of each run are dependent on how the random number generator was seeded. In this work, multiple runs are performed in which different random number seeds are used. Results are averaged over the set of runs. Standard deviations are computed to assess stability.

Evaluation Techniques

Rebuild X-ray Complexes

The flexible docking algorithm will be tested in several ways. A set of high and medium resolution X-ray crystal complexes will be used to verify the accuracy and gauge the speed of the algorithm. The complexes included in the study are presented in Table 8.

Table 8: X-ray Crystal Complexes used to Evaluate Docking Algorithm

Name	Receptor	Ligand (Anchor segment highlighted by box)	Resolution	Refinement
121p[1]	H-Ras P21	Guanosine-5'-[B,G-Methylene] Triphosphate	1.54	0.195

UCSF LIBRARY

Table 8: X-ray Crystal Complexes used to Evaluate Docking Algorithm (Continued)

Name	Receptor	Ligand (Anchor segment highlighted by box)	Resol ution	Refin ement
1ppc[2]	Trypsin	NAPAP	1.8	0.181
1rnt[3]	Ribonu- clease T1 Isozyme	Guanylic Acid	1.9	0.191
1stp[4]	Streptavidin	Biotin	2.6	0.22
1tmn[5]	Thermolysin	N-(1-Carboxy-3-Phenylpropyl)-L-Leucyl-L-Tryptophan	1.9	0.171
2ctc[6]	Carboxypep- tidase A	L-Phenyl Lactate	1.4	0.161

Table 8: X-ray Crystal Complexes used to Evaluate Docking Algorithm (Continued)

Name	Receptor	Ligand (Anchor segment highlighted by box)	Resolution	Refinement
3cpa[7]	Carboxypeptidase A	Glycyl-L-Tyrosine	2.0	not refined
3dfr[8]	Dihydrofolate Reductase	Methotrexate	1.7	0.152

1. Krengel, 1991.
2. Bode, Turk, & Stuerzebecher, 1990.
3. Arni, Heinemann, Maslowska, Tokuoka, & Saenger, 1987.
4. Weber, Ohlendorf, Wendoloski, & Salemme, 1989.
5. Monzingo & Matthews, 1984.
6. Teplyakov, Wilson, Orioli, & Mangani, 1993.
7. Christianson & Lipscomb, 1986.
8. Bolin, Filman, Matthews, Hamlin, & Kraut, 1982.

These testcases have been used to verify other flexible docking methods (Rarey et al., 1996; Welch et al., 1996). The portions of the ligands selected as the anchor segment may not seem intuitive, but are consistent with the protocol presented in the validation of FlexX. The ligands in the set span a range of sizes and flexibility, possessing from 2 to 12 rotatable bonds.

The testcases were prepared for docking in a uniform fashion. The ligand and macromolecules were prepared in SYBYL by removing all crystallographic waters, adding hydrogens, and loading partial charges (SYBYL Molecular Modeling System, Version 6.2, Tripos Associates, Inc., St. Louis, MO). As described in Meng et al. (1992), AMBER united atom charges were used for the macromolecule and Gasteiger-Marsili charges were

used for the ligand. A molecular surface of the binding site was prepared using MS from the MIDAS package (Ferrin, Huang, Jarvis, & Langridge, 1988), by including only surface residues within 8 Angstroms of the ligand position. The DOCK utility, SPHGEN, was used to compute site points. The first cluster which overlapped the ligand was used. SPHGEN can be used as a *de novo* tool to identify binding sites based on shape alone, but the quality of such a prediction was not assessed here. A scoring grid was then constructed to enclose all the site points plus an extra 8 Angstrom margin along each axis. These calculations took less than an hour for each testcase.

The accuracy of docking was measured to check the scoring function and show that the conformation search is of sufficient quality. A high level of sampling was used -- 900 anchor orientations and 90 pruned configurations. Each run was repeated 10 times with a different random number seed to avoid any artifacts from the random number generator. The benchmark score and position was determined by using DOCK to perform a full minimization of the X-ray position. The goal was to reconstruct the true binding mode or find reasonable alternatives.

The speed of docking was measured to verify that typical molecules could be docked in a feasible amount of computer time. A range of sampling was used -- 50-900 anchor orientations and 5-90 pruned configurations. Each run was repeated 10 times with a different random number seed to compute an average and standard deviation for each level of sampling. The minimum level of sampling is identified by the sampling condition in which at least one of the ten runs generates a configuration within two Angstroms from the X-ray position. The time taken for minimum sampling can be compared with the reported timings of other algorithms. The converged level of sampling is identified by the

UCSF LIBRARY

sampling condition in which all ten runs meet the 2 Angstrom target. The point at which sampling converges is an important measure of stability.

DATABASE SCREENING

An important and historic application of DOCK is the screening of a molecule database for members with likely binding activity. For a flexible docking algorithm to be appropriate to screen a large database, the time spent processing each molecule must be small and on the order of tens of seconds. Of course the amount of sampling is under direct user control, so the algorithm can be applied using any arbitrary amount of sampling. The real issue is how much sampling is required to get "good" results. The tools to address this issue were presented in Chapter 1 in the context of rigid molecule docking. The most relevant measure to assess "screening accuracy" was the convergence of the weighted rank correlation.

In this work, we will use dihydrofolate reductase (3dfr) as the test site to which a set of 50 randomly selected molecule from the CMC molecular database (MDL Information Systems, San Leandro, CA) will be screened. The test database will be seeded with methotrexate to include at least one tight binding molecule.

In order to put the performance of our flexible docking algorithm in context, it would be important to compare the results with reported values for the other docking programs. Unfortunately, none have attempted to determine screening accuracy. Consequently, we will use two reasonable alternative algorithms for comparison. An alternative flexible docking algorithm is based on the "flexibase" approach, or multiple conformer rigid docking (Miller, Kearsley, Underwood, & Sheridan, 1994). In this method the database is seeded with multiple instances of each molecule, which represent different confor-

UCSF LIBRARY

mations. Each conformation is docked independently as a rigid molecule. DOCK can accommodate this technique as a default if Anchor-First docking is not requested. Multiple random conformations of each molecule are constructed and docked sequentially. Sampling is controlled by specifying the number of conformations and the number of orientations sampled for each conformation.

The other alternative algorithm is to simply use rigid docking of the single molecule conformation present in the database. The conformations are generated by CONCORD (Rusinko, Sheridan, Nilakantan, Haraki, Bauman, & Venkataraghavan, 1989).

To calculate the screening accuracy of each method at a given level of sampling, the molecule rankings must be compared with an ideal or best possible ranking. Since we want to isolate the performance of the sampling algorithm from any errors in scoring, we cannot base the target rankings on actual binding data. Instead, the target ranking will be based on the best possible score for each molecule at a very high level of sampling. In practise, the best score or each molecule is determined from all the runs combined. In order to compare the scores among different molecules, only the intermolecular term will be used. This is based on the assumption that the intramolecular terms for the docked configuration after minimization are equivalent to the same terms in the uncomplexed ensemble of configurations. In addition, a 0.5 kcal/mol penalty is applied per heavy atom to approximate the desolvation terms that are neglected in the current scoring function. As described in Chapter 1, the absence of desolvation terms causes the current scoring function to consistently favor larger molecules. Given the rankings, the screening accuracy is computed based on the weighted rank correlation described in Chapter 1. This quantity varies from negative one for perfect anti-correlation to positive one for perfect correlation.

UCSF LIBRARY

By weighting the correlation with the rank, then the relative rankings of the top scoring molecules is emphasized. This bias is consistent with the notion that database screening is used primarily to identify the top scoring molecules, and any inaccuracies of molecules lower on the list are of diminishing importance.

RESULTS AND DISCUSSION

Pruning Methods

Three different methods were evaluated to prune the partially-built molecule configurations during each cycle of segment addition. The results of the study are presented in Figure 11. The pruning based on rank alone shows a remarkable amount of focusing about a few binding modes. The closest docked position to the X-ray position is 2.5 Angstroms away. This binding mode corresponds to withdrawing methotrexate outwards along its binding channel by about an Angstrom, and swapping the interactions of the carboxylate groups of the glutamate portion of the inhibitor. This behavior is strongly reminiscent of the local minimum problem, suggesting that rank alone focuses the search on good scoring partial configurations that are ultimately non-optimal. The RMSD only pruning on the other hand appears to be unable to focus on any particular binding mode, placing only a handful of configurations within five Angstroms of the X-ray position. The nearly uniform distribution of configurations at around ten Angstroms from the X-ray position suggests that diversity alone is counterproductive in an optimization algorithm. The last method using both rank and RMSD to prune the configurations manages to reconstruct configurations within an Angstrom of the X-ray position. Interestingly it also constructs a large number of configurations at 2.5 Angstroms from the X-ray position,

UCSF LIBRARY

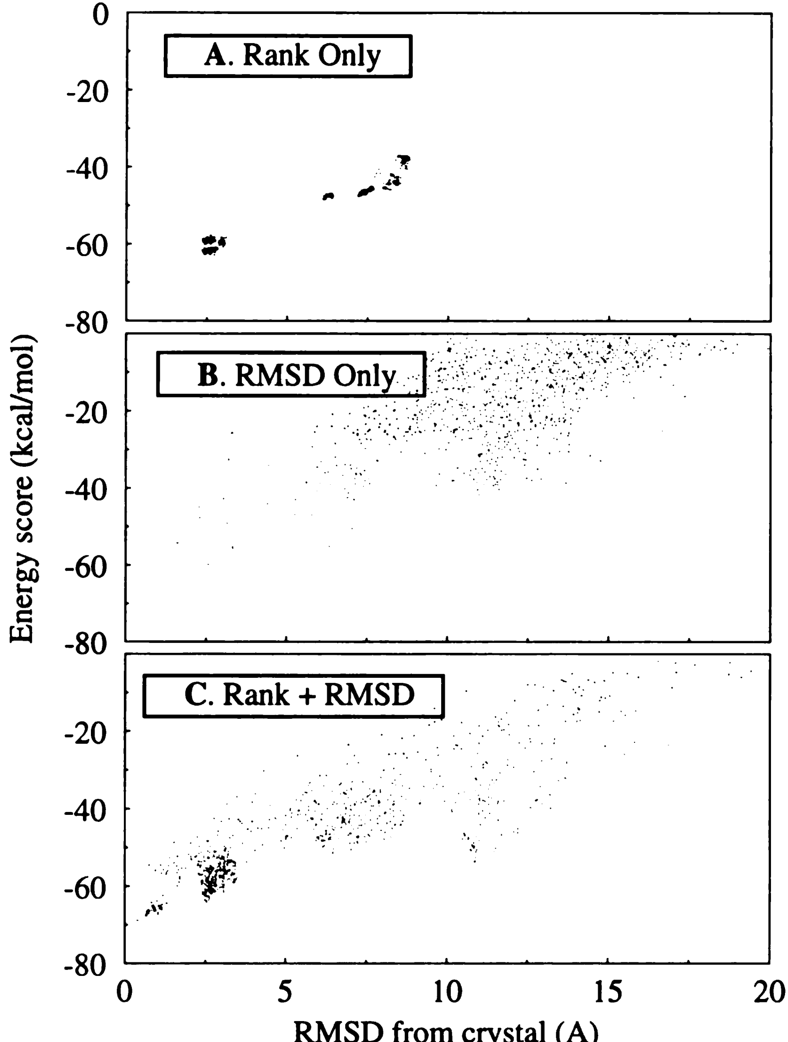


Figure 11. Pruning Methods for Incremental Construction. Methotrexate was redocked to 3dfr using 500 orientations of the anchor and 100 configurations per step during construction while holding the anchor positions fixed. Data is collected over 10 runs using a different random number seed for each run. The results of each plot differ based on the method used to prune the configurations in each stage of construction down to 100 configurations. In **A**, the 100 best ranking configurations were retained. In **B**, the 100 most diverse configurations were retained. The diverse set is selected based on complete-linkage hierarchical clustering of the RMSD distance between each configuration. In **C**, the 100 best ranking and most diverse configurations were retained. Complete linkage hierarchical clustering was performed using the RMSD distance divided by the lower rank (larger value) of the two configurations.

consistent with the rank only pruning, confirming the importance that this binding mode plays early in configuration construction. The ability to overcome the local minimum

inherent in that binding mode is encouraging. Based on these results, rank-weighted RMSD clustering was selected as the method of pruning partial configurations during segment addition.

Rebuild X-ray Complexes

DOCKING ACCURACY

The results of testing the accuracy of the docking algorithm is presented in Table 9.

Table 9: Accuracy of Dockings
Summary of predicted binding modes from 10 high sampling runs

Complex	X-ray position after minimization		Nearest docked position		Top scoring docked position	
	RMSD	Score	RMSD	Score	RMSD	Score
121p	0.49	-136	0.84	-129	0.96	-131
1ppc	0.29	-56	0.76	-53	10.07	-56
1rnt	0.67	-52	0.60	-51	1.32	-52
1stp	0.73	-37	0.63	-34	1.22	-38
1tmn	0.56	-72	0.49	-64	2.69	-68
2ctc	0.76	-39	0.65	-39	1.15	-41
3cpa	1.48	-43	1.25	-38	3.90	-50
3dfr	0.52	-65	0.58	-68	1.03	-69

UCSF LIBRARY

The first two columns of data contain the results of the minimization of the X-ray complexes. A total of 10 minimization runs were performed with different random number seeds. The best scoring results are presented for each testcase. The scores range from -37 kcal/mol (1stp) to -136 kcal/mol (121p). The large, negative score of 121p results from the presence of four negative formal charges on the triphosphate ligand. The interaction energy of biotin with streptavidin (1stp) is less favorable by comparison to the other com-

plexes. This result is inconsistent with the renowned affinity of this interaction. The inconsistency of the energy score with the binding constant is expected since it only estimates the enthalpy of the interaction rather than actual free energy terms. The purpose of this work is not to predict binding constants, but to show that with a satisfactory scoring function, the docking search algorithm succeeds in finding the most likely or at least reasonable binding modes.

The RMSD data reveal that in most systems the ligand moves more than 0.5 Angstroms. This deviation may be a result from fundamental errors in missing free energy terms, or mere differences with the energy function used during crystallographic refinement. The most deviations occur in the 3cpa testcase. Two factors may be at work. The interaction of the peptide ligand with a doubly charged metal ion in the binding site may be incorrectly modeled by a simple Coulombic electrostatic term. In addition, the quality of the crystal structure may be questionable, because the ligand and nearby protein side chains were modeled based on the electron density, but not refined.

The next two columns of data correspond to the nearest docked position found in any of the ten high sampling runs (900 anchor orientations and 90 pruned configurations). The RMSD and score values are the same order of magnitude as the corresponding minimization values. In five out of eight testcases, the docked position is closer than the minimized position. This phenomenon probably arises from premature convergence of the minimizer during docking, since all positions near the X-ray position should collapse to the same position after minimization. Fortunately, the differences in RMSD are within about a tenth of an Angstrom, which is certainly on the order of the uncertainty of the 0.3 Angstrom energy potential grid. In a single testcase (3dfr), the score of the nearest docked

position is better than the minimized score. This indicates the presence of unavoidable local minima that prevent the X-ray position from minimizing to the nearby global minimum.

The last two columns of data correspond to the top scoring docked position found over all of the ten runs. For a docking algorithm employing exhaustive sampling and a perfect scoring function, these positions would coincide with the X-ray positions. Indeed, in five out of eight cases, these positions are within two Angstroms RMSD of the X-ray position. Interestingly, in four out of eight cases, the scores of these positions are better than the minimized X-ray scores, indicating some inconsistency in the scoring function.

The distribution of binding modes of one of the successfully docked molecules (1stp) is presented in Figure 12. The top scoring binding mode is represented by a cluster of positions from 0.5 to 1.5 Angstroms from the X-ray position. This mode is clearly the most populated binding mode constructed by the algorithm. For this testcase, the docking score was better than the minimized X-ray position.

The top scoring position of biotin within the binding pocket of streptavidin is graphically depicted in Figure 13. The position of the hydantoin portion of biotin overlaps well. The alkyl chain shows some deviation in atom position, but it lies within the correct binding groove.

The 1tmn testcase could not be reconstructed accurately by the flexible docking algorithm. The best scoring binding mode is just under three Angstroms from the X-ray position. Since its score is worse than that of the minimized X-ray position, the source of the problem cannot lie in the scoring function. The inability to reconstruct this testcase instead points to a deficiency in the sampling. In future work, this testcase may prove use-

UCSF LIBRARY

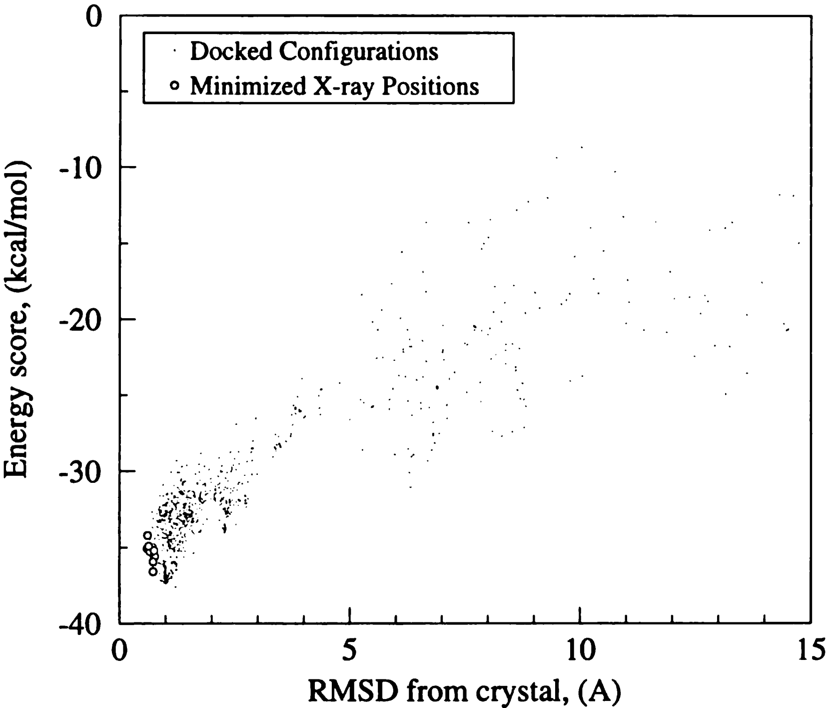


Figure 12. Scatter Plot of Docked Positions of Biotin with Streptavidin (1stp). Biotin was docked to streptavidin using a high level of sampling (900 anchor orientations, 90 configurations/construction step). The top 90 configurations for each of 10 runs is displayed as points. The X-ray position was also minimized in 10 independent runs. The minimized positions are displayed as circles.

ful in helping guide algorithm development.

The 3cpa testcase is another situation in which the best scoring position deviated significantly from the X-ray position. However, the score of this position was better than that of the minimized X-ray position, so the deviation may be isolated to problems with the scoring function. The distribution of binding modes is presented in Figure 14. Four dominant binding modes are visible up to nine Angstroms away from the X-ray position. Intriguingly, all of them are lower in energy than the minimized X-ray energy. The minimized X-ray positions appear to move considerably from the original position. It is clear that the scoring function does not prefer the X-ray position. This situation could indicate errors with the scoring function, or problems with the unrefined crystal structure. The

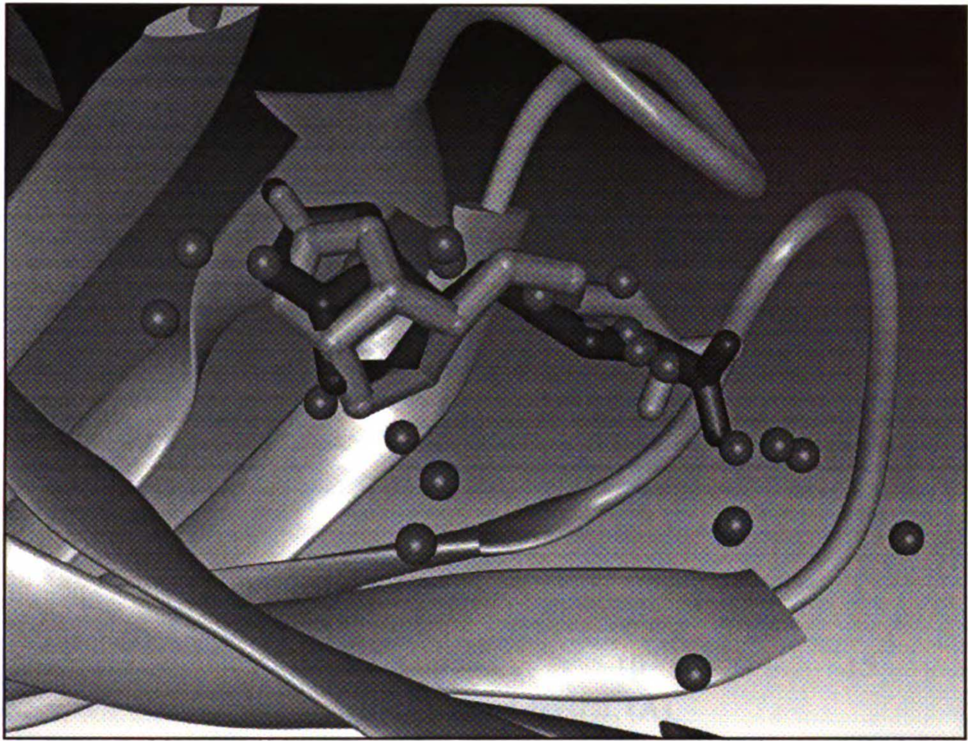


Figure 13. Top-scoring binding mode of biotin with streptavidin. Rendering of top-scoring configuration from data presented in Scatter Plot of Docked Positions of Biotin with Streptavidin (1stp) Streptavidin is depicted as gray ribbon. SPHGEN site points are shown as green spheres. X-ray biotin position is in red. Top scoring biotin position is in yellow. Figure generated with UCSF MidasPlus (Ferrin et al., 1988).

binding modes are depicted graphically in Figure 15. All the docked binding modes occupy approximately the same volume as the X-ray position. They share the common feature of having the carboxy group on the ligand tyrosine interacting with the doubly charged zinc ion. The position nearest to the X-ray position has the carboxy group pulled closer to the zinc ion, which pulls the tyrosine out of its binding pocket. The second mode pulls the tyrosine farther. The third and fourth mode have tyrosine flipped end for end with respect to the crystal position.

The 1ppc testcase has the most distant top scoring position at 10 Angstroms from the X-ray position. This binding mode has the same score as the minimized X-ray posi-

UCSF LIBRARY

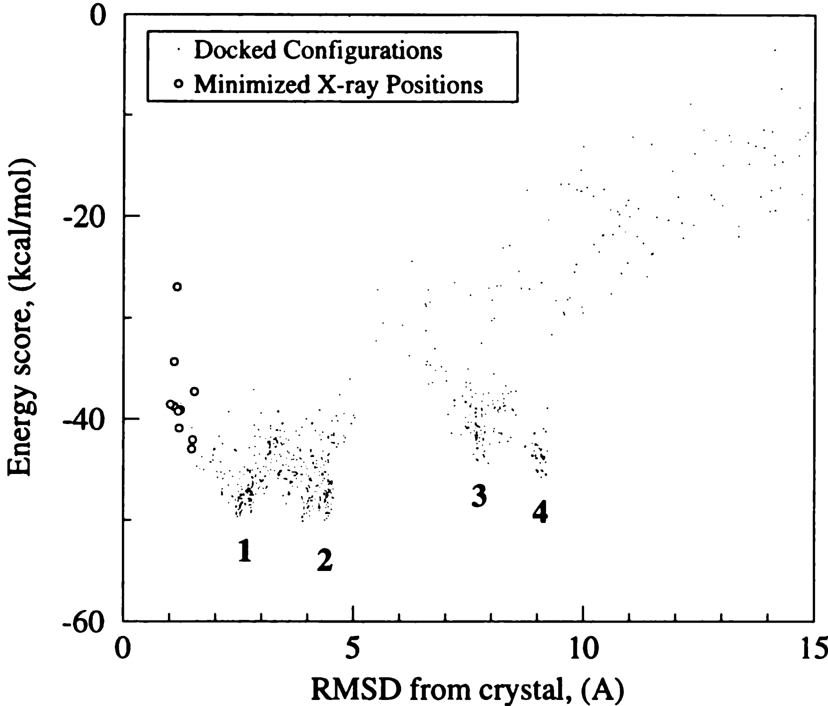


Figure 14. Scatter Plot of Docked Positions of Glycyl-L-Tyrosine with Carboxypeptidase A (3cpa). Glycyl-L-Tyrosine was redocked with a high level of sampling (same as Figure 12). Docked positions are shown as dots. Minimized X-ray positions are shown as circles. The four dominant binding modes are identified numerically.

tion, so the scoring function is unable to distinguish the two modes.

Docking Speed

The results of the evaluation of docking speed are presented in Table 10. All timings are based on using a 200 Mhz R4400 Indy2 processor from Silicon Graphics. Although some of the calculation were performed on a 150 Mhz R10000 Octane processor which is threefold faster for these calculations. In this table, successful docking is based on a particular run generating a configuration within two Angstroms of the X-ray position, although the configuration does not need to be a top scoring configuration.

Minimum sampling has been defined as the level of sampling for which at least one out of ten runs is successful. It provides a lower bound on the search time. This quan-

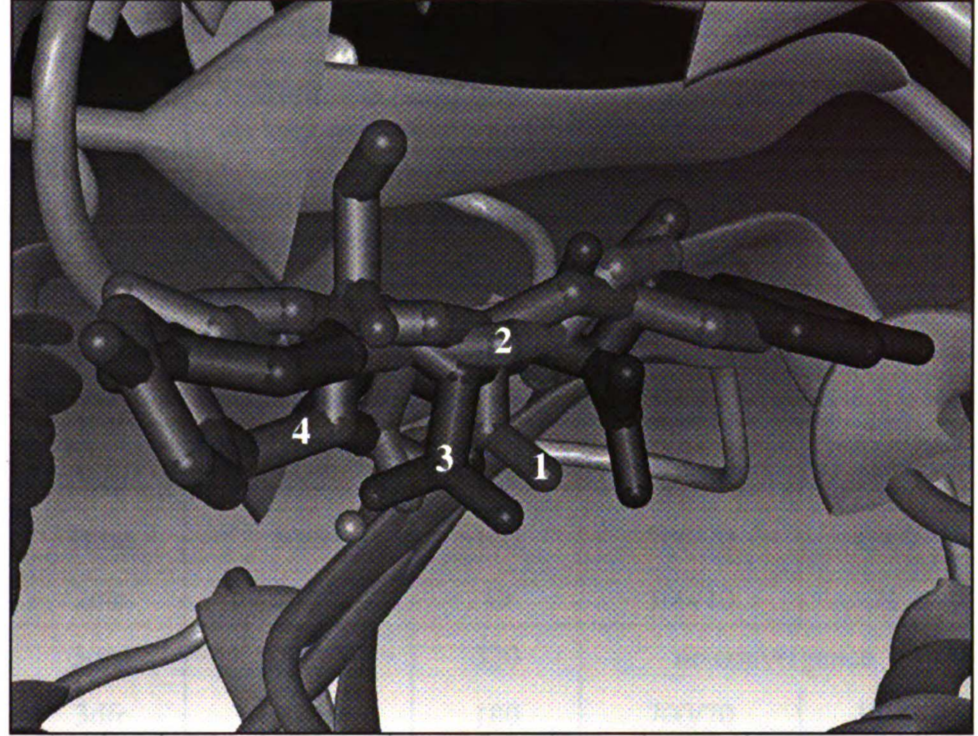


Figure 15. Top-scoring binding modes of glycyl-l-tyrosine with carboxypeptidase A. Rendering of the top-scoring configurations identified in Figure 14. Carboxypeptidase A is depicted as gray ribbon. The catalytic Zn++ ion is shown as a yellow sphere. The X-ray glycyl-l-tyrosine position is in red. The top scoring glycyl-l-tyrosine positions are in separate colors (1=green, 2=cyan, 3=purple, 4=orange). Figure generated with UCSF MidasPlus (Ferrin et al., 1988).

tity is presented for each testcase in the first two columns of Table 10. Minimum sampling varies from 10 seconds (2ctc testcase) to 320 seconds (3cpa testcase). If the molecules for which the top scoring binding mode is outside the two Angstrom cutoff (see Docking Accuracy section in Results and Discussion above), then the upper limit on minimum sampling is 180 seconds (3dfr testcase). These timings are very competitive with previously reported algorithms which for similar testcases required up to 180 seconds (Rarey et al., 1996) and 400 seconds (Welch et al., 1996).

Converged sampling has been defined as the level of sampling at which all runs are successful. It represents the point at which the algorithm is completely stable. This quan-

Table 10: Speed of Dockings
Sampling required to get within 2 Angstroms of X-ray Position

complex	**Minimum Sampling** At least one run succeeds		**Converged Sampling** All 10 runs succeed	
	orientations/ configurations	time (sec)	orientations/ configurations	time (sec)
121p	50/5	88	500/50	890
1ppc	50/5	150	no convergence	
1rnt	50/5	42	300/30	250
1stp	50/5	20	50/5	22
1tmn	50/5	160	700/70	2800
2ctc	50/5	10	100/10	21
3cpa	500/50	320	no convergence	
3dfr	100/10	180	700/70	1200

Docking was performed over a range of sampling levels by adjusting the number of anchor orientations (50 - 900) and partial construction configurations (5-90). At each level of sampling, 10 runs were performed which differed by the value used to seed the random number generator.

tity is presented for each testcase in the last two columns of Table 10. The lowest value is 21 seconds (2ctc). The upper value cannot be determined for the entire set because two of the systems did not converge. Since these testcases prefer a distant binding mode, they can never attain successful convergence with increasing sampling. If these testcases are excluded, then the upper limit is 1200 seconds (3dfr).

Database Screening

The time convergence of screening accuracy for each of the docking methods is presented in Figure 16. The anchor first method clearly outperforms the benchmark algorithms. It converges to an accuracy of 0.9 by about 100 seconds/molecule. The uncertainty in the accuracy is relatively narrow -- about 0.25 units. The multiple conformer

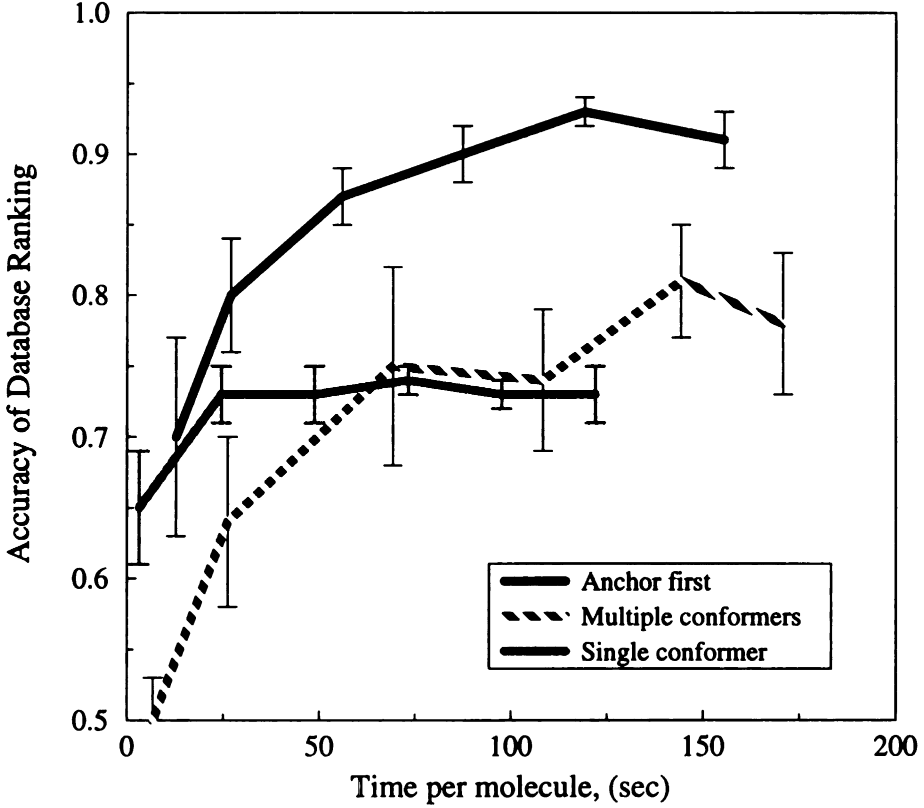


Figure 16. Accuracy versus time of database processing. A testset of 51 molecules was docked to dihydrofolate reductase (3dfr) using three docking techniques over a range of sampling conditions. Each data point represents the average accuracy (+ deviation) of the rankings of 10 docking runs at a particular level of sampling. The ranking accuracy is computed as the weighted correlation of the single-run rankings compared to the overall rankings. The timings are based on a 200Mhz R4400 SGI Indigo2 processor.

UCSF LIBRARY

method is sluggish by comparison. It converges to an accuracy of 0.8 after about 150 seconds/molecule. The uncertainty in the accuracy is considerably higher -- about 0.5 units. The single conformer method does surprisingly well considering the lack of conformational sampling. It converges to an accuracy of about 0.73 in about 20 seconds/molecule. The uncertainty is relatively narrow -- 0.25 units.

The implications of this experiment are that if very little time is allowed for calculation (< 20 seconds/molecule), both single conformer and anchor first docking are com-

petitive. If more time is available (> 50 seconds/molecule), then the anchor first method is superior. The current generation of processors (SGI R10000) reduce these times by a factor of three, which makes flexible docking feasible for database screening.

CONCLUSION

The flexible docking algorithm presented here is moderately accurate in its ability to reconstruct a majority of X-ray crystal complexes to within two Angstroms RMS error. Some failures can be attributed to inaccuracies in the scoring function, which are being addressed in independent research projects. The algorithm is fairly rapid for flexible ligands. Most testcases can be reconstructed within 10 to 100 seconds.

When screening a database of flexible molecules, the method is more accurate than related docking methods. It can achieve a 90 percent accuracy in predicted ranking within 100 seconds on late model processors (SGI R4400), or about 30 seconds on current processors (SGI R10000).

This flexible docking algorithm represents a considerable advance in technology. It will enable new docking applications to be addressed. By utilizing intuitive sampling parameters, the user will have full and predictable control over the docking process. By emphasizing rapid algorithms at a small cost to accuracy, the algorithm will be useful for screening large databases of molecules.

REFERENCES

Allen, M. P. & Tildesley, D. J. (1987). Computer simulation of liquids. New

York: Clarendon Press.

Arni, R., Heinemann, U., Maslowska, M., Tokuoka, R., & Saenger, W. (1987) Restrained least-squares refinement of the crystal structure of the Ribonuclease T1-Guanylic acid complex at 1.9 angstroms resolution. Acta Crystallographer, B43, 549.

Bode, W., Turk, D., & Stuerzebecher, J. (1990). Geometry of binding of benzamidine- and arginine-based inhibitors ... to human alpha-thrombin: X-ray crystallographic determination of the NAPAP-Trypsin complex and modeling of NAPAP-Thrombin and MQPA-Thrombin. European Journal of Biochemistry, 193, 175.

Bolin, J. T., Filman, D. J., Matthews, D. A., Hamlin, R. C., & Kraut, J. (1982). Crystal structures of Escherichia Coli and Lactobacillus Casei Dihydrofolate Reductase refined at 1.7 angstroms resolution. I. General freatures and binding of methotrexate. Journal of Biological Chemistry, 257, 13650.

Christianson, D. W. & Lipscomb, W. N. (1986). X-ray crystallographic investigation of substrate binding to Carboxypeptidase A at subzero temperature. Proceedings of the National Academy of Sciences, USA, 83, 7568.

Clark, K. P. & Ajay. (1995). Flexible ligand docking without parameter adjustment across four ligand-receptor complexes. Journal of Computational Chemistry, 16 (10), 1210-1226.

UCSF LIBRARY

Ferrin, T. E., Huang, C. C., Jarvis, L. E., & Langridge, R. (1988). The MIDAS display system. Journal of Molecular Graphics, 6, 13-27.

Gschwend, D. A. & Kuntz, I. D. (1996). Orientational sampling and rigid-body minimization in molecular docking revisited - on-the-fly optimization and degeneracy removal. Journal of Computer-Aided Molecular Design, 10(2), 123-132.

Judson, R. S., Jaeger, E. P., & Treasurywala, A. M. (1994). A genetic algorithm based method for docking flexible molecules. Theochem-Journal of Molecular Structure, 114, 191-206.

Knegtel, R. M. A., Kuntz, I. D., & Oshiro, C. M. (1997). Molecular docking to ensembles of protein structures. Journal of Molecular Biology, 266 (2), 424-440.

Krengel, U. (1991). PhD. Thesis. Heidelberg University.

Leach, A. R. (1994). Ligand docking to proteins with discrete side-chain flexibility. Journal of Molecular Biology, 235 (1), 345-356.

Leach, A.R., and Kuntz, I.D., Conformational Analysis of Flexible Ligands in Macromolecular Receptor Sites. Journal of Computational Chemistry. (1992) 13 no. 6, 730-748.

UCSF LIBRARY

Makino, S., & Kuntz, I.D. (1997). Journal of Computational Chemistry, In press.

Meng, E. C., Shoichet, B. K., & Kuntz, I. D. (1992). Automated docking with grid-based energy evaluation. Journal of Computational Chemistry, 13(4), 505-524.

Miller, M. D., Kearsley, S. K., Underwood, D. J., & Sheridan, R. P. (1994). Flog - a system to select quasi-flexible ligands complementary to a receptor of known three-dimensional structure. Journal of Computer-Aided Molecular Design, 8(2), 153-174.

Mizutani, M. Y., Tomioka, N., & Itai, A. (1994). Rational Automatic Search Method for Stable Docking Models of Protein and Ligand. Journal of Molecular Biology, 243, 310-326.

Monzingo, A. F. & Matthews, B. W. (1984). Binding of N-Carboxymethyl dipeptide inhibitors to thermolysin determined by X-ray crystallography. A novel class of transition-state analogues for zinc peptidases. Biochemistry, 23, 5724.

Morris, G. M., Goodsell, D. S., Huey, R., & Olson A. J. (1996). Distributed automated docking of flexible ligands to proteins - Parallel applications of AutoDock 2.4. Journal of Computer-Aided Molecular Design, 10 (4), 293-304.

Nelder, J. A., & Mead, R. (1965). Computer Journal, 77, 308.

UCSF LIBRARY

Oshiro, C. M., Kuntz, I. D., & Dixon, J. S. (1995). Flexible ligand docking using a genetic algorithm. Journal of Computer-Aided Molecular Design, 9(2), 113-130.

Pearlman, D. A., Case, D.A., Caldwell, J. W., Ross, W. S., Cheatham, T. E., DeBolt, S., Ferguson, D., Seibel, G., & Kollman, P. (1995). AMBER, A package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. Computer Physics Communications, 91 (1-3), 1-41.

Press, W. H. & Vetterling, W. T. (1989). Numerical recipes: the art of scientific computing. New York: University Press.

Rarey, M., Kramer, B., Lengauer, T., & Klebe, G. (1996). A Fast Flexible Docking Method using an Incremental Construction Algorithm. Journal of Molecular Biology, 261, 470-489.

Rusinko, A., Sheridan, R. P., Nilakantan, R., Haraki, K. S., Bauman, N., & Venkataraghavan, R. (1989). Using concord to construct a large database of 3-dimensional coordinates from connection tables. Journal of Chemical Information and Computer Sciences, 29(4), 251-255.

Teplyakov, A., Wilson, K. S., Orioli, P., & Mangani, S. (1993). High-resolution

structure of the complex between Carboxypeptidase A and L-Phenyl Lactate. Acta Crystallographica, D49, 534-540.

Weber, P. C., Ohlendorf, D. H., Wendoloski, J. J., & Salemme, F.R. (1989). Structural origins of high-affinity Biotin binding to Streptavidin. Science, 243, 85.

Welch, W., Ruppert, J., & Jain, A. N. (1996). Hammerhead: Fast, Fully Automated Docking of Flexible Ligand to Protein Binding Sites. Chemistry and Biology, 3, 449-462.

Chapter 4: Conclusion

RECAP

The primary focus of this thesis project has been the development of search algorithms for molecular docking. Since an important application of this technique is the processing of large molecule databases, the consideration of computation performance has taken utmost importance. Consequently, the proposed algorithms have struck delicate balance between accuracy and speed.

In Chapter 1, we developed a new orientation search algorithm for rigid molecule docking which was a significant improvement over previous algorithms. In addition, we developed protocols to measure the performance of docking search algorithms. These protocols effectively isolate the influences of the uncertainty in the scoring function, so that the relative differences in search methods can be measured directly. In Chapter 2, we extended the docking algorithm to treat flexible molecules. The flexible molecule docking algorithm was shown to be as accurate and fast as other reported methods at rebuilding known X-ray complexes. It was also sufficiently fast for database processing.

FUTURE DIRECTIONS

The docking algorithms presented in this work could be extended on three fronts: enhanced sampling, optimization and scoring.

Sampling can be improved by explicit consideration of ring flexibility. The most straightforward way to treat ring flexibility is through the use of templates which represent highly populated conformations for each type of flexible ring system. Alternatively, a more general -- but expensive -- approach is to use a partial systematic conformation

search on some of the ring bonds followed by ring closure calculations. To include ring relaxation in the score optimization step, a torsion term will need to be added to the intramolecular scoring function in order to keep the torsions in preferred positions. These potentials would also be useful in the treatment of the relaxation of bonds with partial double bond character, which are currently excluded from optimization.

Enhanced optimization is another area in which the docking algorithm can be extended. Although attempts were made to incorporate optimization steps directly into the search algorithm, the docked positions produced by this algorithm should be considered initial placements. Further steps of refinement would be warranted if a high quality docking model is needed. Since output from DOCK is now compatible with the SYBYL modeling package, it would be straightforward to use SYBYL to perform molecular dynamics on the docked complex to increase local optimization. Extensive local optimization could also be incorporated into the docking program using the genetic algorithm.

Improved scoring functions would also be a useful area of development. Since the scoring function provides the connection between the computer model and the real world experiment, improvements in it are critical. To better model binding affinity, the scoring function will need to include terms to represent free energy quantities. The most important phenomenon is the modeling of ligand and receptor desolvation. An ideal treatment of desolvation would correct several deficiencies in the current scoring function. First, correcting for the desolvation of charged groups would penalize highly charged ligands interacting with weakly charged sites as well as uncharged ligands interacting with charged receptor side chains. To a lesser extent, placing nonpolar ligands groups next to hydrogen bonding receptor side chains -- and vice versa -- would also be penalized.

Finally, the burial of nonpolar ligand groups within nonpolar receptor pockets would be aptly rewarded. Since desolvation calculations often involve surface area terms and/or many body terms, they can be expensive to calculate. Therefore, multiple levels of treatment may be necessary depending on the balance of quality versus speed needed for a particular calculation.

CLOSING

The focus of this work has been algorithm development and validation. The orientation and conformation search algorithms represent an important step forward in docking technology. Since these search algorithms do not rely on any particular aspects of the scoring function, they would be compatible with any future developed scoring functions. The validation protocols are also sufficiently general, and should help direct the future development of search algorithms.

UCSF LIBRARY

Appendix 1: Users Guide

Copyright © 1997
Regents of the University of California
All Rights Reserved

Scope of This Guide

This section is intended as a supplement to the DOCK appendix 2: reference manual. It describes the steps a user would typically take to apply the programs to a macromolecule and potential ligands of interest. While the reference manual describes in detail the various input and output files, this guide is meant to convey the process in informal terms. Some of the difficulties we have encountered as well as approaches we have found useful are discussed.

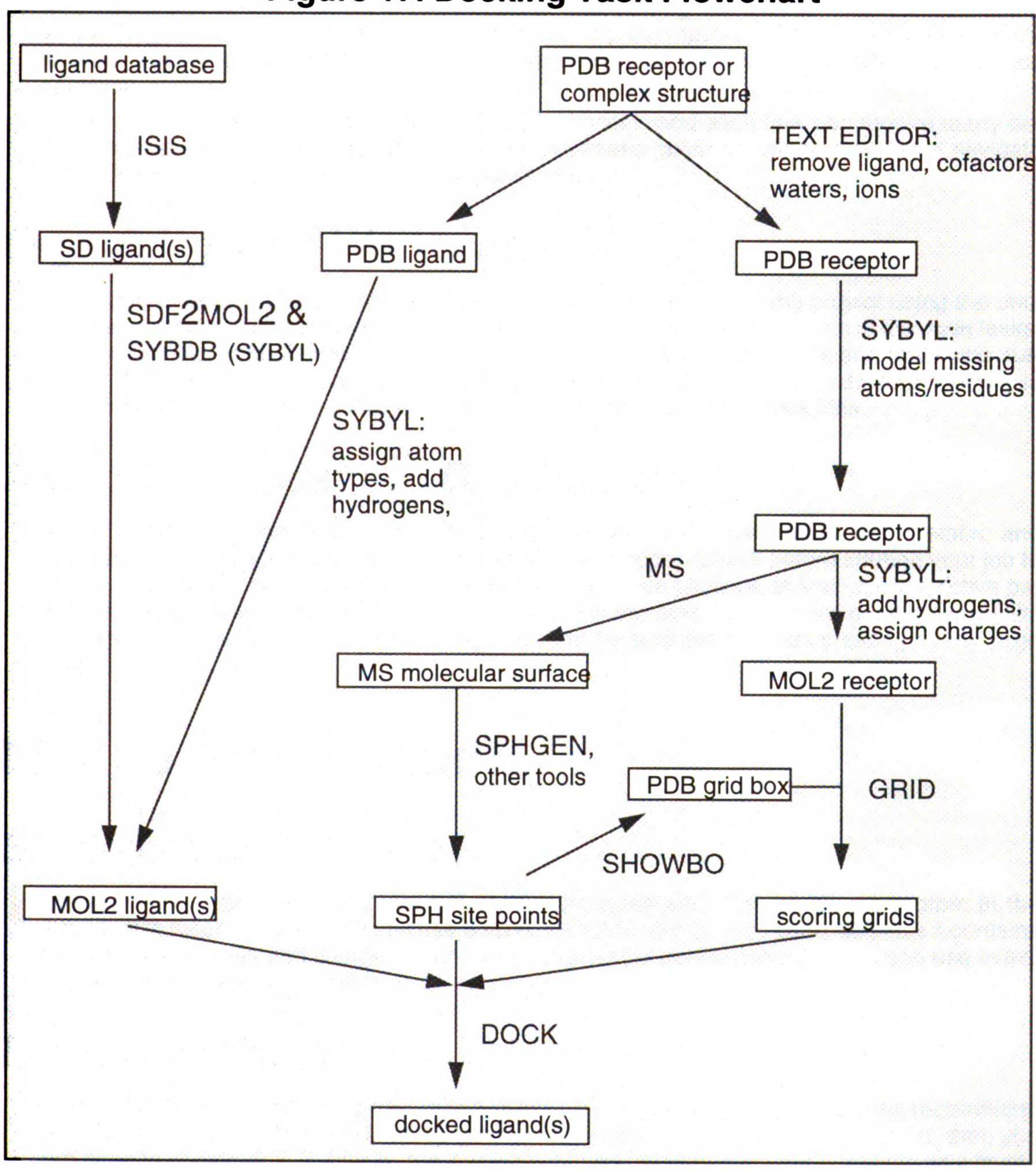
What DOCK Can Do for You

DOCK is a program for locating feasible binding orientations, given the structures of a "ligand" molecule and a "receptor" molecule. What is considered feasible depends on how the orientations are evaluated. Current options are a contact (shape-fitting) score,a force field interaction energy and a new user-defined scoring scheme. DOCK generates many orientations of one ligand. The best-scoring orientation of each molecule is saved, and the best-scoring molecules are written out. Some of the molecules in the list of best-scoring compounds, perhaps with modifications, may be interesting as potential new ligands for the receptor.

Getting Started

written by Cindy Corwin and Todd Ewing

Figure 17. Docking Task Flowchart



Overview

The basic requirement for docking is a structure of the macromolecule of interest. The docking procedure can be divided into four general stages: ligand preparation, site characterization, scoring grid calculation, and docking itself. Please refer to Figure 17 while reading this guide.

Site characterization is the process of deciding what areas of the receptor site to study. This is done by constructing site points to map out the negative image of the active site. These site points are used by DOCK to construct orientations of the ligand.

Scoring grid calculations are necessary so that DOCK can evaluate ligand orientations rapidly.

The final stage of the process is running DOCK and viewing the results. DOCK uses the site points to generate ligand orientations, then uses the precomputed grids to evaluate the orientations. The best-scoring molecules or orientations may be viewed using a molecular graphics program.

There are multiple tasks involved in the docking process, and each task can require many decisions over input parameters. We hope this beginner's guide will make it easier to navigate through the tasks and to select sensible parameters.

Organizing Your Workspace

It is a good idea to make a make a new UNIX directory for each docking project using the UNIX `mkdir` command. Within this project directory make a sub-directory for each of the main tasks. Make a `struc/` sub-directory to hold the ligand and receptor coordinates and molecular surfaces. Make a `site/` sub-directory to hold the site point files. Make a `grid/` sub-directory to hold the scoring grid files. Make a `dock/` sub-directory to hold the dock files.

A Caution Concerning Disk Space

The output from some of the programs associated with DOCK, particularly MS, SPHGEN, and DOCK itself, may require substantial amounts of disk storage. Check before starting your job to make sure there is space available. It is a good idea to be cautious at first: use restrictive parameters choices with only a handful of ligands, to make sure that you are getting the results you desire. While DOCK jobs are running, check to be sure they are not creating overly large files.

Ligand Preparation

Single Ligand

Before you can dock a ligand, you will need atom types and charges for every atom in the ligand. It is recommended that you use SYBYL MOL2 format for the ligand file since it contains fields for atom types and charges. For a single ligand (or several ligands), you can use SYBYL to prepare a MOL2 file for the ligand.

Ligand Database

Check if a database of ligands has already been prepared at your site. Again, we recommend that this database be in SYBYL MOL2 format. If the MOL2 database does not exist, then you will need to construct it. Typically, the Available Chemicals Directory (ACD) is used as a ligand database. This database is distributed by Molecular Design Ltd. (San Leandro, CA) for use with the ISIS database package. The ACD can be output from ISIS in an SD-format file. Use

SDF2MOL2 & SYBDB to generate a MOL2 file from the SD file. This conversion requires SYBYL (from Tripos) to assign atom types and charges.

Site Characterization

Working With Macromolecular Models and Generating the Molecular Surface

Removing Ligands and Crystallographic Waters

The macromolecular structure you are working with may include a ligand, and crystal structures usually contain water molecules and sometimes ions which were found on the surface of the protein. These molecules are usually not included as input to MS. To prepare for molecular surface generation, make a copy of the protein coordinate file. If there is a ligand present, remove it by deleting all of its records (they often start with HETATM in Brookhaven Protein Data Bank format files) from your copy of the file. (Note - sometimes, as in the case of a cofactor or catalytic metal ion, it may make chemical sense to keep a ligand in the PDB file.) Whether or not crystallographic waters and ions should be preserved when generating surfaces for use by SPHGEN is a matter of some debate. In structures of complexes, water molecules and ions are often found in the protein binding pocket along with the ligand(s). However, ligands can displace waters and ions, and the volume of a receptor site will be explored more completely if the waters and ions are removed, so if you don't have particular reasons for preserving any of the water molecules or ions in the crystal, it is probably best to remove all of them. Waters are usually located near the end of the PDB file and are often HETATM records with HOH or WAT residue types. Ions are often near the waters in the PDB file.

Please note that the PDB file used for generating the molecular surface should not include hydrogen atoms. NMR structures will include hydrogens; delete the hydrogens from a copy of each structure and use that copy in MS.

Creating the Molecular Surface

The dot surface which will be used to produce spheres is generated by the program MS, available from Quantum Chemistry Program Exchange (QCPE). When setting up for docking, it is acceptable just to generate surface for the site of interest and adjacent regions (see documentation for **GET_NEAR_RES** and AUTOMS); this will also reduce the computer time used by SPHGEN. *Note: SPHGEN requires that the surface points must have associated normals.*

If you use the QCPE version of MS, you must run REFORMATMS to convert the surface to the format used by SPHGEN (both formats are described in the reference manual section on REFORMATMS). REFORMATMS is interactive and requires the surface and the PDB file used to generate the surface.

Users of the UCSF MidasPlus package may use the output from the DMS program directly as input for SPHGEN.

Representing the Site With Spheres

We typically use SPHGEN to construct shape-based site points, but you may use any other program to construct site points. With the use of other programs you may include considerations of chemical complementarity in your site points. A common alternative to SPHGEN is the Goodford's GRID program (Peter Goodford).

UCSF LIBRARY

SPHGEN

SPHGEN uses the points of the molecular surface and their associated normals to determine spheres to fill the site. It then reduces the number of spheres to one per atom and groups them into clusters. You can inspect these clusters and regroup the spheres if necessary.

Creating INSPH

The parameters which tell SPHGEN exactly how to create the surface are placed in a file called INSPH, which must be present when SPHGEN is run. The contents of this file are described in the reference manual. To create it, make a file with each variable on a separate line. Most of the parameter values given in the reference manual should work fine. You will need to replace *msfil* with the name of your surface file and *outfil* with the desired name of your output file.

Running SPHGEN

SPHGEN must use the directory containing INSPH as its working directory; this means that it should be started while you are in that directory. The SPHGEN output file contains clusters of spheres which have been selected and grouped by SPHGEN; the clusters are listed in order of decreasing size. The last cluster, numbered 0, contains all the spheres produced. It may be used with the program CLUSTER to make new sphere clusters if the original clustered output doesn't describe the site well.

Looking at the Output

Once you've generated spheres, you should look at the sphere clusters using a molecule display program. SHOWSPHERE may be used to generate a PDB-like file of sphere centers for display. It can also generate a surface for the sphere cluster (in the MS format used by SPHGEN). SHOWSPHERE is interactive. You will be prompted for the name of the cluster file (that is, the SPHGEN output), the number of the cluster, and names for the desired output file. In the PDB-like file of sphere center coordinates, each sphere is a separate residue and the spheres are separated by TER cards.

Getting a Good Sphere Cluster

Displaying the protein and sphere centers together should tell you how each sphere cluster is related to the site you are trying to represent. Examine sphere clusters until you find one that occupies the region into which you want to dock ligands. Clusters of 50 or fewer spheres are best; larger numbers of spheres will cause DOCK to use more computer time. It is generally unwise to try docking with more than 100 spheres, although you may be able to use more if your database is small or you are using chemical matching. Initial sphere clusters are sometimes spider-like structures which include the area of interest but also branch into other regions. If your cluster has too many spheres, branches out, or is unsatisfactory for some other reason, you can correct the problem.

The easiest way to fix a sphere cluster is to use graphics to identify spheres that you don't really need, then remove them. When you've found the unnecessary ones, go back to the *original* sphere cluster file (i.e. the one from SPHGEN) and delete the corresponding lines - the residue number in the PDB-like file of centers is the first number in the line in the sphere file. Remember to change the number of spheres listed on the line with the cluster number to reflect the deletions.

If your cluster is large — more than about 100 spheres — and deleting spheres by hand looks too tedious, you can use CLUSTER to break it into smaller clusters. CLUSTER is described in the reference manual; read the documentation completely before you try it. Start with the parameters given and experiment with the values; small changes can make a big difference in the result. Be aware that if the best cluster found is the same as the original input cluster, the program will appear not to have done anything.

The two methods just described may be combined if the best CLUSTER output is not quite right. More spheres can be deleted from the new cluster, or, if the new cluster is too small, additional

UCSF LIBRARY

spheres may be added graphically. A cluster containing all the desired spheres may then be created by editing the SPHGEN output.

If nothing else works, it is possible to run CLUSTER on all possible spheres rather than a preselected group. Use the analytical clustering algorithm in `cluster` on cluster 0, and experiment until you get what you want. Flagging spheres in important regions of the site may help.

Creating the Scoring Grids

GRID

GRID saves information about the steric and electrostatic environment at each point on a grid, so that ligand orientations can be scored rapidly during a DOCK run.

Positioning the Grid

You determine the location and dimensions of the region to be gridded by using the program SHOWBOX to create a box which contains the desired region. For GRID, the box should enclose the volume that the ligand orientations are likely to occupy. An easy way to accomplish this is to generate a box which encloses the spheres to be used for docking along with an extra margin. The box generated should be viewed along with the receptor and possibly regenerated until it looks good.

Preparing the Receptor File

First read the receptor PDB file into a text editor. Remove all waters and complexed ligands. Special attention should be given to the names of atoms at the termini and the residue names for histidine and cysteine. You will need to rename each histidine residue depending on the protonation state you want to assign it: HIP for positively charged (hydrogens on both nitrogens), HID for neutral with the delta nitrogen protonated, and HIE for neutral with the epsilon nitrogen protonated. CYS refers to a cysteine with a free sulfhydryl group; CYX refers to a cysteine involved in a disulfide bond (a half-cystine). Note that some structures in the PDB use CYS in disulfides; these should be edited to CYX.

Second the user must construct a SYBYL MOL2 format of the receptor which includes sybyl atom type and partial charge assignments. We routinely use SYBYL for this task, but other modeling packages can be used provided you have a way to convert the resulting receptor file into MOL2 format. The following instructions will apply to the use of SYBYL for this task.

In SYBYL, activate the BIOPOLYMER menu from the OPTIONS menu. From the BIOPOLYMER menu, select BROOKHAVEN READ to read in the receptor PDB file. A dialogue box will ask if you want to center the molecule. If you need to retain the reference frame of the receptor (e.g. for consistency with other collaborators) then don't center the coordinates. Instead, you will need to manually find the receptor since it will probably not appear on the screen. Hit the lower button on the left side of you SYBYL window which looks like a molecule surrounded by arrows. In the small window that appears, hit the button called "reset extents." Now you should see the receptor from a distance. Use the far right mouse button to rotate the receptor to its highest point on the screen. Use the middle mouse button to translate it to the center of the screen. Then use the combination of mouse buttons to zoom in on the receptor. You may need to rotate it up and translate back to the middle a few times to keep it from escaping the window. You will need to repeat this exercise every time you read in any molecule that is in the unperturbed frame of reference of the receptor (i.e. bound ligands).

Check if all of the atoms were identified properly by SYBYL. You can label problem atoms by selecting LABEL ATOMS. In the atom selection window, press the SET button. If you see a set type called "Unknown Atoms" then select it. Any atoms that were not recognized by SYBYL will now be labelled. This most often occurs with oxygen atoms at the C-terminus, with unusual amino acids, or if cofactors or ligands were not removed. If the terminal carboxyl oxygens are the problem, then rename them with the text editor and reread the receptor. For other problem atoms you will need to consult the SYBYL manual.

UCSF LIBRARY

Next, you should model in any incomplete residues. Check the original PDB file for a list of residues whose density was to weak to model completely. If no list exists, then check the total charge on each residue as reported by GRID when you run it later; if some residues have non-integer charges, then you may need to come back to SYBYL and model them in. To do this, identify the residue to fix. From the BIOPOLYMER menu, select MODIFY, then MUTATE RESIDUE. Click on the residue you want to modify, then select which type of residue you want to mutate it to (use the same type of residue).

Add hydrogens by using the BIOPOLYMER menu option. It is important to add ALL atoms, not just POLAR atoms, since GRID needs them to identify VDW atom types. Load charges from the BIOPOLYMER menu. You may use ALL atom or UNITED atom KOLLMAN charges. Write out the receptor to a MOL2 file by using the write option in the FILE menu.

Running GRID

Input to GRID is interactive. Just type `grid -i grid.in` to launch it. All input parameters will be saved in the file called grid.in. After all parameters have been input, hit CTRL-C to kill the job. Relaunch it in background mode by typing `grid -i grid.in -o grid.out&`.

All recommended parameter values will be suggested by GRID when run interactively, but here are a few suggestions. `grid_spacing` values between 0.2 and 0.5 are recommended; fine grids are preferred if there is sufficient memory in the computer. Any combination of grid point spacing and box size can be used, but it is recommended that about a million total grid points be used. Of course this value depends on memory resources.

A dielectric function of 4.0*r* or 4.5*r* and a cutoff of 10.0 Angstroms or more are appropriate in most cases. (This dielectric corresponds to specifying `distance_dielectric yes`, `dielectric_factor 4` or `4.5`, and `energy_cutoff_distance 10`.) If a constant dielectric is selected, an "infinite" cutoff (one large enough to include the whole receptor) should be used.

It is important to check the residue charges that are output by GRID. If any non-integer charges are reported, then some residues may have improper charges assigned to them, or they are not completely modeled in the input file. If no charged residues are reported, then check to make sure that charges were properly loaded in the input file.

Four output files, named `grid.bmp`, `grid.cnt`, `grid.chm`, and `grid.nrg`, will be produced which hold the bump grid, contact grid, chemical grid and force-field grid, respectively.

UCSF LIBRARY

Running DOCK

Starting a DOCK Run

You are now ready to run DOCK. Since DOCK can use a substantial amount of CPU time, it is a good idea to check whether there are other jobs running on the same machine. Consider any other users sharing your computers when deciding whether to start more than one run at a time. Be aware of any policies your site has regarding submitting background jobs.

The easiest way to select dock parameters is to run DOCK interactively. Do this by typing `dock -i dock.in`. You will be prompted for a value for each parameter. Any value you enter will be stored in the file `dock.in`. This file does not need to exist beforehand. If it does exist, then DOCK will extract all the relevant parameters it can find from the file. For each parameter, DOCK will supply a default value. If you want to use the default value, just hit return. The following tables list recommended values for running DOCK in two different ways: first to dock a single ligand, and second to dock a database of ligands. If you are viewing this manual on-line, then click on any of the keywords to view the reference entry for it.

Table 11. Recommended DOCK parameters for a new user.

11A: General

Keyword	Suggestions
chemical_screen	no
flexible_ligand	no; try later
orient_ligand	yes; searches ligand orientations
multiple_ligands	no
score_ligand	yes; scores each ligand orientation
minimize_ligand	no; try later
parallel_jobs	no; try later

11B: Orientation Search

Keyword	Suggestions
match_receptor_sites	yes; matches ligand to site points
match_ligand_centers	no
random_search	no
uniform_sampling	yes; otherwise need to enter geometric match parameters
total_orientations	500; number of orientations to try
write_orientations	yes; to write out multiple orientations for single molecule
rank_orientations	yes; to save the top scoring orientations
rank_orientation_total	10; to save the top 10 orientations

11C: Matching

Keyword	Suggestions
critical_points	no; try later
chemical_match	no; try later

11D: Scoring

Keyword	Suggestions
intermolecular_score	yes
gridded_score	yes
grid_version	4
bump_filter	yes
bump_maximum	3
contact_score	no
chemical_score	no
energy_score	yes
atom_model	u; for united atom model
vdw_scale	1
electrostatic_scale	1

11E: Input

Keyword	Suggestions
ligand_atom_file	Enter the ligand MOL2 file name here (including the directory path if this file is not in the current directory).
receptor_site_file	Enter the SPHGEN site point file name here.
score_grid_prefix	Enter the GRID file name prefix here.
vdw_definition_file	/dock/parm/vdw.defn

11F: Output

Keyword	Suggestions
ligand_energy_file	dock_nrg.mol2

Table 12. Recommended beginner's DOCK parameters for a database search run

12A: Parameters to Modify from Database Run

Keyword	Previous	Suggestions
multiple_ligands	no	yes; to consider multiple molecules
write_orientations	yes	no; to write only the best orientation for each molecule

12B: Multiple Ligands

Keyword	Suggestions
ligands_maximum	first try 10 to make sure everything is working, then set it to a number larger than the number of molecules in input file
initial_skip	0; **n**>0 will skip the first **n** molecules in input file
interval_skip	0; **n**>0 will skip **n** molecules for each molecule processed
heavy_atoms_minimum	4; to discard small molecules
heavy_atoms_maximum	50; to discard large molecules
rank_ligands	yes; to save a top score list
rank_ligand_total	50; to save the top 50 molecules
restart_interval	100; to save restart info every 100th molecule processed

12C: Additional Input and Output

Keyword	Suggestions
quit_file	dock.quit
dump_file	dock.dump
info_file	dock.info
restart_file	dock.rst

If you happen to enter the wrong value for any parameter and wish to change it, then you may edit the dock.in file directly and modify the parameter value. Once all parameters have been

UCSF LIBRARY

entered, DOCK should begin the calculation and the `dock.in` file is complete. You may kill the process with a CTRL-C and relaunch the process in background by typing "`dock -i dock.in -o dock.out&`". If you would like to run DOCK multiple times from the same directory, then you may use a different name for `dock.in` and `dock.out`. Just be sure to change the names of the output files inside the new `dock.in` file so that two processes don't end up overwriting each other's output files.

Check a few minutes after you start the run to be sure that it is still going; if it has stopped, look for mistakes in the input. Beginners should check disk usage occasionally while the job is running, just in case the program is creating incredibly large files which might overflow the available space.

During a database search run (which can take anywhere from hours to days to weeks to finish), you can follow DOCK's progress through the database by inspecting the `*.info` file.

Restarting a Search Run

In database search mode of DOCK (when `multiple_ligands`, `orient_ligand`, and `rank_ligands` are selected), DOCK periodically saves information necessary to restart the search from its current location in the database in a `*.rst` file. If there is a power failure or the system crashes, you can set up a new run to start where the last one was stopped. First, make a copy of `dock.out` so that status of the previous run are saved. Then relaunch the job using the `-r` flag at the command line. (Do not change the remaining files, since DOCK needs them to restart successfully.) When the restarted run finishes, the sorted list of ligands in the output file will include the top scorers from the entire database.

Looking at the Results

DOCK puts the resulting molecule orientations in a file for each type of scoring function used. The scores are given in the comment records at the beginning of each molecule record. If you have selected MOL2 format for your output files and your graphical viewer does not read this format, then convert the file to PDB by typing `dock -i dock_nrg.mol2 -o dock_nrg.pdb`.

Other Post-Docking Tasks

Depending on your particular project, you might be interesting in any one or several of the following post-docking techniques:

- Rescoring of hits with alternative scoring function;
- Redocking of hits with increased orientation sampling and/or conformational sampling;
- Similarity searching based on hits; or
- Further molecular modeling/molecular dynamics/FEP of hits;

Advanced Techniques

written by Todd Ewing

Introduction

This section of the manual provides a discussion of many of the advanced features available in DOCK. It is intended for users who already have some familiarity with using DOCK.

These features include:

- Orientation Search
- Conformation Search
- Scoring
- Database Processing
- Chemical Screening
- Macromolecular Docking

Orientation Search

DOCK version 4.0 has a new orientation search algorithm, or matching algorithm, which is more robust than before (see Ewing and Kuntz [6]). An orientation search is requested with the `orient_ligand` parameter. The amount of orientation sampling can be controlled in two ways:

- Uniform Sampling —Specify the number of orientations, and dock will generate matches until enough orientations passing the bump filter have been formed. Matches are formed best first, with respect to the difference in the ligand and site point internal distances.
- Matching —Specify the distance and node parameters, and DOCK will generate all the matches which satisfy them. The number of orientations scored is equal to the total matches minus the orientations discarded by the bump filter.

There are a number of sophisticated options available to tailor the orientation search. These options include:

- Random Search
- Degeneracy Checking
- Ligand Mirroring
- Chemical Matching
- Critical Points

Multiple orientations may be written out for each molecule using the `write_orientations` parameter, otherwise only the best orientation is recorded. A ranked list of the orientations may be written using the `rank_orientations` parameter. Otherwise, all orientations passing a score cutoff are written out. The score cutoff is specified with the `contact_maximum` parameter and so on for each type of scoring. If `write_orientations` is requested without scoring, then all orientations are written.

UCSF LIBRARY

Uniform Sampling

With uniform_sampling, DOCK performs the same amount of orientation searching on each molecule. If the match_receptor_sites parameter is set, then matching is used as a black box engine for the orientation search (otherwise a random search is performed). The only sampling parameter needed is the total_orientations parameter, which is the number of desired orientations which survive the bump filter. Matches are formed in order of the smallest distance error first, so that the highest quality orientations are guaranteed to come sooner rather than later. This method of control is incredibly easy. It is useful for most applications of docking, particularly when docking a single molecule. It is also useful for flexible docking using the anchor_search option, since otherwise it is difficult to find match parameters that provide good sampling for a range of anchor fragment sizes.

Matching

If the match_receptor_sites parameter is set but not the uniform_sampling parameter, then matching is performed and controlled by the match parameters listed in Table 13. The matching parameters provide an intuitive way to control sampling. When multiple molecules are docked, matching will bias sampling towards molecules with more internal distance similarity with the receptor site points. The additional chemical and critical matching constraints provide a way to prune matching and further bias sampling towards more interesting molecules.

Table 13. Description of Matching Parameters

distance_tolerance	The distance tolerance can be viewed as the uncertainty in the distance comparisons or site point positions. The more generous the uncertainty in the distance comparisons, the more sampling will be performed. This parameter should be the first parameter to adjust if you need to change the amount of sampling.
distance_minimum	The distance minimum allows matching to focus on the longer distances which convey more information about molecule or site shape. This value can be conveniently set large enough to discard atoms directly bonded to each other. When docking large molecules, this value can be set higher.
nodes_minimum	The minimum number of nodes must be at least three to specify a unique rigid transformation. A value of four or more will allow every match to include information about chirality. Match chirality can be used to explore the mirror image of a molecule for docking. The higher this parameter, the better the ligand atoms in the match represent the entire molecule.
nodes_maximum	This value may be set arbitrarily high to prevent it from influencing matching. It may be set equal to the nodes minimum when performing pharmacophore-style matching if only a few specific site interactions are of interest.

Random Search

The random_search option is intended for advanced users. If match_receptor_sites is also set then random matching is performed, in which ligand centers and receptor sites are randomly matched regardless of internal distances. Otherwise, a random transformation search

is performed, in which ligands are randomly rotated and translated within the rectangular box enclosing all the site points.

Site Point Construction

The random_search option is useful for exploring issues relating to site point construction. As discussed in Ewing and Kuntz [6], both random matching and random transformation were useful control algorithms to test the effectiveness of distance-based matching. The relative performance of random matching with respect to random transformation indicates how well the site points map out the relevant volume of the active site. The relative performance of distance-based matching with respect to random matching indicates how well individual positions of each site point correspond to good ligand atom positions. By using both of these search methods, an advanced user may quantify the quality of site points constructed by alternative methods to SPHGEN.

The random transformation search may in fact be used to construct site points to supplement those from SPHGEN. Using this search, the user may probe a site with different molecular probes much like the atomic probes used in Goodford's GRID program. The best-scoring positions may then be used to position site points.

Degeneracy Checking

Degeneracy checking is a method implemented during matching to increase the diversity of the resulting orientations. It is selected with the check_degeneracy parameter. It is not an available feature if uniform_sampling has been selected. The method of Gschwend and Kuntz [11] implemented in dock version 3.5 has been updated to be easier to use and more robust. Degenerate matches are now defined as matches which are a subset of a larger match. In the nomenclature of graph theory, the surviving matches are maximally connected and are true cliques.

For degeneracy checking to work, nodes_maximum must be greater than nodes_minimum so that subsets can occur. In general, just set nodes_maximum arbitrarily high (15 or so). At most a two-fold reduction in matches is achieved using this feature.

Ligand Mirroring

When a match contains four or more nodes, the chirality of the ligand and receptor points involved in the match is checked. Half of the time, the ligand and receptor points have opposite chirality. See Ewing and Kuntz [6] for more discussion. Normally these improper matches are discarded, but they can be rescued with the reflect_ligand option, which allows the chirality of the ligand to be reversed by using its mirror image. This is useful for molecules which are either achiral or are available as a racemate.

Chemical Matching

The chemical_match feature is used to incorporate information about the chemical complementarity of a ligand orientation into the matching process. As in Kuhl et. al [15], chemical labels are assigned to site points and ligand atoms. The site point labels are based on the local receptor environment. The ligand atom labels are based on user-adjustable chemical functionality rules. These labeling rules are identified with the chemical_definition_file parameter and reside in an editable file (see chem.defn on page 168). A node in a match will produce an unfavorable interaction if the atom and site point components have labels which violate a chemical match rule. The chemical matching rules are identified with the chemical_match_file parameter and reside in an editable file (see chem_match.tbl on page 169). If a match will produce unfavorable interactions, then the match is discarded. The speed-up from this technique depends how extensively site points have been labeled and the stringency of the match rules, but an improvement of two-fold or more can be expected.

UCSF LIBRARY

The process of labeling site points must currently be done by hand. The user should load the site points and the receptor coordinates into a graphic program and study the local environment of each point. Developing an automated method to perform this task is still an active area of research. Labeled site points may be input as either a SPH format or SYBYL MOL2 format coordinate file. Check SPHGEN on page 147 for file format specifications. An example is shown in Table 14. To store labeled site points in a MOL2 file, select an atom type for each label of interest. Then edit the chem.defn file to include the selected atom types. Site point definitions can be distinguished from ligand atom definitions by explicitly requiring that no bonded atoms can be attached. The example chem.defn on page 168 includes a site point definition as the last definition for each label. Using the convention in that example file, site points should be labeled as follows: hydrophobic, "`C.3`"; donor, "`N.4`"; acceptor, "`O.2`"; polar, "`F`".

Table 14. Example of chemical labels in SPH format

```
DOCK 3.5 receptor_spheres
color hydrophobic       1
color acceptor          2
color donor             3
cluster     1   number of spheres in cluster    49
    7   2.34500  36.49000  16.93500   1.500    0 0  1
    8  -0.05200  42.29900  14.18800   1.500    0 0  1
    9  -0.67000  41.20600  11.59800   1.500    0 0  1
   17  -6.00000  34.00000  17.00000   1.500    0 0  3
   18  -5.00000  29.00000  22.00000   1.500    0 1  3
                            ...
```

Caveats

It can take a significant amount of effort to chemically label a large site and to verfiy that the docking results are what were expected. If you use this chemical matching, plan to spend some time in preparation and validation BEFORE running an entire database of molecules.

In concert with degeneracy checking, chemical matching is able to discard matches that not only contain bad interactions but that can be expanded to include other bad interactions. Although this helps reduce the bad interactions in an orientation, it can only do so within the constraints of the `distance_tolerance`, which can be rather tight. In addition, the number of interactions monitored in a match is usually small (3-5) compared to the total number of ligand atoms, so the preponderance of atoms may be in less than favorable environments. Therefore, chemical matching does not guarantee that all resulting orientations are chemically complementary, but instead that the resulting orientations are *enriched* in complementarity.

It must be pointed out that the ultimate arbiter of which orientations of a ligand are saved is actually the scoring function. If the scoring function is unable to discriminate what the user feels are bad chemical interactions, then any improvement with chemical matching will probably be obscured. In addition, if score optimization is used, then the orientation will be perturbed from the original chemically-matched position to a new score-preferred positions.

Critical Points

The `critical_points` feature is used to focus the orientation search into a subsite of the receptor active site [4, 23]. For example, identifying molecules that interact with the catalytic residues might be of chief interest. Any number of points may be identified as critical, and any number of groupings of these points may be identified. Consequently, several receptor subsites may be targeted simultaneously. If a particular cluster of critical points is big enough to interact with more than one ligand atom, then use the `multiple_points` parameter. An al-

ternative to using critical points is to discard all site points that are some distance away from the subsite of interest, while retaining enough site points to define unique ligand orientations.

This feature can be highly effective at reducing matching by five-fold or more. It is particularly useful to also assign chemical labels to the critical points to further focus sampling.

Conformation Search

The conformation of a flexible molecule may be searched or relaxed using the `flexible_ligand` option. Only the torsion angles are modified, not the bond lengths or angles. Therefore, the input geometry of the molecule needs to be of good quality. A structure generated by CONCORD is sufficient.

A flexible molecule is treated as a collection of rigid segments. The first step in segmentation is ring identification, and is illustrated in Figure 18. All bonds within molecular rings are treated as rigid. This classification scheme is a first-order approximation of molecular flexibilty, since some amount of flexibility can exist in non-aromatic rings. To treat such phenomenon as sugar puckering and chair-boat hexane conformations, the user will need to supply each ring conformation as a separate input molecule. Later versions of dock may treat these phenonmenon in an automated fashion.

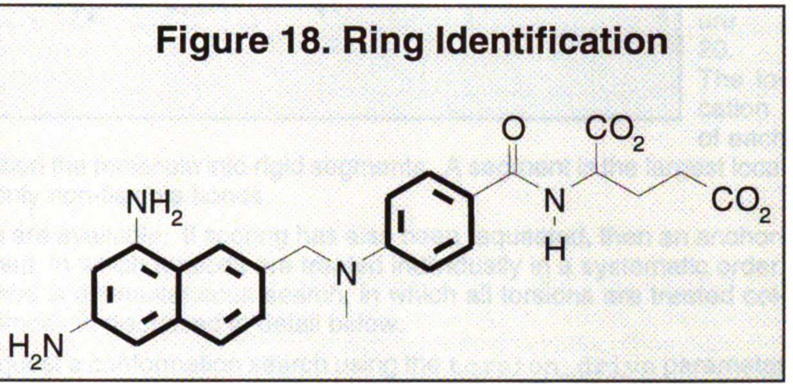


Figure 18. Ring Identification

Additional bonds may be specified as rigid by the user. This is used to isolate a portion of the molecule from the conformation search. These bonds are identified in the SYBYL MOL2 format file containing the molecule. The bonds are designated as members of a STATIC BOND SET named RIGID. Please see SYBYL MOL2 format on page 161 for an example of such an identification.

Figure 19. Flexible Bond Identification

The second step is flexible bond identification, and is illustrated in Figure 19. Each flexible bond is associated with a label defined in an editable file (see flex.defn on page 172). The parameter file is identified with the `flex_definition_file` parameter. Each label in the file contains a definition based on the atom types (and chemical environment) of the bonded atoms. Each label is also flagged as minimizable. Typically, bonds with some degree of double bond character are excluded from minimization so that planarity is preserved. Each label is also associated with a set of preferred torsion positions.

UCSF LIBRARY

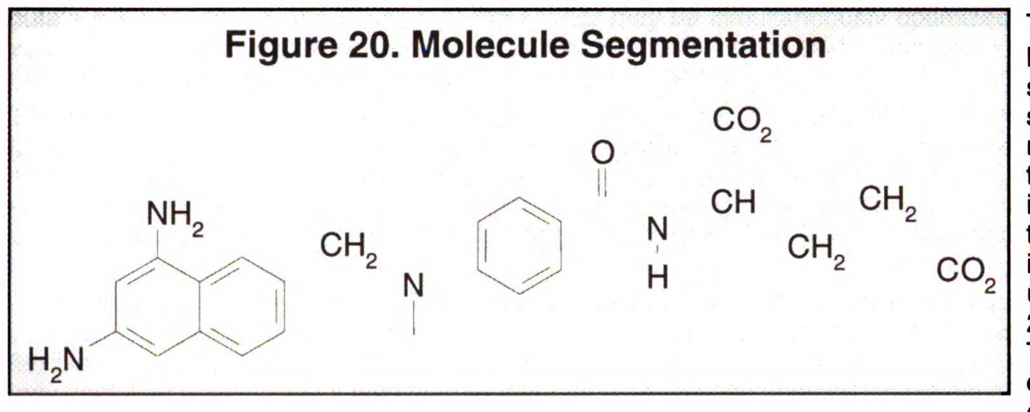


Figure 20. Molecule Segmentation

The last step in segmentation is illustrated in Figure 20. The location of each flexible bond is used to partition the molecule into rigid segments. A segment is the largest local set of atoms that contains only non-flexible bonds.

Two types of flexible search are available. If scoring has also been requested, then an anchor-first search may be performed, in which torsions are treated individually in a systematic order. Otherwise, the default method is a simultaneous search, in which all torsions are treated collectively. Each of these methods is discussed in detail below.

In addition, the user may request a conformation search using the `torsion_drive` parameter and/or torsion minimization using the `torsion_minimize` parameter. The torsion angle positions reside in an editable file (see flex_drive.tbl on page 173) which is identified with the `flex_drive_file` parameter. Internal clashes are detected during the torsion drive search based on the `clash_overlap` parameter, which is independent of scoring function.

If `multiple_ligands` are being processed, then the `flexible_bond_maximum` cutoff is used to discard overly flexible molecules.

When scoring is requested, the user has the option of computing intramolecular terms using the `intramolecular_score` parameter. For the sake of speed, only the interactions between segments is considered. If a segment has not moved, then the contribution of its interaction with the receptor to the intermolecular score is not recalculated. If any two segments have not moved, then the contribution of their interaction to the intramolecular score is not recalculated.

Anchor-First Search

The anchor-first search is an efficient divide-and-conquer algorithm based on the method of Leach and Kuntz [19] and the Greedy algorithm. It is specified using the `anchor_search` parameter. An anchor segment is selected from the rigid segments either manually or automatically. Manual selection of the anchor is performed by identifying the anchor atom using a STATIC ATOM SET named ANCHOR in the molecule input file. For an example, please see SYBYL MOL2 format on page 161. If such a set was not designated in the input file, then automatic selection is performed by default. If the `multiple_anchors` parameter is set, then all segments which pass the `anchor_size` cutoff are tried separately as anchors. Otherwise, only the largest rigid segment is used as an anchor. When an anchor has been selected, then all other segments are partitioned into layers arranged concentrically about the anchor segment.

The anchor is processed separately (either oriented, score, and/or minimized). The remaining segments are subsequently re-attached if the `peripheral_search` parameter is set. This step may be left out to perform a preliminary analysis of how a flexible ligand might dock. The minimization of the anchor is selected with the `minimize_anchor` parameter. The peripheral segments are re-attached one at a time, in the order of nearest layer first, and within a layer, largest segment first. If `torsion_drive` has been selected, then the torsion positions of the intervening bond are searched. If `torsion_minimize` has been selected, then the intervening torsion may be relaxed. Minimization of the bond is performed in isolation, or in concert with inner torsions if the `reminimize_layer_number` parameter is set to a non-zero value. Relaxing multiple layers helps prevent the search from getting stuck in dead-ends. Although com-

UCSF LIBRARY

putationally expensive, the position of the anchor may be simultaneously optimized with the `reminimize_anchor` parameter. When all segments have been added, the entire molecule may be relaxed if the `reminimize_ligand` parameter is set.

The extent of the peripheral search is under full user control. After each stage of growth, the best, most diverse conformations and orientations are retained to seed the next stage. The number of seeds retained is set by the `peripheral_seeds` parameter. The best, most diverse configurations are found using a multiple-linkage, hierarchical clustering algorithm based on a rank-weighted RMSD comparison of the configurations. The best scorer of each cluster becomes a seed.

As the number of seeds is increased, the anchor-first method approaches an exhaustive search. The time demand grows linearly with the number of peripheral seeds, the number of flexible bonds and the number of torsion positions per bond. This search technique is particularly useful for docking, but it also may be used for conformation analysis and stand-alone minimization.

Simultaneous Search

If an anchor-first search is not selected, then a simultaneous search is performed by default. All torsions are searched and/or minimized in concert. The conformation search is performed prior to the orientation search, so each conformation is docked independently. The extent of the search is controlled with the `maximum_conformations` parameter. If the total conformations for a molecule is below this cutoff, an exhaustive search is performed. Otherwise, a random search of the maximum conformations is performed, in which torsion positions are selected randomly from the allowed positions. The time demand grows exponentially with the number of flexible bonds up to the maximum conformations. This search technique is useful for conformation analysis and for constructing a chemical screen database (see Chemical Screening on page 108).

Torsion Minimization

The torsion angles of rotatable bonds may be included in the score optimization. This provides for a much more efficient conformation search since fewer torsion positions need to be sampled. Each torsion flagged for movement is assigned a simplex vertex along with the six rigid body degrees of freedom. Only non-bonded interatomic terms are included in the scoring evaluation; no explicit torsion terms are included. Therefore, only torsions flagged as minimizable in the flex.defn file are included (e.g. double bonds are excluded by default). When an anchor-first search is performed with segments from multiple layers being minimized, then inner torsions are assigned smaller initial torsion step values since perturbations in these torsions have a greater impact on conformation.

Scoring

DOCK uses several types of scoring functions to discriminate among orientations and molecules. Scoring is requested using the score_ligand parameter. Each scoring function is treated independently during the calculation and results are written to separate output files. In order to combine the results of two or more scoring functions, apply the additional functions in separate post-docking scoring calculations.

DOCK will score the interactions of a ligand with the receptor if the intermolecular_score parameter is set. If flexible_ligand is set, then DOCK will score the interactions between rigid segments within a molecule if the intramolecular_score parameter is set. The total score is the sum of the intramolecular score and the intermolecular score, EXCEPT when different molecules are being compared for a rank_ligands list. After a flexible molecule has been docked, and it is being considered for the ranked ligand list, then the total score is set equal to only the intermolecular score plus whatever size penalty the user has specified with the contact_size_penalty parameter and so on for each type of scoring.

To enable rapid score evaluation during docking, the score potentials are precalculated on a three-dimensional grid using GRID. However, continuum scoring may be performed by turning off the gridded_score flag. Continuum scoring may be used to evaluate a ligand:receptor complex without the investment of a grid calculation, or to perform a more detailed calculation without the numerical approximation of the grid. Continuum scoring is also triggered when an intramolecular_score is requested, but is only used for intramolecular score terms. When continuum scoring is requested, then score parameters normally supplied to GRID, must also be supplied to DOCK. It is left to the user to make sure consistent values are supplied to both programs. Older grids calculated by CHEMGRID may also be read by specifying a value of 3.5 for the grid_version parameter.

The score is used to identify interesting orientations and molecules. If a top-scoring list is requested, like rank_orientations or rank_ligands, then DOCK will maintain a sorted list of a user-defined length for output. But if sorted lists are not requested, then DOCK will need to know what score cutoff to use to write out orientations or molecules. This cutoff is supplied by the user with the contact_maximum parameter and so on for each scoring function. This score cutoff may be overridden for orientations near the input orientation using the rmsd_override parameter.

Bump Filter

Orientations may be filtered prior to scoring to discard those in which the molecule significantly overlaps receptor atoms. This feature is enabled with the bump_filter flag, but is only available if the gridded_score flag is also set. At the time of construction of the bump filter, the amount of atom VDW overlap is defined with the bump_overlap parameter. At the time of bump evaluation the number of allowed bumps is defined with the bump_maximum parameter. If score optimization is being performed, then a few number of bumps should be allowed, since the minimizer can recover from such clashes. In addition, a few bumps often indicate an orientation that interacts intimately with the site and often leads to a strongly favorable orientation after minimization.

Contact Score

The contact score function is enabled with the contact_score flag. The contact score is a simple summation of the number of heavy atom contacts between the ligand and receptor. At the time of construction of the contact scoring grid, the distance threshold defining a contact is set with the contact_cutoff_distance. Atom VDW overlaps are penalized by checking the bump filter grid, or with the contact_clash_overlap parameter for the intramolecular score. The amount of penalty is specified with the contact_clash_penalty parameter.

UCSF LIBRARY

The contact score provides a simple assessment of shape complementarity. It can be useful for evaluating primarily non-polar interactions.

Energy Score

The energy score is activated with the `energy_score` parameter. It is based on the non-bonded terms of the molecular mechanic force field (please refer to Equation 4 on page 140 for more background). During grid construction (and continuum scoring) every term in the function may be tailored by the user. The distance dependence of the Coulombic function is set with the `distance_dielectric` parameter. The dielectric constant is adjusted with the `dielectric_factor` parameter. The distance dependence of the Lennard-Jones function is set with the `attractive_exponent` and `repulsive_exponent` parameters. Typically a 6-12 potential is used, but it can be softened up by using a 6-8 or 6-9 potential. Regardless of the exponent values selected, the same radii and well-depths are used. The VDW well-depths and radii are stored in an editable file (see vdw.defn on page 167) which is identified with the `vdw_definition_file` parameter. In addition, the model for non-polar hydrogens may be selected with the `atom_model` parameter. With a united-atom model, the non-polar hydrogens are given zero VDW potentials and any partial charge residing on them is transferred to the adjacent carbon. The united atom model provides a smoother intermolecular potential which requires fewer steps of minimization. However, the all-atom model is more accurate, and perhaps captures some aromatic interaction coulombic terms otherwise missing.

Chemical Score

Chemical scoring allows the energy scoring function to be further tailored to enhance recognition of chemical complementarity. The attractive portion of the VDW term can often dominate the energy for uncharged molecules. With chemical scoring this term is scaled depending on the chemical labels assigned to the interacting atoms. It is activated by the `chemical_score` parameter. The chemical labels and definitions are the same as those for chemical matching (see chem.defn on page 168). The chemical interaction table resides in an editable file (see chem_score.tbl on page 170) and is identified with the `chemical_score_file` parameter.

Chemical scoring can be used to incorporate qualitative aspects of solvation. For instance hydrophobic-polar interactions can be made non-attractive or even repulsive. Further, it can be used to screen for molecules that contain a particular functional group (in concert with chemical matching) for presentation to a receptor active site. The interaction table could also be derived using statistical techniques from binding and structure data to improve the modeling of a particular site or class of sites.

Chemical scoring is used for intermolecular scoring only. If intramolecular score is requested, then the regular energy score is computed for internal energy.

This type of scoring should be considered experimental. Parameterization is left to the user. It should be used at your own risk.

RMSD Score

The RMSD score evaluates of the difference in conformation and orientation of two identical molecules. It is available when `gridded_score` is not requested and is activated by the `rmsd_score` parameter. The reference molecule is supplied with the `receptor_atom_file`, and the molecule to check is supplied with the `ligand_atom_file`, which may contain multiple molecules.

Since the RMSD is treated as a score, it may in fact be minimized. This procedure is useful for evaluating the difference of a CONCORD conformation from crystal conformation.

UCSF LIBRARY

Score Optimization

Score optimization allows the conformation and orientation of a molecule to be adjusted to improve the score. Although the calculation is expensive, it makes the conformation and orientation search more efficient because less sampling becomes necessary. Optimization is activated with the `minimize_ligand` parameter. The optimizer currently uses the simplex algorithm which does not require evaluation of derivatives. It does however depend on a random number generator which makes it not only sensitive to the initial seed provided with `random_seed` parameter, but also to the order of evaluation. So results will vary if molecules are supplied in a different order. The amount of variance should be small, though. For detailed calculations, it is recommended that the optimization be repeated with different random number seeds to check convergence.

The initial step size of the minimizer is specified with the `initial_translation`, `initial_rotation`, and `initial_torsion` parameters. The length of minimization may be controlled with the `maximum_iterations` parameter.

Even if several scoring functions have been requested, not all need to be minimized. Specification of which functions are to be minimized is done with `contact_minimize` and so forth for each scoring function. The termination criteria of minimization is specified with `contact_convergence` and so forth.

Since minimization may converge prematurely, each call to the minimizer is actually composed of multiple cycles. The number of cycles is controlled with the `maximum_cycles` parameter. Additional cycles of minimization are spawned if the previous simplex has made a significant difference in the conformation or orientation AND if the score has passed below a threshold set by the user. The difference in configuration is measured by the vector magnitude of the final simplex vertex array. The difference must exceed the `cycle_convergence` parameter to be significant. A value of 1.0 for this parameter would correspond to at least one of the simplex vertices moving a distance equal to the initial step size. The score threshold prevents repeated cycles of minimization of a configuration that is really of no interest. The score threshold is set with `contact_termination` an so forth for each scoring function.

The performance of the minimizer can be monitored using the `-p` flag (see Command-line Arguments on page 117 and Performance on page 139).

UCSF LIBRARY

Database Processing

The most common application of DOCK is to process a database of molecules to find potential inhibitors or ligands of a target macromolecule. However, with the new separation of components in version 4.0, the database processing tools can be combined with other tasks, like stand-alone scoring, score optimization, or chemical screening.

Database processing is signalled with the `multiple_ligands` parameter. A subset of the database may be processed using the `ligands_maximum`, `initial_skip`, and `interval_skip` reading parameters and the `heavy_atoms_minimum` and `heavy_atoms_maximum` size-selection parameters. If scoring has been selected, then molecules can be output as a ranked list using the `rank_ligands` parameter. When comparing molecules, the score of large molecules may be penalized using the `contact_size_penalty` parameter and so on for each scoring function. If ligands are not ranked, then all orientations recorded for each molecule are written (see Orientation Search on page 94 for how multiple orientations are recorded). When no orientation search is performed (i.e. stand-alone scoring), then molecules are written if they pass a score cutoff set by the `contact_maximum` parameter and so on for each scoring function.

Database jobs produce two output files in addition to the molecule output files. The `restart_file` parameter specifies the file which stores the current `rank_ligands` list. If the job is terminated prematurely, then it may be restarted with the `-r` flag (see Command-line Arguments on page 117) and the run is initialized with information in the existing restart file. The frequency at which the restart file is updated is specified with the `restart_interval` parameter. In addition, the `info_file` parameter specifies the file which stores information about the current progress of the run.

Database jobs may also be interacted with during execution via the presence of two input files. The `dump_file` parameter specifies a file whose presence will trigger the job to write out the current results to the info file, the restart file and the molecule output files. The user may create this file at any time to inspect the current results. The dock job will automatically delete the dump file after a dump has taken place. In addition, the `quit_file` parameter specifies a file whose presence will trigger the job to write out results (like a dump request) and also terminate execution. This parameter is useful for gently terminating a job without loss of information for restart at a later time. The presence of either of these files is only checked in between processing molecules, so it may take up to a minute for such a file to be noticed.

Preliminary Docking

Docking an entire database can take a considerable amount of time. The length of time depends primarily on the sampling parameters for the orientation and conformation search and the minimization parameters. Even when docking is distributed over multiple workstations, the calculation can take several days or weeks. Since the optimal search parameters are site dependent, it is important to do some preliminary docking calculations with subset of the database to identify good parameters.

As sampling parameters are increased, the results will initially improve but will eventually converge. The optimal parameters correspond to where the results have just converged. Multiple short docking jobs can be submitted using UNIX shell scripts. The results that should be monitored are presented in Ewing and Kuntz [6]. The most important is the weighted rank correlation, which reports how well the rankings of the top-scoring molecules have been predicted.

The following is a discussion of different ways to construct a subset of the molecule database.

Extracting specific molecules

Since the PTR format database file contains the molecule name and description, entries can be retrieved based on these fields. Use UNIX `fgrep` to select the molecule, or molecules.

UCSF LIBRARY

```
fgrep " BENZENE " database.ptr > benzene.ptr
fgrep -f subset.list database.ptr > subset.ptr
```

The file `molecules.list` would contain a list of the names you would like to extract. The subset molecules can be readily converted to a format for viewing with the following command.

```
dock -i subset.ptr -o subset.pdb
```

Extracting a random subset

A random subset of a molecular database can be used to help identify appropriate docking parameters, before docking an entire database. Selecting a random subset is easy using a PTR format database file. Use the following UNIX `nawk` command to select an average of one out every 1000 entries in the database.

```
nawk '{if (rand < 0.001) print}' database.ptr > subset.ptr
```

Extracting an interval subset

Alternatively, if you literally want one molecule for every 1000 molecules without any randomness, then use a different call to UNIX `nawk`.

```
nawk '{if ((++n % 1000) == 0) print}' database.ptr > subset.ptr
```

This can also be achieved by using the `interval_skip` multiple ligand parameter in dock. This latter method is much slower, however, because the coordinates of all the skipped molecules must be read.

Parallel Jobs

Since a database docking calculation is ideal for parallelization over multiple computers, the parallel jobs feature was added to ease the organizational burden of this task. This feature is activated with the `parallel_jobs` parameter. With this feature, a dock job can be launched on every workstation or cpu at a user's disposal. These jobs process a single database, each at its own pace. To prevent each job from duplicating each other's work unnecessarily, a server job is used to parse the database and hand out molecules, one at a time, to each client job. Each client job and the server job requires its own input file and output files. When all processing is complete, the user must coalesce the results from each client job.

Client or server behavior is designated using the `parallel_server` parameter. Setting it to "`yes`" causes server behavior; "`no`" causes client behavior. The client name is defined with the `server_name` parameter. Any number of client jobs can be delegated to the server using the `client_total` parameter. Each client name must be supplied with the `client_name_1` and subsequent parameters. It is recommended that the server job be executed on the computer which stores the molecule database. Then in the event of any network difficulties, the server job is never disconnected from the database.

The client jobs need to the have the `parallel_jobs` parameter activated, but not the `parallel_server` parameter. The `server_name` parameter must be consistent with that supplied to the server job. The name of the client job is specified with the `client_name` parameter, which must one of the client names supplied to the server. The client jobs need to be launched from the same directory as the server job since they communicate via local temporary files. This requires that client jobs can only be launched on machines that cross-mount the working directory. Clients may be taken off-line (via the `quit_file`) and restarted without disrupting the server or other client jobs. If the server job is given a quit signal, then it automatically signals all client jobs to quit as well.

Client jobs may be instructed to either store a ranked list or to write out all results to file. Since the PTR format files take up so little disk space, an entire database can be written out without taken up more space than the top few thousand molecules written in SYBYL MOL2 format (about 25 megabytes of disk space). If the clients store ranked lists, then make sure that the list length for each client is equal to the total length of interest (a few hundred at least). This rule helps avoid artifacts from the parallelization when the results are coalesced.

UCSF LIBRARY

When all jobs are complete, then the results must be combined. This process can be done seamlessly by using the UNIX `cat` command on the output molecule files. If PTR format is used, then molecules can be reranked using the UNIX `sort` command on the score field. If SYBYL MOL2 format is used, then perform stand-alone scoring on the catenated file and output the molecules in a ranked list.

UCSF LIBRARY

Chemical Screening

The chemical screening option enables rapid screening of molecules either prior to or following docking. It is requested with the `chemical_screen` parameter. It can be used to perform a pharmacophore screen or a similarity screen. It is based on the same chemical labels and definitions as chemical matching and the chemical score (see chem.defn on page 168). These tools should be viewed as rudimentary in functionality compared to what is available in commercial small-molecule software packages. However, considering their ease of use and compatibility with DOCK, they should be of interest to the DOCK user.

The chemical search key encodes information about the number of distances between chemical groups in a molecule and the magnitudes of the distances. The number of distances between two different chemical groups, n_{ij}, records information about the composition of the molecule. The distances themselves are stored in a binary fingerprint, f_{ij}, which records information about the spatial distribution of properties in the molecule. Since the binary fingerprint may lose information about the frequency of occurrence of certain distances, particularly for flexible molecules, the number of distances record is necessary.

Keying a database is performed with the `construct_screen` parameter. The fingerprint architecture is specified with the `distance_begin`, `distance_end`, and `distance_interval` parameters. When keying the database, a simultaneous search of conformations should be performed to generate an ensemble of conformers. For each conformer, the distances between chemical groups is computed and stored in the binary fingerprint. The keyed database is actually a PTR format database file with the additional fields for chemical keys. Once the database has been keyed, it may then be searched using the `screen_ligands` parameter.

Pharmacophore Screen

A pharmacophore search is a useful way to prescreen a database to identify molecules that might interact favorably with a small number of receptor atoms with well-defined geometry. It is requested with the `pharmacophore_screen` parameter. The pharmacophore is actually a set of points with associated chemical labels serving as a three dimensional hypothesis for a binding model. The pharmacophore could be constructed from a set of active ligands, or extracted from a receptor structure. This feature is useful when only a few interactions with the receptor are of interest and can be used to construct a pharmacophore. The chemical screen would then be able to discard all molecules that would never be able to make the desired interactions.

Given a molecule ***a*** and a pharmacophore ***b***, the rules for determining whether the molecule might satisfy the pharmacophore are for all *i* and *j* not equal to *i*:

$$n_{ij}^{b} \le n_{ij}^{a} \qquad \text{Equation 1}$$

$$f_{ij}^{b} = f_{ij}^{a} \cap f_{ij}^{b} \qquad \text{Equation 2}$$

where n_{ij} and f_{ij} are defined in the Chemical Screening section, above.

In order to account for uncertainty in the distances, a `distance_tolerance` is used to blur the distance key of the molecule.

After the chemical screen run has discarded molecules that could never satisfy the pharmacophore, then a regular dock run can be performed to find the molecules that do satisfy it. In the follow-up run, use a simultaneous search of conformations along with chemical matching to the pharmacophore site points.

Similarity Screen

A similarity search is useful to identify all molecules in a database which might be similar in chemical property distribution to a particular molecule of interest. Such a search is useful after docking, once a set of active molecules has been identified. It is activated with the `similarity_screen` parameter. The cutoff for writing out hits is specified with the `dissimilarity_maximum` parameter. Dissimilarity is used to be consistent with the other scoring functions in which smaller values represent favorable values.

Since the chemical key contains both a binary fingerprint and an integer count of distances, a modified Tanamoto index is used as shown in Equation 3. This similarity metric maintains the connection between the fingerprint and the count of chemical distances.

$$Similarity = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} min(n_{ij}^{a}, n_{ij}^{b})\ count(f_{ij}^{a} \cap f_{ij}^{b})}{\sum_{i=1}^{N} \sum_{j=1}^{N} max(n_{ij}^{a}, n_{ij}^{b})\ count(f_{ij}^{a} \cup f_{ij}^{b})} \qquad \text{Equation 3}$$

For flexible molecules, the distance fingerprints can become saturated. The similarity can still be discerned by the distance count element of the key.

Some chemical groups may be treated as equivalent to other groups (e.g. hydroxyls and hydrogen bond donors). Such chemical equivalency can be supplied in an editable file (see chem_screen.tbl on page 171) and identified with the `chemical_screen_file` parameter.

UCSF LIBRARY

Macromolecular Docking

Though DOCK is typically used to process small molecules, it can be used to study the interactions of macromolecular ligands. The chief difference in protocol is that to use the match_receptor_sites procedure for the orientation search, then special ligand centers must be used to represent the ligand. This is signalled by setting the match_ligand_centers parameter. The ligand centers must reside in a file identified with the ligand_center_file parameter.

The ligand centers may be constructed with SPHGEN, using spheres to describe the positive image of the macromolecule. See Shoichet and Kuntz [26], for examples and discussion of macromolecular docking.

If multiple orientations are written to PDB formatted file, then the residue numbers are not disturbed. Normally, dock gives each orientation in the output PDB file a sequential residue number. However, if multiple substructures (residues) are present in the molecule input file, then this procedure is precluded.

References

1. Blaney, J.M. Ph.D. dissertation, University of California, San Francisco, 1982.
2. Connolly, M.L. Analytical molecular surface calculation. *J. Appl. Cryst.* **16**: 548-558, 1983.
3. Connolly, M.L. Solvent-accessible surfaces of proteins and nucleic acids. *Science.* **221**: 709-713, 1983.
4. DesJarlais, R.L., and Dixon, J.S. A shape and chemistry-based docking method and its use in the design of HIV-1 protease inhibitors. *J. Comput.-Aided Molec. Design* **8**(3): 231-242, 1994.
5. DesJarlais, R.L., Sheridan, R.P., Seibel, G.L., Dixon, J.S., Kuntz, I.D. and Venkataraghavan, R. Using shape complementarity as an initial screen in designing ligands for a receptor binding site of known three-dimensional structure. *J. Med. Chem.* **31**(4): 722-729, 1988.
6. Ewing, T.J.A, and Kuntz, I.D. Critical evaluation of search algorithms used in automated molecular docking. *J. Comput. Chem.* **15**: ???, 1997.
7. Ferro, D.R. and Hermans, J. A different best rigid-body molecular fit routine. *Acta Cryst.* **A33**: 345-347, 1977.
8. Fletcher, R. "Practical Methods of Optimization." New York: Interscience, 1960.
9. Gilson, M.K., Sharp, K.A. and Honig, B.H. *J. Comp. Chem.* **9**: 327, 1987.
10. Goodford, P.J. A computational procedure for determining energetically favorable binding sites on biologically important macromolecules. *J. Med. Chem.* **28**: 849-857, 1985.
11. Gschwend, D.A, and Kuntz, I.D. Orientational sampling and rigid-body minimization in molecular docking revisited — On-the-fly optimization and degeneracy removal. *J. Comput.-Aided Molec. Design*, **10**:123-132, 1996.
12. Kabsch, W. A solution for the best rotation to relate two sets of vectors. Acta Cryst. **A32**: 922-923, 1976.
13. Kabsch, W. A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Cryst.* **A34**: 827-828, 1978.
14. Klapper, I., Hagstrom, R., Fine, R., Sharp, K. and Honig, B. *Proteins.* **1**: 47-59, 1986.
15. Kuhl, F.S., Crippen, G.M., and Friesen, D.K. A Combinatorial Algorithm for Calculating Ligand Binding. *J. Comput. Chem.* **5**:24-34, 1984.
16. Kuntz, I.D., Blaney, J.M., Oatley, S.J., Langridge, R. and Ferrin, T.E. A geometric approach to macromolecule-ligand interactions. *J. Mol. Biol.* **161**: 269-288, 1982.
17. Kuntz, I.D. Structure-based strategies for drug design and discovery. *Science.* **257**: 1078-1082, 1992.
18. Kuntz, I.D., Meng, E.C. and Shoichet, B.K. Structure-based molecular design. *Acc. Chem. Res.* **27**(5): 117-123, 1994.
19. Leach, A.R., and Kuntz, I.D. Conformational analysis of flexible ligands in macromolecular receptor sites. *J. Comput. Chem.* **13**(6): 730-748, 1992.

20. Meng, E.C., Shoichet, B.K. and Kuntz, I.D. Automated docking with grid-based energy evaluation. *J. Comp. Chem.* **13**: 505-524, 1992.

21. Meng, E.C., Gschwend, D.A., Blaney, J.M. and Kuntz, I.D. Orientational sampling and rigid-body minimization in molecular docking. *Proteins.* **17**(3): 266-278, 1993.

22. Meng, E.C., Kuntz, I.D., Abraham, D.J. and Kellogg, G.E. Evaluating docked complexes with the HINT exponential function and empirical atomic hydrophobicities. *J. Comp-Aided Mol. Design.* **8**: 299-306, 1994.

23. Miller, M.D., Kearsley, S.K., Underwood, D.J. and Sheridan, R.P. FLOG - A system to select quasi-flexible ligands complementary to a receptor of known three-dimensional structure. J. Comput.-Aided Mol. Design

24. Nelder, J.A. and Mead, R. *Computer Journal* **7**: 308, (1965).

25. Richards, F.M. *Ann. Rev. Biophys. Bioeng.* **6**: 151-176, 1977.

26. Shoichet, B.K. and Kuntz, I.D. Protein docking and complementarity. *J. Mol. Biol.* **221**: 327-346, 1991.

27. Shoichet, B.K., Bodian, D.L. and Kuntz, I.D. Molecular docking using shape descriptors. *J. Comp. Chem.* **13**(3): 380-397, 1992.

28. Shoichet, B.K., Stroud, R.M., Santi, D.V., Kuntz, I.D. and Perry, K.M. Structure-based discovery of inhibitors of thymidylate synthase. *Science.* **259**: 1445-1450, 1993.

29. Shoichet, B.K. and Kuntz, I.D. Matching chemistry and shape in molecular docking. *Protein Eng.* **6**(7): 723-732, 1993.

30. Weiner, S.J., Kollman, P.A., Case, D.A., Singh, U.C., Ghio, C., Alagona, G., Profeta, S., Jr. and Weiner, P. A new force field for molecular mechanical simulation of nucleic acids and proteins. *J. Am. Chem. Soc.* **106**: 765-784, 1984.

31. Weiner, S.J., Kollman, P.A., Nguyen, D.T. and Case, D.A. An all atom force field for simulations of proteins and nucleic acids. *J. Comp. Chem.* **7**: 230-252, 1986.

Appendix 2: Reference Manual

Copyright © 1997
Regents of the University of California
All Rights Reserved

UCSF LIBRARY

DOCK

Overview

Version 1.0/1.1

Robert Sheridan, Renee DesJarlais, Irwin Kuntz

The program DOCK is an automatic procedure for docking a molecule into a receptor site. The receptor site is characterized by centers, which may come from SPHGEN or any other source. The molecule being docked is characterized ligand centers, which may be its non-hydrogen atoms or volume-filling spheres calculated in SPHGEN. The ligand centers and receptor centers are matched based on comparison of ligand-center/ligand-center and receptor-center/receptor-center distances. Sets of ligand centers match sets of receptor centers if all the internal distances match, within a value of `distance_tolerance`. Ligand-receptor pairs are added to the set until at least `nodes_minimum` pairs have been found. At least three pairs must be found to uniquely determine a rotation/translation matrix that will orient the ligand in the receptor site. A least-squares fitting procedure is used (Ferro and Hermans, 1977). Once an orientation has been found, it is evaluated by any of several scoring functions. DOCK may be used to explore the binding modes of an individual molecule, or be used to screen a database of molecules to identify potential ligands.

Version 2.0

Brian Shoichet, Dale Bodian, Irwin Kuntz

DOCK version 2.0 was written to give the user greater control over the thoroughness of the matching procedure, and thus over the number of orientations found and the CPU time required (Shoichet, Bodian, and Kuntz, 1992). In addition, certain algorithmic shortcomings of earlier versions were overcome. Versions 2.0 and higher are particularly useful for macromolecular docking (Shoichet and Kuntz, 1991) and applications which demand detailed exploration of ligand binding modes. In these cases, users are encouraged to run CLUSTER in conjunction with SPHGEN and DOCK.

To allow for greater control over searches of orientation space, the ligand and receptor centers are preorganized according to their internal distances. Starting with any given center, all the other centers are presorted into "bins" based on their distance to the first center. All centers are tried in turn as "first" positions, and all the points in a bin which has been chosen for matching are tried sequentially. Ligand and receptor bins are chosen for matching when they have the same distance limits from their respective "first" points. The number of centers in each bin determines how many sets of points in the receptor and the ligand will ultimately be compared. In general, the wider the bins, the greater the number of orientations generated. Thus, the thoroughness of the search is under user control.

Version 3.0

Elaine Meng, Brian Shoichet, Irwin Kuntz

Version 3.0 retained the matching features of version 2.0, and introduced options for scoring (Meng, Shoichet, and Kuntz, 1992). Besides the simple contact scores mentioned above, one can also obtain molecular mechanics interaction energies using grid files calculated by CHEMGRID (which is now superseded by GRID in version 4.0). More information about the ligand and receptor molecules is required to perform these higher-level kinds of scoring. Point charges on the receptor and ligand atoms are needed for electrostatic scoring, and atom-type information is needed for the van der Waals portion of the force field score. Input formats (some of them

new in version 3.5) are discussed in various parts of the documentation; one example of a "complete format" (including point charges and atom type information) is SYBYL ASCII (MOL2) format (Tripos Associates, Inc., St. Louis, MO 63117). Parameterization of the receptor is discussed in the documentation for CHEMGRID. In DOCK, ligand parameters are read in along with the coordinates; input formats are described below. Currently, the options are: contact scoring only, contact scoring plus Delphi electrostatic scoring, and contact scoring plus force field scoring. Atom-type information and point charges are not required for contact scoring only.

Version 3.5

Mike Connolly, Daniel Gschwend, Andy Good, Connie Oshiro, Irwin Kuntz

Version 3.5 added several features: score optimization, degeneracy checking, chemical matching and critical clustering.

Version 4.0

Todd Ewing, Irwin Kuntz

Version 4.0 was a major rewrite and update of DOCK. A new matching engine was developed which is more robust, efficient, and easier to use. Orientational sampling can now be controlled directly by specifying the number of desired orientations. Additional features include chemical scoring, chemical screening, and ligand flexibility.

UCSF LIBRARY

Command-line Arguments

Table 15. DOCK command-line arguments

flag	optional argument	behavior
`-i`	`input_file`	INPUT FILE — Parameters are extracted from `input_file`, or `dock.in` if not specified.
`-o`	`output_file`	OUTPUT FILE — Results are written to `output_file`, or `dock.out` if not specified. If a `-o` flag is present then DOCK runs in batch mode, otherwise it runs interactively (see below).
`-s`		STANDARD INPUT — Parameters are entered interactively without the construction of an input file. This option is generally not recommended.
`-r`		RESTART — Run is initialized with information from a restart file. This option is used to restart a `rank_ligands` docking run that was terminated prematurely.
`-p`		PROFILE — Time spent in docking routines and minimizer statistics are profiled. This option helps the user to identify the bottleneck of a particular calculation and to choose optimal minimizer parameters. See Performance on page 139.
`-t`		TERSE — Reduced output level. This option helps reduce the size of the output file when docking a large number of molecules.
`-v`		VERBOSE — Increased output level. This option allows additional data to be included in the output. Recommended for single molecule runs.

DOCK may be executed in either interactive or batch mode, depending on whether output is written to a file. In interactive mode, the user is requested only for parameters relevant to the particular run and default values are provided. This mode is recommended for the initial construction of the input file and for short calculations. In batch mode, input parameters are read in from the input file and all output is written to the output file. This mode is recommended for long calculations once an input file has been generated interactively.

Interactive mode

- `dock -i dock.in`

  When launched this way, DOCK will extract all relevant parameters from `dock.in` (or any file supplied by the user). If additional parameters are needed (or if the `dock.in` file is non-existent or empty), DOCK will request them one at a time from the user. Reasonable default values are presented. Any parameters supplied by the user will be automatically appended to the `dock.in` file. If the user would like to change any previously entered values, the user can edit in the `dock.in` file using a text editor.

- `dock -i`

  DOCK will behave as above, but will assume the input file to be `dock.in`.

UCSF LIBRARY

- `dock -s`

  DOCK will run interactively, but will not check any input file for parameters and will not append any entered parameters to a file.

Batch mode

- `dock -i dock.in -o dock.out`

  DOCK will run in batch mode, extracting all relevant parameters from `dock.in` (or any file supplied by the user) and will write out all output to `dock.out` (or any file supplied by the user). If any parameters are missing or incorrect, then execution will halt and an appropriate error message will be reported in `dock.out`.

- `dock -i -o`

  DOCK will behave as above, but will assume the i/o files to be `dock.in` and `dock.out`.

- `dock`

  If a file called `INDOCK` is present in the current working directory, then DOCK will use it as an input file. Output will be written to a file called `OUTDOCK`. This mode is present for reverse compatibility with previous versions of DOCK.

Molecule File Conversion

- `dock -i old.mol2 -o new.pdb`

  If the input and output file have recognized coordinate file extensions (`*.mol2`, `*.pdb`, `*.xpdb`, `*.sph`, `*.ptr`), then DOCK will automatically construct its own parameter input file to perform a file conversion operation.

UCSF LIBRARY

Input Parameters

File Format

Input parameters may be supplied in a text file. There are few simple format rules for this file.

Table 16. Format of Input File

Rule	Consequence
Parameter names must be the first word on a line to be recognized.	Any character or word preceding a parameter effectively comments it out.
The word following a recognized parameter name is interpreted as the parameter value.	Comments can appear on the same line, after a parameter and value.
Blank lines are allowed.	Comments can appear on separate lines.
Empty files are allowed.	Missing parameters will be flagged (batch mode) or requested (interactive mode).
The order of parameters is irrelevant.	

Parameters

DOCK has a large number of input parameters because of its wide array of functionality. Parameters are read in hierarchically, with the most general parameters first. They are also ordered according to broad classes of functionality in order to make the process of selection more intuitive. For instance all scoring-related parameters are requested together. Since only the relevant parameters are requested based on preceding selections, it is recommended that initial parameter selection be done interactively to generate a sensible input file as painlessly as possible.

The following list of parameters are grouped according to the order they are requested at run time. Again, many parameters are listed here that will not be requested during your particular run.

The default values listed here may be different from those provided during a particular run, because DOCK tries to recommend sensible parameter values based on preceding selections.

Table 17. DOCK input Parameters

17A: General

Parameters in this category determine the general behavior of the dock run. Each option can be used in isolation and many in combination to allow DOCK to perform a diverse repertoire of tasks. If none of these flags are selected, then DOCK simply reads in a single molecule and writes it out, which is useful for file format conversion.

Parameter	Type	Default	Description
chemical_screen	boolean	no	Flag to perform chemical screening.
flexible_ligand	boolean	no	Flag to allow ligand flexibility.
orient_ligand	boolean	no	Flag to perform an orientation search.
multiple_ligands	boolean	no	Flag to consider multiple ligands.
score_ligand	boolean	no	Flag to perform scoring of orientations or conformations.
minimize_ligand	boolean	no	Flag to perform local score minimization of orientations or conformations.
parallel_jobs	boolean	no	Flag to perform docking as parallel jobs orchestrated by a server dock job.
random_seed	integer	0	Integer to seed the random number generator.

17B: Chemical Screening

Chemical screening allows for rapid filtering of a database based on chemical and 3D distance descriptors. Before any screening can be done, the database must be keyed using the `construct_screen` parameter at which time ligand flexibility should be considered. Then `screen_ligands` can be activated to do pharmacophore screening or similarity screening. Make sure to use the same distance bin dimensions. Folding the keys will be done automatically for similarity screening. Folding can also be done manually to process an unfolded database and convert it into a folded one. See Chemical Screening on page 108 for more discussion.

Parameter	Type	Default	Description
construct_screen	boolean	no	Flag to construct distance chemical keys for each input molecule.
screen_ligands	boolean	no	Flag to screen input molecules based on distance chemical keys.
pharmacophore_screen	boolean	no	Flag to screen based on whether molecule keys include pharmacophore pattern. With unfolded keys, matching is performed.
similarity_screen	boolean	no	Flag to screen based on similarity of molecule keys to target keys.
dissimilarity_maximum	float	0.25	Maximum dissimilarity with target to write out molecule.
distance_begin	float	2	Smallest distance of interest in fingerprint.
distance_end	float	17	Largest distance of interest in fingerprint.
distance_interval	float	0.5	Distance resolution of fingerprint. The total number of distance keys cannot exceed 30, since a 32-bit key is currently used with the terminal bits being reserved for out-of-range values.

17C: Ligand Flexibility (Part 1 of 3)

Ligand flexibility parameters can be used in various combinations to relax an input conformation, to perform a conformational search of a molecule, or to perform a completely flexible docking of a molecule. See Conformation Search on page 99 for more discussion.

Parameter	Type	Default	Description
anchor_search	boolean	no	Flag to organize rigid segments into concentric layers around an anchor. Only the anchor is included in initial scoring/orienting. The outer segments should be regrown with a peripheral_search. • yes = Anchor-First Search is performed. • no = Simultaneous Search of all torsions is performed prior to other processing.
multiple_anchors	boolean	no	Flag to select several anchors for a molecule. • yes = All rigid segments meeting anchor_size criteria will be tried independently as anchors. • no = The largest segment is used.
anchor_size	integer	10	Minimum number of heavy atoms for an anchor if multiple_anchors requested. If no segment satisfies this value, then the largest segment is used.
peripheral_search	boolean	no	Flag to identify the conformations of outer segments after an anchor_search. To eliminate the combinatorial problem, a single segment is added at a time. • yes = Molecule is regrown from the anchor. • no = Only the anchor is processed, which is useful for a preliminary analysis of flexible docking.
peripheral_seeds	integer	25	Number of conformations/ orientations to seed the next level of the peripheral_search.
torsion_drive	boolean	no	Flag to perform systematic search of torsions.

17C: Ligand Flexibility (Part 2 of 3)

(Repeat) Ligand flexibility parameters can be used in various combinations to relax an input conformation, to perform a conformational search of a molecule, or to perform a completely flexible docking of a molecule. See Conformation Search on page 99 for more discussion.

Parameter	Type	Default	Description
clash_overlap	real	0.5	Amount of atom VDW overlap allowed during a torsion_drive. If two ligand atoms approach closer than this fraction of sum of the VDW radii, then the conformation is discarded. Similar to bump_overlap in GRID. • 0 = Complete overlap allowed. • 1 = No overlap allowed.
maximum_conformations	integer	1000	Maximum conformations to sample per molecule for a torsion_drive without an anchor_search. • If the total possible conformations is <= maximum_conformations then an exhaustive systematic search is performed. • Otherwise, a random search of maximum_conformations is performed, with torsion values selected randomly from the flex_drive_file.
torsion_minimize	boolean	no	Flag to perform torsion relaxation based on intramolecular_score and/or intermolecular_score.
reminimize_layer_number	integer	2	Number of previous layers to minimize while minimizing the current segment. Only neighboring segments in inner layers (and their neighbors in outer layers) are minimized. This feature helps rescue conformations from dead ends during a peripheral_search.
minimize_anchor	boolean	no	This flag is present during a peripheral_search. It specifies whether the anchor is minimized during the initial anchor docking. In general this flag should be turned on.

17C: Ligand Flexibility (Part 3 of 3)

(Repeat) Ligand flexibility parameters can be used in various combinations to relax an input conformation, to perform a conformational search of a molecule, or to perform a completely flexible docking of a molecule. See Conformation Search on page 99 for more discussion.

Parameter	Type	Default	Description
reminimize_anchor	boolean	no	If a partially-built molecule is minimized during a peripheral_search, but no minimizable bonds are active, then a rigid-body minimization is performed. This flag will force such a rigid-body reminimization throughout the search. This method is a more expensive (but effective) way to rescue conformations.
reminimize_ligand	boolean	no	Flag to reminimize the molecule after a peripheral_search, letting all torsions and the anchor position relax simultaneously. This process resolves any accumulated strain built up during the search.
flexible_bond_maximum	integer	10	The maximum number of flexible bonds allowed in a molecule, when multiple_ligands are processed.

17D: Orientation Search

Orienting the ligand is fundamental to the docking process. Orienting is traditionally done by matching. Alternative orienting procedures are also available, particularly the random search, which can optionally be run without any site points. For typical uses, the traditional matching process is still recommended. Please refer to Orientation Search on page 94 for more discussion.

Parameter	Type	Default	Description
`match_receptor_sites`	`boolean`	`no`	Flag to perform traditional site point-directed matching.
`match_ligand_centers`	`boolean`	`no`	Flag to use ligand centers read in from a separate file for matching instead of ligand heavy atoms. Please refer to Macromolecular Docking on page 110 for more discussion.
`random_search`	`boolean`	`no`	Flag to randomly search ligand orientations. • With `match_receptor_sites`, all matches are constructed randomly rather than based on distance comparisons. • Otherwise, orientations are randomly constructed inside the smallest rectangular volume that encloses all points read in as site points. See Random Search on page 95 for more discussion.
`uniform_sampling`	`boolean`	`yes`	Flag to generate the same number of orientations for every ligand. This feature provides extremely easy control over orientation sampling. It is recommended for most dock runs, when match geometry is not critical.
`total_orientations`	`integer`	`500, or 100 if > 1 ligand`	With `uniform_sampling`, this sets number of orientations to generate for each molecule.
`write_orientations`	`boolean`	`no`	Flag to write out more than the best orientation per molecule.
`rank_orientations`	`boolean`	`no`	Flag to rank molecule orientations by score.
`rank_orientation_total`	`integer`	`100`	Number of ranked orientations to store.

17E: Matching

Matching is the traditional procedure driving the orientation search in DOCK. If `uniform_sampling` was not selected, then the amount of sampling is controlled by the node and distance parameters. Other constraints on sampling are available based on chemical labeling or critical clusters. See Matching on page 95 for further discussion.

Parameter	Type	Default	Description
`nodes_minimum`	`integer`	`3`	Smallest number of atom-site point interactions needed to construct an orientation.
`nodes_maximum`	`integer`	`10`	Largest number of atom-site point interactions considered to construct an orientation
`distance_tolerance`	`real`	`0.25`	Maximum difference between all intra-ligand and intra-receptor distances in a match. This is the chief sampling parameter for matching.
`distance_minimum`	`real`	`2.0`	Minimum intra-ligand or intra-receptor distance allowed in a match. This parameter biases matching toward longer distances which convey more information about ligand or site shape.
`check_degeneracy`	`boolean`	`no`	Flag to discard matches that are subsets of larger matches.
`reflect_ligand`	`boolean`	`no`	Flag to dock the mirror image of a ligand to rescue an improper match. Half of all matches with 4 or more nodes require reflection, otherwise they are discarded. Use of this parameter is not generally recommended.
`critical_points`	`boolean`	`no`	Flag to force matching to include members from a particular group (or groups) of site points in every match. This parameter is useful to focus docking around a few key residues in an active site.
`multiple_points`	`boolean`	`no`	Flag used in combination with `critical_points` to allow multiple points from the same critical cluster to appear in a match.
`chemical_match`	`boolean`	`no`	Flag to use chemical labeling to identify bad interactions within a match.

17F: Multiple Ligands

The parameters in this category control the processing of a database of ligands. See Database Processing on page 105 for more discussion.

Parameter	Type	Default	Description
ligands_maximum	integer	1000	The maximum number of ligands to read in from the input file. This INCLUDES skipped ligands.
initial_skip	integer	0	The initial number of ligands to skip. This is useful to position the reading stream to a particular point in the input file.
interval_skip	integer	0	The number of ligands to skip for every ligand processed. This is useful to perform a preliminary scan a database for timing purposes, or to coordinate the processing of a database over multiple machines.
heavy_atoms_minimum	integer	0	Minimum number of heavy atoms. In general, set this to at least three (or nodes_minimum) when an orientation search is performed.
heavy_atoms_maximum	integer	100	Maximum number of heavy atoms.
rank_ligands	boolean	no	Flag to rank best molecules. • yes = Only top scorers written. • no =For each molecule, the best orientation (or set of orientations with write_orientations) is written.
rank_ligand_total	integer	100	Number of ranked ligands.
restart_interval	integer	100	Number of ligands to process between each time restart information is saved.

UCSF LIBRARY

17G: Scoring (Part 1 of 4)

Each orientation is scored according to the options selected in this section. Several scoring functions exist and are treated independently of each other. To filter molecules based on more than one function simultaneously, you will need to rescore in a subsequent step. Most scoring is speeded up by precalculating a potential on a 3D grid, but continuum scoring is available.

Parameter	Type	Default	Description
intramolecular_score	boolean	no	Flag to compute score between rigid segments. This feature is available if flexible_ligand is selected.
intermolecular_score	boolean	no	Flag to compute score between ligand and receptor.
gridded_score	boolean	yes	Flag to use precomputed grids to evaluate the score, otherwise a continuous evaluation is made.
grid_version	real	4	Option to select grids computed by current version of GRID or by version 3.5 CHEMGRID.
grid_points	integer	1000000	If a version preceding 4 is selected, then this parameter specifies how many grid points are contained in the grids.
bump_filter	boolean	no	Flag to screen each orientation for clashes with receptor prior to scoring and minimizing.
bump_maximum	integer	0	Maximum number of allowed bumps.
contact_score	boolean	no	Flag to perform contact scoring.
contact_cutoff_distance	real	4	Interaction distance for contact scoring. Please refer to contact_cutoff_distance in GRID.

17G: Scoring (Part 2 of 4)

(Repeat) Each orientation is scored according to the options selected in this section. Several scoring **functions exist** and are treated independently of each other. To filter molecules based on more **than one function** simultaneously, you will need to rescore in a subsequent step. Most scoring is **speeded up** by precalculating a potential on a 3D grid, but continuum scoring is

Parameter	Type	Default	Description
contact_clash_overlap	real	0.75	Amount of VDW overlap allowed. If two atoms approach closer than this fraction of the sum of their VDW radii, then the contact score is penalized. • 0 = Complete overlap allowed. • 1 = No overlap allowed.
contact_clash_penalty	real	50	Amount that contact score is penalized for each clash.
chemical_score	boolean	no	Flag to perform chemical scoring. This feature is included for experimental purposes only. Parameterization is left to the user. Use at your own risk.
energy_score	boolean	no	Flag to perform energy scoring.
energy_cutoff_distance	real	10	Maximum distance between two atoms for their contribution to the energy score to be computed.
distance_dielectric	boolean	yes	Flag to make the dielectric depend linearly on the distance.
dielectric_factor	real	4	Coefficient of the dielectric. See Equation 4 on page 140 for context.
attractive_exponent	integer	6	Exponent of attractive Lennard-Jones term for VDW potential.
repulsive_exponent	integer	12	Exponent of repulsive Lennard-Jones term for VDW potential.

UCSF LIBRARY

17G: Scoring (Part 3 of 4)

(Repeat) Each orientation is scored according to the options selected in this section. Several scoring functions exist and are treated independently of each other. To filter molecules based on more than one function simultaneously, you will need to rescore in a subsequent step. Most scoring is speeded up by precalculating a potential on a 3D grid, but continuum scoring is

Parameter	Type	Default	Description
atom_model	string	u	Flag for how to model non-polar hydrogens. • u = United atom model. Hydrogens attached to carbons are assigned a zero VDW well-depth and the partial charge is transferred to the carbon. • a = All atom model. Hydrogens attached to carbons have regular VDW well-depth and partial charge is not modified.
vdw_scale	real	1	Scaling factor of vdw component of energy score.
electrostatic_scale	real	1	Scaling factor of electrostatic component of energy score.
rmsd_score	boolean	no	Flag to perform rmsd scoring, which is the rmsd of the molecules in the ligand_atom_file with respect to the molecule in the receptor_atom_file. Both molecules must have identical atoms.

17G: Scoring (Part 4 of 4)

(Repeat) Each orientation is scored according to the options selected in this section. Several scoring functions exist and are treated independently of each other. To filter molecules based on more than one function simultaneously, you will need to rescore in a subsequent step. Most scoring is speeded up by precalculating a potential on a 3D grid, but continuum scoring is

Parameter	Type	Default	Description
contact_maximum chemical_maximum energy_maximum rmsd_maximum	real	0	If orientations or ligands to be written, but not ranked, then they must pass this score cutoff to be written to file.
contact_size_penalty chemical_size_penalty energy_size_penalty	real	0	If ligands to be ranked, then they may be penalized by this value for each heavy atom. This helps correct for the uncomplexed score and reduce the size bias.
rmsd_override	real	0	If orientations to be written, but not ranked, then orientations with an RMSD (with respect to the input orientation) less than this value are written to file regardless of score.

UCSF LIBRARY

17H: Minimization (Part 1 of 2)

Minimization allows on-the-fly adjustment of a molecule's orientation and/or conformation to improve its score. Though this calculation is CPU intensive, it improves the efficiency of the orientation or conformation search. The simplex algorithm uses random displacements to seed the search. Consequently, minimization results vary depending on the random seed selected by the user and the order of input molecules and site points. If high-quality results are required, then repeat the run several times with different random seeds.

Parameter	Type	Default	Description
contact_minimize chemical_minimize energy_minimize rmsd_minimize	boolean	n	Flags to perform minimization with respect to each scoring function.
initial_translation	real	1	The maximum initial step size (in Angstroms) of the simplex in each cartesian dimension. The actual step size is a random value between zero and this value.
initial_rotation	real	0.1	The maximum initial step size (unitless) for each axis of rotation. The quaternion representation is used. A value of one corresponds to a 180 degree rotation.
initial_torsion	real	10	The maximum initial step size (in degrees) for each torsion when `flexible_ligand` set. When several layers are minimized together, the outermost layer gets a full step and the step for each inner layer is divided by 2, 3, etc.
maximum_iterations	integer	100	Maximum number of simplex iterations for each cycle of minimization.
contact_convergence chemical_convergence energy_convergence rmsd_convergence	real	0.5	Convergence criteria with respect to each scoring function. At any iteration, if all vertices of the simplex have a score within this value of the best score, then minimization terminates.

17H: Minimization (Part 2 of 2)

(Repeat) Minimization allows on-the-fly adjustment of a molecule's orientation and/or conformation to improve its score. Though this calculation is CPU intensive, it improves the efficiency of the orientation or conformation search. The simplex algorithm uses random displacements to seed the search. Consequently, minimization results vary depending on the random seed selected by the user and the order of input molecules and site points. If high-quality results are required, then repeat the run several times with different random seeds.

Parameter	Type	Default	Description
maximum_cycles	integer	1	Maximum number of minimization cycles. After the first cycle, the initial step sizes are divided by 2, 3, etc.
cycle_convergence	real	1	The distance a minimization cycle must travel to trigger another cycle. The vector distance of the final simplex vertices is used, which is normalized with respect to the initial step size.
contact_termination chemical_termination energy_termination rmsd_termination	real	1	If the score is greater than this value after a cycle of minimization, then no more cycles are attempted. This parameter is useful to avoid prolonged minimization of unrecoverable orientations or conformations.

17I: Parallel Jobs (Part 1 of 2)

The parallel job parameters provide a convenient way to process a large database of molecules and distribute the workload over many computers. One dock job must be configured as a server job. Any number of other jobs running on separate machines are configured as client jobs. The server job reads the entire database and parses molecules out to the client jobs for processing. It is recommended that the server job be executed on the computer which stores the database. See Database Processing on page 105 for more discussion.

Parameter	Type	Default	Description
parallel_server	boolean	no	Flag to identify this job as the server job.
server_name	string	server	The name used by the server job to communicate with client jobs.

UCSF LIBRARY

17I: Parallel Jobs (Part 2 of 2)

(Repeat) The parallel job parameters provide a convenient way to process a large database of molecules and distribute the workload over many computers. One dock job must be configured as a server job. Any number of other jobs running on separate machines are configured as client jobs. The server job reads the entire database and parses molecules out to the client jobs for processing. It is recommended that the server job be executed on the computer which stores the database. See Database Processing on page 105 for more discussion.

Parameter	Type	Default	Description
`client_total`	`integer`	`5`	The number of client jobs at the disposal of this server job.

`client_name_1`
`client_name_2`
`client_name_3 ...`
	`string`	`clientN`	If `parallel_server` set, then this list of names of client jobs is requested.

`client_name`	`string`	`client`	If `parallel_server` not set, then the name of this particular client job is requested. To be recognized by the server, this name must be included in the `client_name_N` list supplied to the server job.

17J: Input (Part 1 of 2)

The user must supply several kinds of input files.

- **Coordinate** Contains molecules/site points (*.mol2, *.pdb, *.xpdb, *.ptr, *.sph).
- **Grid** Contains precalculated score potentials.
- **Parameter** Contains VDW, chemical and flexibility parameters (*.defn, *.tbl).
- **Control** Empty. Used to interact with a running job (*.quit, *.dump).

Parameter	Default	Description
`ligand_atom_file`	`ligand.mol2`	File containing ligand atom coordinates.
`ligand_center_file`	`ligand_center.sph`	File containing ligand site points (see Macromolecular Docking on page 110).
`receptor_site_file`	`receptor_site.sph`	File containing receptor site points.
`score_grid_prefix`	`score_grid`	Prefix for files containing precalculated score grids.
`receptor_atom_file`	`receptor.mol2`	File containing receptor atom coordinates. Used for continuum (no grids) scoring when `gridded_score` not set.
`vdw_definition_file`	`$PATH/vdw.defn`	File containing VDW labels and parameters.

17J: Input (Part 2 of 2)

(Repeat) The user must supply several kinds of input files.

- **Coordinate** Contains molecules/site points (*.mol2, *.pdb, *.xpdb, *.ptr, *.sph).
- **Grid** Contains precalculated score potentials.
- **Parameter** Contains VDW, chemical and flexibility parameters (*.defn, *.tbl).
- **Control** Empty. Used to interact with a running job (*.quit, *.dump).

Parameter	Default	Description
chemical_definition_file	$PATH/chem.defn	File containing chemical labels and definitions.
chemical_match_file	$PATH/ chem_match.tbl	File containing chemical interaction table for use when chemical_match set.
chemical_score_file	$PATH/ chem_score.tbl	File containing chemical interaction table for use when chemical_score set.
chemical_screen_file	$PATH/ chem_screen.tbl	File containing chemical interaction table for use when similarity_screen set.
flex_definition_file	$PATH/flex.defn	File containing flexible bond labels and definitions for use when flexible_ligand set.
flex_drive_file	$PATH/ flex_drive.tbl	File containing torsion parameters for a torsion_drive search.
quit_file	*.quit	When multiple_ligands is set, this file may be created by the user at any time to signal the dock job to terminate execution. If rank_ligands is set, the best molecules are written to file and an up-to-date restart file is generated.
dump_file	*.dump	If rank_ligands is set, this file may be created by the user at any time to signal the dock job to write the best molecules to file and resume execution.

17K: Output

DOCK writes up to three kinds of output files.

- **Coordinate** Contains docked molecules (*.mol2, *.pdb, *.xpdb, *.ptr, *.sph).
- **Info** Contains current ligand rankings.
- **Restart** Contains restart information.

Parameter	Default	Description

`ligand_out_file`
`ligand_contact_file`
`ligand_chemical_file`
`ligand_energy_file`
`ligand_rmsd_file`
	`ligand_out.mol2`
	`ligand_cnt.mol2`
	`ligand_chm.mol2`
	`ligand_nrg.mol2`
	`ligand_rms.mol2`
		Files containing docked, minimized, rescored, or reformatted molecules for each type of scoring.

`info_file`	`*.info`	File containing summary information about the current `rank_ligands` list.
`restart_file`	`*.rst`	File containing detailed information about current `rank_ligands` list which is sufficient for restarting if the current job is prematurely terminated. See Command-line Arguments on page 117 for restart instructions.

UCSF LIBRARY

Output

DOCK reports several kinds of information as output which is either written to screen in interactive mode or written to file in batch mode. Each section of the output is discussed below.

Heading

```
      UUUUUUUUU     CCCCCCC     SSSSSSS    FF/    FFF/
      UU/    UU/  CC/     CC/  SS/     SS/  FF/ FFF/
     UU/    UU/  CC/     CC/  SS/          FFFFF/
    UU/    UU/  CC/     CC/  SS/          FF/ FF\
   UU/    UU/  CC/     CC/  SS/     SS/  FF/    FF\
 UUUUUUUUU/    CCCCCCC/    SSSSSSS/    FF/      FF\


University of California at San Francisco, DOCK 4.0

__________________Job_Information_________________
launch_time                           Wed Mar 19 12:05:01 1997
host_name                             mycomputer
memory_limit                          126062592
working_directory                     /usr/people/user
user_name                             user
```

The heading lists general information about the current job. The `launch_time`, `host_name` and `working_directory` fields are useful to help the user archive the run conditions. The `memory_limit` field refers to the amount of RAM in bytes available to the job. This is can at most be equal to the physical memory of the computer, but may be less if limits have been imposed on the users shell. If the job requires more RAM than is available, then execution is stopped and an error message is reported.

Parameters

```
________________General_Parameters________________
chemical_screen                 no
flexible_ligand                 no
orient_ligand                   no
multiple_ligands                yes
score_ligand                    no
parallel_jobs                   no

____________Multiple_Ligand_Parameters____________
ligands_maximum                 <infinity>
initial_skip                    0
interval_skip                   0

____________________File_Input____________________
ligand_atom_file                old.mol2

____________________File_Output___________________
ligand_none_file                new.pdb
```

All input parameters are echoed into the output. In fact, the output file may be used as an input file for another run. This is a useful technique to clean up an old, cluttered input file and make a tidy, well-organized input file.

This particular parameter list is nearly the shortest possible since most general parameters have been turned off. It is constructed automatically when DOCK is used to convert a molecule file format (see Molecule File Conversion on page 118).

If a parameter is not needed it is not reported in the output. The user can override this feature and force all parameters to be written out by using the `-v` flag (see Command-line Arguments on page 117).

Results

```
_________________________Docking_Results_________________________
Name            : ligand1
Description     : ****
Orientations tried                                :        616
Orientations scored                               :        500

Best energy score                                 :     -59.02
  Intramolecular energy score                     :      12.18
  Intermolecular energy score                     :     -71.20
RMSD of best energy scorer (A)                    :       2.39

Elapsed cpu time (sec)                            :     492.87
```

The docking results of each molecule are output as above. In terse mode (`-t` flag) all results are printed on a single line.

This particular output is from the flexible docking of a molecule.

Performance

```
_______________________Docking_Performance_______________________

Procedure timings                                 time (s)  percent
Read                                                  0.11        0
Screen                                                0.00        0
Orientation Search                                    0.48        0
Orientation Score                                    49.90       10
Conformation Anchor                                   0.05        0
Conformation Peripheral                             442.38       90
Other                                                 0.02        0
Total                                               492.94      100

Minimizer usage                          minimum   average  maximum
Calls per molecule                          4200      4200     4200
Score improvement per call                     0   2.7e+10  8.7e+13
Vertices per call                              1         3        6
Cycles per call                                1         2        5
Iterations per cycle                           1        15      100
```

This section of the output is generated if the user used the `-p` command-line flag (see Command-line Arguments on page 117). It is intended to help the user to understand the performance bottlenecks of a particular run and to make informed parameter choices. Since minimization can consume significant portions of cpu time, it is profiled in more detail.

This particular output is from a flexible docking run in which a majority of the cpu time is spent during the peripheral conformation search.

GRID

Author: Elaine Meng

Overview

GRID creates the grid files necessary for force field scoring (Meng, Shoichet, and Kuntz, 1992). Force field scores are approximate molecular mechanics interaction energies, consisting of van der Waals and electrostatic components:

$$E = \sum_{i=1}^{lig} \sum_{j=1}^{rec} \left[\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} + 332 \frac{q_i q_j}{D r_{ij}} \right] \qquad \text{Equation 4}$$

where each term is a double sum over ligand atoms i and receptor atoms j, A_{ij} and B_{ij} are van der Waals repulsion and attraction parameters, r_{ij} is the distance between atoms i and j, q_i and q_j are the point charges on atoms i and j, D is the dielectric function, and 332 is a factor that converts the electrostatic energy into kilocalories per mole.

An efficient grid calculation requires the use of a geometric mean approximation for the van der Waals parameters:

$$A_{ij} = \sqrt{A_{ii}}\sqrt{A_{jj}} \quad \text{and} \quad B_{ij} = \sqrt{B_{ii}}\sqrt{B_{jj}} \qquad \text{Equation 5}$$

where the single-atom-type parameters are calculated from van der Waals radius, R, and well depth, ε, according to:

$$A = \varepsilon[2R]^{12} \quad \text{and} \quad B = 2\varepsilon[2R]^{6} \qquad \text{Equation 6}$$

Using this approximation, Equation 4 can be rewritten:

$$E = \sum_{i=1}^{lig} \left[\sqrt{A_{ii}} \sum_{j=1}^{rec} \frac{\sqrt{A_{jj}}}{r_{ij}^{12}} - \sqrt{B_{ii}} \sum_{j=1}^{rec} \frac{\sqrt{B_{jj}}}{r_{ij}^{6}} + 332 q_i \sum_{j=1}^{rec} \frac{q_j}{D r_{ij}} \right] \qquad \text{Equation 7}$$

Three values are stored for every grid point k, each a sum over receptor atoms that are within a user-defined cutoff distance of the point:

$$A_{rec} = \sum_{j=1}^{rec} \frac{\sqrt{A_{jj}}}{r_{ij}^{12}}, \quad B_{rec} = \sum_{j=1}^{rec} \frac{\sqrt{B_{jj}}}{r_{ij}^{6}}, \text{ and } Q_{rec} = 332 \sum_{j=1}^{rec} \frac{q_j}{D r_{ij}} \qquad \text{Equation 8}$$

These values, with interpolation, are multiplied by the appropriate ligand values to give the interaction energy. CHEMGRID calculates the grid values and stores them in files. The values will be read in during a DOCK run and used for force field scoring. Substituting Equation 8 into Equation 7, the interaction energy is:

$$E = \sum_{i=1}^{lig} [\sqrt{A_{ii}} A_{rec} - \sqrt{B_{ii}} B_{rec} + q_i Q_{rec}] \qquad \text{Equation 9}$$

Atoms that fall outside the grid, if any, are given interaction energies of zero.

The user determines the location and dimensions of the grid box using the program SHOWBOX. It is not necessary for the whole receptor to be enclosed; only the regions where ligand atoms may be placed need to be included. The box merely delimits the space where grid points are located, and does not cause receptor atoms to be excluded from the calculation. Besides a direct specification of coordinates, there is an option to center the grid at a sphere cluster center of mass. Any combination of spacing and *x, y,* and *z* extents may be used.

Command-line Arguments

Table 18. GRID command-line argument summary

flag	optional argument	behavior
`-i`	`input_file`	INPUT FILE — Input parameters extracted from `input_file`, or `grid.in` if not specified
`-o`	`output_file`	OUTPUT FILE — Output written to `output_file`, or `grid.out` if not specified
`-s`		STANDARD INPUT — Input parameters entered interactively
`-t`		TERSE — Reduced output level
`-v`		VERBOSE — Increased output level

GRID may be executed in either interactive or batch mode, depending on whether output is written to a file. In interactive mode, the user is requested only for parameters relevant to the particular run and default values are provided. This mode is recommended for the initial construction of the input file. In batch mode, input parameters are read in from the input file and all output is written to the output file. This mode is recommended once an input file has been generated interactively.

Interactive mode

- `grid -i grid.in`

  When launched this way, GRID will extract all relevant parameters from `grid.in` (or any file supplied by the user). If additional parameters are needed (or if the `grid.in` file is non-existent or empty), GRID will request them one at a time from the user. Reasonable default values are presented. Any parameters supplied by the user will be automatically appended to the `grid.in` file. If the user would like to change any previously entered values, the user can edit in the `grid.in` file using a text editor.

- `grid -i`

  GRID will behave as above, but will assume the input file to be `grid.in`.

- `grid -s`

  GRID will run interactively, but will not check any input file for parameters and will not append any entered parameters to a file.

Batch mode

- `grid -i grid.in -o grid.out`

UCSF LIBRARY

GRID will run in batch mode, extracting all relevant parameters from `grid.in` (or any file supplied by the user) and will write out all output to `grid.out` (or any file supplied by the user). If any parameters are missing or incorrect, then execution will halt and an appropriate error message will be reported in `grid.out`.

- `grid -i -o`

  GRID will behave as above, but will assume the i/o files to be `grid.in` and `grid.out`.

Input Parameters

File Format

See Table 16. on page 119 for file format specifications.

Parameters

Like DOCK, GRID should be executed in interactive mode to construct an input file since not all parameters need to be specified for most runs. After all parameters have been entered, use CTRL-C to kill the process and resubmit it in batch mode using the same input file.

Table 19. GRID input parameters

19A: General Parameters

These parameters control the overall behavior of GRID execution.

Parameter	Type	Default	Description
`compute_grids`	`boolean`	`no`	Flag to compute scoring grids.
`grid_spacing`	`real`	`0.3`	The distance between grid points along each axis.
`output_molecule`	`boolean`	`no`	Flag to write out the coordinates of the receptor into a new, cleaned-up file. Atoms are resorted to put all residue atoms together. Terminal SYBYL capping groups are merged with the terminal residues.

19B: Scoring Parameters (Part 1 of 2)

These parameters determine which grids are made and how they will be computed.

Parameter	Type	Default	Description
contact_score	boolean	no	Flag to construct contact grid.
contact_cutoff_distance	real	4	Maximum distance between heavy atoms for the interaction to be counted as a contact.
chemical_score	boolean	no	Flag to construct chemical grid.
energy_score	boolean	no	Flag to perform energy scoring.
energy_cutoff_distance	real	10	Maximum distance between two atoms for their contribution to the energy score to be computed.
atom_model	string	u	Flag for how to model of non-polar hydrogens. • u = United atom model. Hydrogens attached to carbons are assigned a zero VDW well-depth and the partial charge is transferred to the carbon. • a = All atom model. Hydrogens attached to carbons have regular VDW well-depth and partial charge is not modified.
attractive_exponent	integer	6	Exponent of attractive Lennard-Jones term for VDW potential. See Equation 4 on page 140 for context.
repulsive_exponent	integer	12	Exponent of repulsive Lennard-Jones term for VDW potential.
distance_dielectric	boolean	yes	Flag to make the dielectric depend linearly on the distance.

19B: Scoring Parameters (Part 2 of 2)

(Repeat) These parameters determine which grids are made and how they will be computed.

Parameter	Type	Default	Description
dielectric_factor	real	4	Coefficient of the dielectric.
bump_filter	boolean	no	Flag to screen each orientation for clashes with receptor prior to scoring and minimizing.
bump_overlap	real	0.75	Amount of VDW overlap allowed. If the probe atom and the receptor heavy atom approach closer than this fraction of the sum of their VDW radii, then the position is flagged as a bump. • 0 = Complete overlap allowed. • 1 = No overlap allowed.

19C: File Input

Input file parameters.

Parameter	Default	Description
receptor_file	receptor.mol2	Receptor coordinate file. Partial charges and atom types need to be present.
box_file	site_box.pdb	File containing SHOWBOX output file which specifies boundaries of grid.
vdw_definition_file	$PATH/vdw.defn	VDW parameter file.
chemical_definition_file	$PATH/chem.defn	Chemical label definition file for use when chemical_score set.

19D: File Output

Output file parameters.		
Parameter	**Default**	**Description**
score_grid_prefix	grid	Core file name of grids (file extension will be appended automatically).
receptor_out_file	receptor_out.mol2	File for cleaned-up receptor when output_molecule set.

Output

The output of GRID contains several types of information. Like DOCK, it outputs general information about the current job and echoes the parameters selected from the input file. In addition, it reports information about the receptor and the grids.

Receptor information

```
Reading in coordinates of receptor.
Merging AMN 163 cap residue with THR1 1 residue.
Merging CXL 164 cap residue with ALA162 162 residue.
CHARGED RESIDUE THR1                    :     1.000
CHARGED RESIDUE ARG9                    :     1.000
CHARGED RESIDUE LYS51                   :     1.000
CHARGED RESIDUE ARG52                   :     1.000
CHARGED RESIDUE GLU56                   :    -1.000
CHARGED RESIDUE ARG57                   :     1.000
CHARGED RESIDUE HIP77                   :     1.000
CHARGED RESIDUE ASP78                   :    -1.000
CHARGED RESIDUE LYS127                  :     1.000
CHARGED RESIDUE ASP146                  :    -1.000
CHARGED RESIDUE HIP153                  :     1.000
CHARGED RESIDUE GLU156                  :    -1.000
CHARGED RESIDUE LYS161                  :     1.000
CHARGED RESIDUE ALA162                  :    -1.000

Total charge on UNNAMED                 :     4.000
```

This portion of the output lists any merged cap residues. The cap residues are introduced by SYBYL. Charged residues are also listed. If any residues have a non-integer charge, then either the charges were not properly loaded into the receptor input file, or some atoms are missing from the input file. These problems should be resolved before continuing with the grid calculation.

To display more information about parameters that GRID assigns to the receptor, use the -v option (see Command-line Arguments on page 141).

Grid Information

```
Reading in grid box information.
Box center of mass                     :   -1.315   36.145   21.153
Box dimensions                         :   27.525   26.519   26.686
Number of grid points per side [x y z] :       93       90       90
Total number of grid points            :   753300

Generating scoring grids.
Percent of protein atoms processed     :        0
Percent of protein atoms processed     :       10
Percent of protein atoms processed     :       20
...etc...
```

Information about the grid geometry is reported. The total number of grid points can be adjusted by changing the input box dimensions or changing the grid spacing. Generally a value under 1 million is appropriate, but this depends on system resources.

The progress of the calculation is reported as the percent of protein atoms processed. This calculation usually takes up to 30 minutes depending on the grid geometry and the receptor size.

SPHGEN

Author: Irwin D. Kuntz
Modified by: Renee DesJarlais, Brian Shoichet

Overview

SPHGEN generates sets of overlapping spheres to describe the shape of a molecule or molecular surface (Kuntz *et al.,* 1982; DesJarlais *et al.,*1988). For receptors, a negative image of the surface invaginations is created; for a ligand, the program creates a positive image of the entire molecule. Spheres are constructed using the molecular surface described by Richards (1977) calculated with the program MS (Connolly, 1983a, 1983b). Each sphere touches the molecular surface at two points and has its radius along the surface normal of one of the points. For the receptor, each sphere center is "outside" the surface, and lies in the direction of a surface normal vector. For a ligand, each sphere center is "inside" the surface, and lies in the direction of a reversed surface normal vector. Spheres are calculated over the entire surface, producing approximately one sphere per surface point. This very dense representation is then filtered to keep only the largest sphere associated with each receptor surface atom. The filtered set is then clustered on the basis of radial overlap between the spheres using a single linkage algorithm. This creates a negative image of the receptor surface, where each invagination is characterized by a set of overlapping spheres. These sets, or "clusters," are sorted according to numbers of constituent spheres, and written out in order of descending size. The largest cluster is typically the ligand binding site of the receptor molecule. The program SHOWSPHERE writes out sphere center coordinates in PDB format and may be helpful for visualizing the clusters.

Input

The input file names and parameters are read from a file called `INSPH`, which should not contain any blank lines:

parameter	format	example
msfil	`A80`	`2ptc.ms`
srftp	`A1`	`R`
dentag	`A1`	`X`
dotlim	`F`	`0.0`
radmax	`F`	`4.0`
radmin	`F`	`1.4`
outfl	`A80`	`2ptc.clus`

msfil is the name of the file containing the molecular surface calculated using the program MS and must include surface normals. SPHGEN expects the Fortran format

(A3, I5, X, A4, X, 2F8.3, F9.3, X, A3, 7X, 3F7.3).

This format is quite different from the QCPE molecular surface file format. For more details, see the documentation for REFORMATMS and AUTOMS.

surftp indicates whether the spheres should lie "outside" the surface, as for a receptor (`R` or `r`), or "inside" the surface, as for a ligand (`L` or `l`).

dentag allows the user to specify that a subset of the surface points are to be used in calculating the spheres. This "density tag" may be set to the values `1, 4, 9`, or `0`, indicating that only points having 1, 4, 9, or 0, respectively, in column 42 of the molecular surface file will be used; alternatively, the value `X` or `x` indicates that all points will be used. It is recommended that `X` be used unless the system is particularly large (>75,000 surface points). It is most efficient to use a partial molecular surface (if it is known in advance which region is of interest) as the calculation time scales approximately with the square of the number of points.

dotlim is used to prevent the generation of large spheres whose points of surface contact are quite close together. Each pair of points *i* and *j* are examined as potential sphere-defining points. *dotlim* is the lower limit on the dot product of the vector from *i* to *j* and the vector from the sphere center to *j* for points defining a sphere. *dotlim* is typically set to 0.0, although possible values range from –1.0 to 1.0; negative values, however, may be useful for flat sites such as the major groove of B-form DNA.

radmax is the maximum sphere radius in Angstroms. Spheres with radii larger than *radmax* are discarded. This is important for the clustering done within SPHGEN, where clusters are defined as sets of overlapping spheres. Decreasing *radmax* decreases the cluster sizes by eliminating large "connector" spheres. In general, values from 4.0 to 5.0 Angstroms are used; values of 0.0 or less default to 5.0 Angstroms.

radmin is the minimum sphere radius in Angstroms. Spheres with radii smaller than *radmin* are discarded. This should be unnecessary because the molecular surface should not produce spheres of radius less than the probe radius. However, some versions of MS occasionally place surface points very close together. This can result in SPHGEN generating very small spheres which are not useful in characterizing the shape of the active site. It is generally advisable to keep spheres with radii equal to the probe radius (typically 1.4 or 1.5 Angstroms). Note that *radmin* can be set to 0.0 to allow the use of the "extra radius" surface developed by David Barry (unpublished results).

outfil is the name of the file to which the clustered spheres will be written.

Output

Some informative messages are written to a file called `OUTSPH`. This includes the parameters and files used in the calculation. The spheres themselves are written to *outfil*. They are arranged in clusters with the cluster having the largest number of spheres appearing first. The sphere cluster file consists of a header followed by a series of sphere clusters. The header is the line

DOCK 3.5 receptor_spheres

followed by a color table. The color table contains color names (format `A30`) each on a separate line. As SPHGEN produces no colors, the color table is simply absent. The sphere clusters themselves follow, each of which starts with the line

cluster n number of spheres in cluster i

where `n` is the cluster number for that sphere cluster, and `i` is the number of spheres in that cluster. Next, all spheres in that cluster are listed in the format

(I5, 3F10.5, F8.3, I5, I2, I3)

where the values correspond to, respectively,

- The number of the atom with which surface point *i* (used to generate the sphere) is associated.
- The *x, y,* and *z* coordinates of the sphere center.
- The sphere radius.
- The number of the atom with which surface point *j* (second point used to generate the sphere) is associated.
- The critical cluster to which this sphere belongs.
- The sphere color. The color is simply an index into the color table that was specified in the header. So 1 corresponds to the first color in the header, 2 for the second, *etc.* 0 corresponds

to unlabeled.

The clusters are listed in numerical order from largest cluster found to the smallest. At the end of the clusters is cluster number 0. This is not an actual sphere cluster, but a list of *all* of the spheres generated whose radii were larger than the minimum radius, *before* the filtering heuristics (*i.e.* allowing only one sphere per atom and using a maximum radius cutoff) and clustering were performed. Cluster 0 may be useful as a starting point for users who want to explore a wider range of possible clusters than is provided by the standard SPHGEN clustering routine. The program CLUSTER takes the full sphere description as input, and allows the user to explore different sphere descriptions of the site. This is particularly useful for macromolecular docking; it is often inefficient to use spheres that fill the entire volume of the "ligand" macromolecule. In addition, only a portion of a cavity in the "receptor" macromolecule may be of interest for docking purposes. If the standard clustered output from SPHGEN provides a satisfactory description of the ligand molecule or receptor site, running CLUSTER is not necessary.

The program creates three temporary files: `temp1.ms`, `temp2.sph`, and `temp3.atc`. These are used internally by SPHGEN, and are deleted upon completion of the computation.

UCSF LIBRARY

Accessories

ADDPRH

Author: Andrew Leach

ADDPRH is used to add hydrogens atoms to proteins. Either PDB or AMBER atom names can be specified. Hydrogens are added in "favorable" geometries, but this does not take into account intramolecular hydrogen bonding. Hydrogens are also added to waters; to add some variety into the orientation of these hydrogens, a random orientation about the three euler angles is performed before adding the hydrogens.

AUTOMS

Author: Andy Good, Daniel Gschwend

AUTOMS is an extremely useful tool for setting up for SPHGEN. This script converts PDB files to QCPE MS input format, runs a QCPE MS surface calculation, converts the resulting surface to UCSF MS format with REFORMATMS, creates a SYBYL dot file of the surface (if MS2DOT is available), and prepares an `INSPH` file for running SPHGEN. The requirements are only two files: a receptor PDB file, and an `exclude.pdb` file. The `exclude.pdb` file is a subset of the receptor and contains the residues which should not be surfaced in the MS calculation. This file can be created in several ways. The easiest, if SYBYL is available, is to select all residues in the receptor within some radius of a known ligand, invert the selection, and write out the file in Brookhaven format as `exclude.pdb`. One alternative that does not require SYBYL is to use GET_NEAR_RES to locate all receptor atoms or residues within a specified radius of a known ligand and invert the resulting pdb file using INVERTPDB.

Usage: `autoMS` *`receptor_PDB_file [surface_density] [probe_radius]`*

where *`surface_density`* and *`probe_radius`* are optional, defaulting to 3.0 dots/Å2 and 1.4Å, respectively. **Note**: before using the first time, the directory specification for the DOCK hierarchy must be updated inside the AUTOMS script.

CHARGE

Author: Daniel Gschwend

CHARGE is a simple NAWK script which provides residue composition for a protein, including number of charged residues and total charge. GRID's output should agree with the value generated from this program.

Usage: `charge` *`pdbfile`*

CHEMPROP

Author: Renee DesJarlais

Overview

The interactive program CHEMPROP is designed to aid the user in examining the properties of a receptor in the vicinity of a small molecule that has been oriented using DOCK. The program takes as input a receptor coordinate file, a file containing the coordinates of one or more ligands, and several parameters, described below, depending on the option chosen. There are two options: electrostatics and hydrogen bonding.

In the electrostatic option, the electrostatic potential from the receptor is calculated at each ligand molecule atom center. The electrostatic potential at the position of ligand atom *j*, e_j, is calculated using equation 1, where q_i is the partial charge on the receptor atom *i*, *D* is the dielectric constant, and r_{ij} is the distance between atoms *i* and *j*:

$$e_j = \sum_{i=1}^{lig} \frac{q_i}{Dr_{ij}} \qquad \text{Equation 10}$$

Only the receptor atoms contribute to the value of the electrostatic potential. The partial atomic charges used are those from the AMBER united-atom force field (Weiner, *et al.*, 1984). Only standard amino acid residues can be accommodated by this program. The receptor file must include the hydrogen atoms attached to nitrogens, hydroxyl oxygens, and sulfurs, and the lone pairs on the sulfurs. A new PDB-format coordinate file is written out for the ligand molecules, where the electrostatic potential is printed in the temperature factor column. The molecules can then be displayed using a molecular graphics package and colored according to electrostatic potential.

The hydrogen bond option helps the user identify places on the ligands where it might be appropriate to design in a group capable of hydrogen bonding to the receptor. Potential hydrogen bond positions are identified as any ligand atom within a user-specified distance of a receptor nitrogen or oxygen atom. This option was intended mainly for use with ligand heavy atom (non-hydrogen) coordinates only. Two files are output: a coordinate file, and a file listing the potential hydrogen bonds for each ligand in the input file. In the coordinate file, each potential hydrogen bond is written in PDB format as a residue with two atoms. One atom is located at the receptor nitrogen or oxygen and the other is located at the ligand atom. The residues are separated by TER cards. The residue and atom names depend on the protein atom. The residue is named ACC if the protein atom is a carbonyl oxygen, DNR if the protein atom is an amide or amine nitrogen, and `doa` if the protein atom is a hydroxyl oxygen or a histidine side chain nitrogen. The receptor and ligand atoms in an ACC residue are named `A` and `D`, respectively; in a DNR residue they are named `D` and `A`, respectively; and in a DOA residue they are named `E1` and `E2`, respectively. Viewing these possible hydrogen bonds with the receptor and ligand is useful for design purposes and assessing whether the angles are consistent with strong hydrogen bonding.

Usage

The input file names and parameters can be entered interactively. Interactive use is reasonable when fewer than about 30 ligands are to be examined. The user is prompted for the names of the receptor and ligand files and the type of calculation (electrostatic potential or hydrogen bond). If an electrostatic potential calculation is being performed, the user must select a constant or distance-dependent dielectric, a cutoff distance, and a name for the output file. If a hydrogen bond calculation is being performed, the user must enter a cutoff distance for hydrogen bonds and names for the output files. Finally, after performing one calculation, the user is given the option of performing another without exiting from the program.

CLUSTER

Authors: Brian Shoichet, Irwin D. Kuntz

Overview

CLUSTER allows greater flexibility in creating a sphere description of a site or molecule. It is particularly useful in macromolecular docking, and in general, when the original cluster file from SPHGEN does not adequately describe the site or molecule of interest.

The CLUSTER program is a more elaborate version of the cluster subroutine in SPHGEN (Kuntz *et al.*, 1982). A single-linkage clustering algorithm is applied, based on the radial overlap between spheres. Unlike SPHGEN, CLUSTER does not heuristically remove spheres; it can operate on the total set of possible spheres rather than just the largest sphere per surface atom. This complete description is contained in the cluster 0 from SPHGEN. User-defined criteria control the clustering process; clusters can be tailored to a certain size (number of spheres), a certain range of sphere radii, or a certain region of space. The program allows one to try different clustering parameters without rerunning SPHGEN.

Input

The program can be run either interactively or from a command file. The following is an explanation of a sample command file. These command files should not contain any blank lines.

parameter	format	example
clufil	`A80`	`2ptc.all`
nclus	`I`	`1`
maxrad	`F`	`5.0`
m2xrad	`F`	`5.0`
povlap	`F`	`10.0`
clusiz	`I`	`60`
minsiz	`I`	`20`
minflg	`I`	`1`
outfil	`A80`	`2ptc.all.rcl`
yn	`A1`	`y`
(if *yn* is `y` or `Y`):		
minrad	`F`	`1.3`
rincr	`F`	`0.2`
nearnb	`F`	`0.5`
nearad	`F`	`0.25`

clufil is the file containing the input spheres from SPHGEN.

nclus is the number of the cluster to recluster (there is generally more than one in the original SPHGEN cluster file). When the full set of spheres is being used, there is only one "cluster" and *nclus* should be set to 0.

maxrad is the primary maximum sphere radius for clustering, in Angstroms. Only spheres with radii less than or equal to *maxrad* can be used as linkers between groups of spheres, making them into a single larger cluster. Values from 2.5 to 5.0 Angstroms are generally most useful. *maxrad* also defines the end point for analytical clustering (see below); it is the final value of *rcut*.

m2xrad is the secondary maximum sphere radius for clustering, in Angstroms. This variable allows spheres with radii larger than *maxrad* to be included in clusters, but does not allow them to act as linkers. *m2xrad* must be equal to or greater than *maxrad*; smaller values default to *maxrad*. All spheres exceeding the *m2xrad* criterion will be discarded. *m2xrad* is typically set to *maxrad* for analytical clustering and 5.0 Angstroms otherwise.

povlap is the percent radial overlap between two spheres necessary to define a pair. If this variable is set to 0.0, spheres will be defined as overlapping when they intersect to any degree. The larger the value of *povlap*, the greater the overlap necessary to define the spheres as a pair for cluster purposes. Typical values range from 0.0 to 20.0.

clusiz is the maximum number of spheres allowed to be in a cluster. Growth of a cluster is frozen when this limit is reached; spheres that would otherwise be added are discarded. Limiting the cluster size leads to decreased coalescence and therefore greater numbers of clusters. Values of 50 to 75 are suggested.

minsiz is the minimum number of spheres a cluster must have to be included in the output. *minsiz* must be less than *clusiz*; values of 20 to 30 are suggested.

minflg is the minimum number of flagged spheres a cluster must have to be included in the output. Flagging is done by placing any non-blank characters following the information for the sphere(s) of interest in *clufil*. ***The flagging feature is no longer supported.***

yn indicates whether analytical clustering will be done. Analytical clustering refers to iteratively increasing the value of the primary maximum sphere radius. It is especially useful when the input sphere set is large (>1000, as when the full sphere description is being used). If *yn* equals `N` or `n`, analytical clustering will not be done, and no further input is read. Analytical clustering replaces *maxrad* with the variable *rcut*, which increases from *minrad* to *maxrad* in step sizes of *rincr*. Each value of *rcut* corresponds to a cycle of clustering. In this way, the user can quickly determine which parameters will yield a cluster of the desired size. For a set of 1000 spheres, a typical analytical run with averaging takes about 20 seconds on a Silicon Graphics Iris 4D/25 workstation. Most of the CPU time is spent in averaging the spheres; for this reason, run time scales approximately with the square of the number of spheres.

minrad is the starting value for *rcut* in analytical clustering.

rincr is the incremental increase in *rcut* per iteration in analytical clustering.

nearnb is the maximum distance between the centers of spheres that may be averaged into a composite sphere, for analytical clustering. This variable is used to simplify very large sets of spheres. When clusters are written out, only the sphere closest to the composite will be included. A value of 0.5 Angstroms is reasonable for sets of approximately 1000 spheres.

nearad is the maximum difference in magnitude between the radii of spheres that may be averaged into a composite sphere, for analytical clustering. A value of 0.25 Angstroms is reasonable for sets of approximately 1000 spheres.

Output

For analytical clustering, the output sphere cluster file consists of several sphere cluster files concatenated together. Each group begins with its own header (`dock 3.5 receptor_spheres`, *etc.*). The user must hand-edit this file to select the best group of clusters.

COLSPH

Author: Mike Connolly

COLSPH reads and writes a sphere cluster file, adding a color (label) table and sphere labels. The color is determined by evaluating either a Delphi map or a force-field grid at the sphere center, and comparing that value to the ranges specified in a user-defined *range file*, which defines ranges for a series of colors. The program prompts for the information it needs:

- scoring map type (2 = Delphi, 3 = force-field)
- the Delphi map file name or the force-field grid prefix
- range file name
- input sphere cluster file name

- output sphere cluster file name

The range file is the only one that needs explanation. Its format is as follows:

`color_name range_min range_max`

The spacing between the values does not matter. The `color_name` can be up to 30 characters in length. The `range_min` and `range_max` must be *real* numbers, and specify the lower and upper ends of the range respectively. There may be up to 100 colors and 200 range statements.

CONDENSE

Author: Daniel Gschwend

CONDENSE takes as input a residue list generated by GET_NEAR_RES and compacts it into a format suitable for an MS `-i` file (for the UCSF version of MS - for the QCPE MS version, see AUTOMS). This functionality comes in handy when attempting to generate an MS surface for a portion of a receptor. The program GET_NEAR_RES can be used to obtain a listing of all receptor atoms within a specified distance of a small molecule, *e.g.* a crystallographically observed ligand. CONDENSE will then reformat and compact the listing to less than 100 entries for compatibility with UCSF MS. This program supports command-line operation: type `condense -h` for details.

CONECT

Author: Elaine Meng

CONECT is an interactive program that appends CONECT records to a PDB file containing the coordinates of a single molecule. The user is prompted for the names of the input and output files. The presence of a bond is determined as follows: two atoms are bonded if the distance between them is less than or equal to the sum of their covalent bond radii plus a tolerance of 0.4 Angstroms.

CONVSYB

Author: Elaine Meng

The interactive program CONVSYB converts SYBYL MOL2 format files into a number of other formats useful for DOCK and some of the accessory programs. Output options include "extended PDB format," DOCK 3.0 database format, and standard PDB format. CONVSYB assumes that each @<TRIPOS>MOLECULE record type indicator corresponds to a single covalently bonded structure, and uses the first 9 characters of the line following this record as the refcode for the structure. Multi-MOL2 files are handled correctly.

FDAT2PDB

Author: Daniel Gschwend

Converts a Cambridge Crystallographic Database FDAT file (as from a search with QUEST) to PDB format. Multiple structures in the same FDAT file are handled correctly.

Usage: `fdat2pdb` *`cambridge_prefix`* (an input file called *`cambridge_prefix`*`.fdat` is expected, and output file called *`cambridge_prefix`*`.pdb` will be created).

GET_NEAR_RES

Author: Daniel Gschwend

Given a ligand PDB file and a receptor PDB file, performs either of two functions:

- Writes a PDB file containing all atoms of all residues in the receptor which have their **C**α atoms within a specified distance of the ligand. A list file is also written which gives the closest Cα-to-ligand distance for each residue written.
- Writes a PDB file containing all atoms of all residues in the receptor which have **any** atom within a specified distance of the ligand. A list file is also written which gives the closest receptor-ligand distance for each residue written.

The list file output may be converted to an UCSF MS `-i` file with the CONDENSE program. This utility can also be used with the AUTOMS program using QCPE MS surfaces: to generate an `exclude.pdb` file from GET_NEAR_RES output, see INVERTPDB.

HBDATA

Author: Andy Good

This program automatically runs Goodford's grid program (version 11.01, distributed independently; Goodford, 1985) for a variety of probes, creating SYBYL contour files and files containing centers of favorable interaction energy for SPHGEN file creation. It also runs GRIN and will stop if GRIN encounters a problem with the PDB file. The program requires an input file called `HBIN` which should follow this example:

`3dfr`	name of parent protein (PDB file name - used for file naming)
`/bert/grid11.01`	root directory for GRID
`-12.0`	minimum *x* coordinate
`26.0`	minimum *y* coordinate
`4.0`	minimum *z* coordinate
`9.0`	maximum *x* coordinate
`48.0`	maximum *y* coordinate
`33.0`	maximum *z* coordinate
`1.0`	grid density - # of planes per Angstrom of GRID map
`-7.0`	highest permitted interaction energy included in site point creation
`1.0`	exclusion sphere to remove local low energy pts. near each local
minimum	
`2`	# of probes in probe list that will be used - first two are `N3+`, `O::`

Notes for use in conjunction with MOL2SPH: The GRID energy of each point is assigned to the charge field for reference purposes and atom types are set to dummy. Both of these fields must be corrected, the SPHGEN file converted into PDB format using SHOWSPHERE, the resulting atom types modified and combined within SYBYL before they can be written to MOL2 format in preparation for MOL2SPH conversion.

INVERTPDB

Author: Daniel Gschwend

A shell script to extract all atoms in a larger PDB file which are not in a smaller PDB file, where the latter is a subset of the former. For example, after acquiring a PDB file of all protein atoms or residues within a specified radius of a known ligand with GET_NEAR_RES, the inverse atoms (*i.e.* those far away from the ligand) may be selected to generate an `exclude.pdb` file for use with PDB2MS or AUTOMS.

Usage: `invertPDB` *`larger_PDB_file smaller_PDB_file`* `>` *`inverted_PDB_file`*

IDTOSYB

Author:Elaine Meng

This program is designed to convert PDB files into SYBYL MOL2 format files. SYBYL does a less than satisfactory job of assigning atom types and bonds when reading in small molecule PDB files. IDTOSYB uses bond length data from the Cambridge Structural Database to assign atom types in a far superior fashion, although still not perfect. Hydrogens need not be present in the input, and multiple ligands in the same file are processed if separated by TER cards.

Usage: see PDB2SYB, a front-end to IDTOSYB.

MOL2SPH

Author: Andy Good

Takes SYBYL MOL2 format files and converts them into SPHGEN format files, allowing automatic coloring of the receptor. Currently, `N.3` atoms are classed as H-bond donors, `O.3` as H-bond acceptors, and `S.3` as hydrophobes, but these color setting can be changed to user preferences by simply hacking the file. The charge value can also be used to assign critical clusters, with the critical cluster number equal to the truncated value of the charge.

MS2DOT

Author: Andy Good

MS2DOT converts UCSF MS surface files into SYBYL dot files. This allows visualization in SYBYL of the surface created for use with SPHGEN. *Note: this program is not installed by default, as it requires* SYBYL *programming libraries for compilation.* To install this program you must explicitly compile it with `make ms2dot` in the `accessories` source directory, provided you have SYBYL libraries installed. Please read the `Makefile` in the `accessories` directory for further details.

OLDSCORE

Author: Elaine Meng

OLDSCORE is an interactive program that allows scoring of DOCK output according to the "old," exponential algorithm (as in version 1.1; DesJarlais *et al.,* 1988). The user is interactively asked for the names of the receptor file, the ligand file, and a file for output, plus the values of *concut, dmin,* and *discut.* There may be many ligands in the ligand file, as long as each ends with a TER card. Bad contacts are counted. There are slight differences in the scores directly from DOCK 1.1 and the scores reported by this program, because the coordinates of ligand orientations are rounded to three decimal places when written out.

PDB2MS

Author: Andy Good

This program is used to create input files for QCPE MS from standard PDB files. The receptor file must be called `fort.1`; the residues of the receptor which should be included in the surface calculation but not have a surface drawn should be placed in a file called `fort.2`; output is written to `fort.3`. This program is part of the automated process of AUTOMS.

PDB2SYB

Author: Daniel Gschwend

This program is a front-end to IDTOSYB.

Usage: `pdb2syb` *`PDB_file`*

Note: before using the first time, update the pathnames inside the script for location of IDTOSYB.

PDBRENUM

Author: Daniel Gschwend

PDBRENUM is a versatile residue and atom renumbering utility. It provides many features with respect to where to start numbering, how to treat separate molecules, and also supports individual treatment of water molecules, including their separation by TER cards for easier visualization. PDBRENUM also attempts to update CONECT records when atoms are renumbered. This program supports command-line operation: type `pdbrenum -h` for details.

PDBTOSPH

Author: Elaine Meng

PDBTOSPH makes a sphere cluster file out of the non-hydrogen ATOM records (not HETATM) in a PDB file.

PTRENTRY

Author: Todd Ewing

PTRENTRY is a self-running nawk script which extracts selected molecule entries from a PTR format file and displays the data for each molecule in a readable format. A single molecule, or a range of molecules, may be selected.

- To extract the first molecule, type: `ptrentry file.ptr 1`
- To extract the first 10 molecules, type: `ptrentry file.ptr 1 10`

PTRFIELD

Author: Todd Ewing

PTRFIELD is a self-running nawk script which extracts selected fields from a PTR format file. It is useful to shuffle the field order in a file to make sorting sorting easy. This would be accomplished as follows:

- `ptrfield file.ptr NRG FILE FPOS END | sort +1n > file_sort.ptr`

QCPE_MS

Author: Mike Connolly

SPHGEN requires that a molecular surface be calculated and that it be converted to the UCSF MS format using REFORMATMS. QCPE_MS is distributed only as a tool for the AUTOMS script. You should contact QCPE directly for obtaining the complete software package for MS (#429) - see Sources on page 174 for contact information.

REFORMATMS

Author: Renee DesJarlais

REFORMATMS converts an MS file of QCPE format to an MS file of the format read by SPHGEN. REFORMATMS requires a Brookhaven Protein Data Bank coordinate file as well as the QCPE MS file as input. The program is interactive.

The QCPE MS file must be in the long format: The Fortran format for this file and the information contained in it are listed below:

```
(3I5, I2, 3F9.3, 4F7.3, I2)
```

n1, n2, n3, shape, x, y, z, area, xn, yn, zn, buried

n1	atom number of atom that surface point is on or closest to
n2	other atom that probe touches (0 for convex)
n3	third atom that probe touches, *n3>n2* (0 for convex and saddle)
shape	1: convex, 2: saddle, 3: concave
x, y, z	coordinates of surface point
area	solvent-accessible area
xn, yn, zn	components of the unit vector normal
buried	0: exposed, 1: buried, blank: not determined

The lines must be sorted by *n1*.

The MS format that SPHGEN reads and the information that it contains are listed below:

(A3, I5, 2X, A3, 3(F8.3, X), X, A3, 4F7.3)

resnm, nres, atnm, x, y, z, srftag, area, xn, yn, zn

resnm	residue name of the closest atom, or the atom itself (if *srftag* = A)
nres	residue number of the closest atom, or the atom itself (if *srftag* = A)
atnm	name of the closest atom, or the atom itself (if *srftag* = A)

	x, y, z	coordinates of the point or atom
point	*srftag*	A: atom, SR0: reentrant point, SS0: saddle point, SC0: convex
	area	solvent-accessible area (blank if *srftag* = A)
	xn, yn, zn	components of the unit vector normal (blank if *srftag* = A)

RMSD

Author: Daniel Gschwend

Calculates root mean squared deviation in Angstroms per atom for two PDB files. Treatment of hydrogens is optional. Also, one may compare two files side by side, or use the first molecule in the first file as a reference for all molecules in the second file. All atoms must be in identical order and have the same atom names for both input files. This program supports command-line operation: type `rmsd -h` for details.

SDF2MOL2 & SYBDB

Author: Daniel Gschwend

As an interface to the commonly used databases from Molecular Design Limited, we are providing two conversion schemes to convert SD files (such as from ISIS) to SYBYL MOL2 format files.

The conversion scheme described here was developed to be easy to use and accurate in its atom-typing and partial charge computation. Although still an area of active development, it has to date been tested visually on several thousand compounds.

This scheme consists of two programs run sequentially, SDF2MOL2 is written in Fortran and SYBDB in SYBYL's programming language, SPL. Taken together, one can convert an MDL SDF-format database into a SYBYL MOL2 format database which has appropriate SYBYL atom types assigned, hydrogens added, and partial charges computed. To find out more about the conversion process, including how to use it, please consult the `00README` file in the directory `./source/database/sdf2mol2` under the DOCK root.

Note: The second phase conversion requires SYBYL for hydrogen addition and charge computation. The former program, SDF2MOL2, may still be of some use to users who do not have SYBYL but have molecular modeling packages than can read MOL2 format (e.g. INSIGHT). Hydrogen addition, substructure removal, and charge computation must then be performed within the context of your own modeling package.

SHOWBOX

Author: Elaine Meng

SHOWBOX is an interactive program that allows visualization of the location and size of the grids that will be calculated by the program GRID, using any graphics program that can display PDB format. The user is asked whether the box should be automatically constructed to enclose all of the spheres in a cluster. If so, the user must also enter a value for how closely the box faces may approach a sphere center (how large a "cushion" of space is desired) and the sphere cluster filename and number. If not, the user is asked whether the box will be centered on manually entered coordinates or a sphere cluster center of mass. Depending on the response, the coordinates of the center or the sphere cluster filename and number are requested. Finally, the user must enter the desired box dimensions (if not automatic) and a name for the output PDB-format box file.

SHOWSPHERE

Authors: Stuart Oatley, Elaine Meng, Daniel Gschwend

SHOWSPHERE is an interactive program; it produces a PDB-format file of sphere centers and an MS-like file of sphere surfaces, given the sphere cluster file and cluster number. The surface file is not in QCPE MS format (see the documentation on REFORMATMS on page 158 for further details); generation is optional. The user may specify one cluster or "all," and multiple output files will be generated, with the cluster number appended to the end of the name of each file. The input cluster file is created using SPHGEN. SHOWSPHERE requests the name of the sphere cluster file, the number of the cluster of interest, and names for the output files. Information is sent to the screen while the spheres are being read in, and while the surface points are being calculated.

SPLITMOL

Author: Daniel Gschwend

A full-featured structure file splitting utility which accepts either PDB or SYBYL MOL2 format files. A range of molecules can be extracted from the input, and a user-specifiable number of molecules is put into each file created. This program supports command-line operation: type `splitmol -h` for details.

Molecule File Formats

Dock programs read and write in several coordinate formats.

- SYBYL MOL2 format
- PDB format
- PTR format
- SPH format

SYBYL MOL2 format

This format is used for general molecule input and output of DOCK. Although previous versions of DOCK supported an extended PDB format to store molecule information, the current version now uses MOL2 as the primary molecule format. This format has the advantage of storing all the necessary information for atom features, position, and connectivity. It is also a standardized format that other modeling programs can read.

Specification

Please refer to sybyl documentation for format specifications.

Of the many record types in a MOL2 file, DOCK recognizes the following: MOLECULE, ATOM, BOND, SUBSTRUCTURE and SET. In the MOLECULE record, DOCK utilizes information about the molecule name and number of atoms, bonds, substructures and sets. In the ATOM record DOCK utilizes information about the atom names, types, coordinates, and partial charges. In the BOND record, DOCK utilizes the atom identifiers for the bond. In the SUBSTRUCTURE record, DOCK records the fields, but does not utilize them. The SET records are entirely optional. They are used only in special circumstances, like when ligand flexibility is considered..

Example

This example file illustrates all the elements of the MOL2 file read and written by dock. It includes optional SET records which are used by the ligand flexibility routines.

```
@<TRIPOS>MOLECULE
Histidine
   20    20     1     0     2
SMALL
GASTEIGER
****
Histidine with Main Chain as Rigid Anchor
```

```
@<TRIPOS>ATOM
     1 N1        -1.0947   0.5371    1.7186 N.4      1 <1>        0.2252
     2 C2        -0.9885   0.9170    0.2765 C.3      1 <1>        0.0213
     3 C3        -0.2043  -0.1565   -0.4766 C.3      1 <1>        0.0354
     4 C4        -2.3725   1.0376   -0.3154 C.2      1 <1>        0.0897
     5 O5        -2.7546   2.1336   -0.8057 O.co2    1 <1>       -0.5442
     6 C6         1.1797  -0.2771    0.1153 C.2      1 <1>        0.0328
     7 N7         2.2791   0.4215   -0.2757 N.pl3    1 <1>       -0.3074
     8 C8         1.5387  -1.0911    1.1173 C.2      1 <1>        0.0462
     9 C9         3.3256   0.0285    0.4990 C.2      1 <1>        0.0853
    10 N10        2.9039  -0.8872    1.3511 N.2      1 <1>       -0.2465
    11 H11       -1.6259   1.2643    2.2287 H        1 <1>        0.2001
    12 O12       -3.1452   0.0423   -0.3188 O.co2    1 <1>       -0.5442
    13 H13       -0.1461   0.4545    2.1242 H        1 <1>        0.2001
    14 H14       -0.4726   1.8710    0.1898 H        1 <1>        0.0918
    15 H15       -0.7202  -1.1105   -0.3899 H        1 <1>        0.0385
    16 H16       -0.1270   0.1200   -1.5261 H        1 <1>        0.0385
    17 H17        2.3126   1.1114   -1.0125 H        1 <1>        0.1528
    18 H18        0.8943  -1.7774    1.6466 H        1 <1>        0.0845
    19 H19        4.3357   0.4040    0.4286 H        1 <1>        0.1000
    20 H20       -1.5855  -0.3703    1.8010 H        1 <1>        0.2001
@<TRIPOS>BOND
     1     1     2 1
     2     2     3 1
     3     2     4 1
     4     3     6 1
     5     4     5 ar
     6     6     7 1
     7     6     8 2
     8     7     9 1
     9     8    10 1
    10     9    10 2
    11     1    11 1
    12     4    12 ar
    13     1    13 1
    14     2    14 1
    15     3    15 1
    16     3    16 1
    17     7    17 1
    18     8    18 1
    19     9    19 1
    20     1    20 1

@<TRIPOS>SUBSTRUCTURE
     1 ****          1 TEMP              0 ****  ****    0 ROOT
@<TRIPOS>SET
ANCHOR           STATIC     ATOMS     <user>    **** Anchor Atom Set
1 2
RIGID            STATIC     BONDS     <user>    **** Rigid Bond Set
2 1 3
```

PDB format

This format should be used only for display purposes. Since it does not contain fields for atom types or partial charges, vital information will be lost if dock output is routinely stored in this format. It is recommended that all dock output be stored in either SYBYL MOL2 format or PTR format. If you site does not have access to a method to view MOL2 format, just convert files to PDB when vewing is necessary. See Molecule File Conversion on page 118 for instructions.

PTR format

Specification

PTR (or pointer) format is a compact representation for molecules which does not actually contain coordinates. Instead, it contains the transformations to the coordinates and "points" back to a source file where the untransformed coordinates reside.

All information about a molecule is packed into one line. Each item of data is identified by a field name. Some fields are required, like the molecule source information. Some are optional, like the coordinate transformation information. All others are considered comments and are ignored, like the molecule name and score information.

Table 20. PTR Format Data Fields

20A: Required PTR Fields

These fields must be present in every PTR record.

Field	Type	Description
`<FILE>`	`String`	Directory path and name of file which contains molecule.
`<FPOS>`	`Integer`	Byte position in file of molecule.
`<END>`		End of record flag.

20B: Optional PTR Fields(Part 1 of 2)

If these fields are present, they will be read and used to update the molecule. The transformation fields will modify the position or conformation of the molecule.

Field	Type	Description
`<TRANS>`	`Real[3]`	XYZ translation vector in Angstroms.
`<ROT>`	`Real[3]`	Quaternion rotation vector (unitless, range 0-1).
`<TORS>`	`Integer`	Number of rotatable torsions.
`<REFL>`	`Integer`	Flag for chiral reflection (0 or 1).

20B: Optional PTR Fields(Part 2 of 2)

If these fields are present, they will be read and used to update the molecule. The transformation fields will modify the position or conformation of the molecule.

Field	Type	Description
`<T1>,<T2>, ...`	`Integer, Real`	Bond identifier and torsion angle.
`<KEY>`[u]	`Integer`	Number of chemical keys.
`<KFOLD>`[u]	`Integer`	Flag for folded chemical keys.
`<KI1>,<KI2>, ...`[u]	`Integer`	Chemical key label and distance fingerprint

u. This field is only read during a chemical screening run.

20C: Comment PTR Fields

These fields are generated for output only. They are completely ignored during input.

Field	Type	Description
`<ID>`	`Integer`	Line position in ptr file.
`<SRC_ID>`	`Integer`	Line position in input ptr file.
`<NAME>`	`String`	Molecule name.
`<DESCR>`	`String`	Molecule description.
`<BMP>`	`Integer`	Number of bumps.
`<CNT>`	`Real`	Contact score.
`<CHM>`	`Real`	Chemical score.
`<NRG>`	`Real`	Total energy score.
`<INTRA>`	`Real`	Intramolecular component of score.
`<INTER>`	`Real`	Intermolecular component of score.
`<VDW>`	`Real`	VDW component of score.
`<ELE>`	`Real`	Electrostatic component of score.
`<RMSD>`	`Real`	RMS deviation of current orientation from input.

Example

Database Entry

```
<ID> 10 <NAME> CAMYLOFIN_C00000105 <DESCR> **** <FILE> /marco/db/
mol2.95.1/cmc/cmc.2.mol2 <FPOS> 46204 <END>
```

Flexible Docking Output

```
<ID> 5 <SRCID> 5 <NAME> DANAZOL_C00002993 <DESCR> **** <FILE> db/
db3.mol2 <FPOS> 14144 <TRANS> 5.00991 29.9234 16.2096 <ROT> 0.270641
1.05337 0.0158567 <TORS> 2 <TANCHOR> 1 <T1> 20 153.624 <T2> 21 -28.0673
<REFL> 0 <BMP> 2 <NRG> -32.43 <INTRA> -0.48 <INTER> -31.95 <VDW> -32.53
<ELECTRO> 0.09 <RMSD> 34.95 <END>
```

Chemical Screen Database Entry

```
<ID> 1 <NAME> DAZOQUINAST_C00005118 <DESCR> **** <FILE> /marco/db/
mol2.95.1/cmc/cmc.1.mol2 <FPOS> 9689688 <KEY> 5 <KFOLD> 1 <KI1> 0 <KJ1>
10 2d7 <KJ2> 35 3fff <KJ3> 0 0 <KJ4> 20 f65 <KJ5> 0 0 <KI2> 1 <KJ2> 21
fd <KJ3> 0 0 <KJ4> 28 ab7e <KJ5> 0 0 <KI3> 2 <KJ3> 0 0 <KJ4> 0 0 <KJ5>
0 0 <KI4> 3 <KJ4> 6 c1a <KJ5> 0 0 <KI5> 4 <KJ5> 0 0 <END>
```

Usage

Check out the PTRENTRY and PTRFIELD utilities to help manipulate PTR files.

The <FILE> field can contain the relative path or absolute path of a filename. If you use the absolute path, then the ptr file can be used in any other directory. Otherwise, the ptr file is only useful in the idrectory in which it was originally made.

SPH format

Please refer to SPHGEN on page 147 for a description of the file format.

Parameter Files

Introduction

The parameter files contain atom and bond data needed during DOCK calculations. The definition (*.defn) files contain atom and bond labeling data. The table (*.tbl) files contain additional data for chemical interactions and flexible bond torsion positions. They may be edited by the user.

Atom definition rules

The definition files use a consistent atom labeling convention for which an atom in virtually any chemical environment can be identified. The specification of adjacent atoms is nested using the elements listed in Table 21. Some sample definitions are provided in Table 22.

- Each element must be separated by a space.
- If more than one adjacent atom is specified, then ALL must be present (i.e. a boolean AND for rules within a line).
- If a label can have multiple definition lines, then any ONE of them must be satisfied for inclusion (i.e. a boolean OR for rules on different lines).

Table 21. Atom definition elements

element	function
atom type	Specifies partial or complete atom type. A partial specification is more general (i.e. "C" versus "C.3"). An asterisk (*) specifies ANY atom type.
( )	Specifies atoms that must be bonded to parent atom.
[ ]	Specifies atoms that must NOT be bonded to parent atom.
integer	Specifies the number of an atom that must be bonded.

Table 22. Example definitions

example	explanation
`C.2 ( 2 O.co2 )`	A carboxylate carbon.
`.3 [ 3 H ]`	Any sp^3 hybridized atom that is not attached to three hydrogens.
`C. [ O. ] [ N. [ 2 O.2 ] [ 2 C. ] ]`	Any carbon not attached to an oxygen or a nitrogen (unless the nitrogen is a nitro or tertiary nitrogen).

vdw.defn

This file contains atom labels and definitions for van der Waals atom typing.

- The following data types are associated with each atom: VDW radius, VDW well-depth, flag for heavy atom, number of attached atoms.
- Some labels are used only for the united-atom model, some for only the all-atom model, and some for either.
- A label may have multiple definitions.

Table 23. Sample entries from vdw.defn

```
name            Carbon_sp/sp2
atom_model      either
radius          1.850
well_depth      0.120
heavy_flag      1
valence         4

definition      C
--------------------------------------------------
name            Carbon_All_sp3
atom_model      all
radius          1.800
well_depth      0.060
heavy_flag      1
valence         4

definition      C.3
--------------------------------------------------
name            Carbon_United_CH3
atom_model      united
radius          2.000
well_depth      0.150
heavy_flag      1
valence         4

definition      C. ( 3 H )
```

chem.defn

This file contains labels and definitions for chemical labeling.

- Nothing in addition to a label is assigned to an atom.
- A label may have multiple definition lines.

Table 24. Entries from chem.defn

```
name            hydrophobic

definition      C. [ O. ] [ N. [ 2 O.2 ] [ 2 C. ] ] ( * )
definition      N.pl3 ( 3 C. )
definition      Cl ( C. )
definition      Br ( C. )
definition      I ( C. )
definition      C.3 [ * ]
--------------------------------------------------------------
name            donor

definition      N. ( H )
definition      N.4 [ * ]
--------------------------------------------------------------
name            acceptor

definition      O. [ H ] [ N. ] ( * )
definition      O.3 ( 1 * ) [ N. ]
definition      O.co2 ( C.2 ( O.co2 ) )
definition      N. [ H ] [ N. ] [ O. ] [ 3 . ] ( * )
definition      O.2 [ * ]
--------------------------------------------------------------
name            polar

definition      O.3 ( H )
definition      F [ * ]
```

chem_match.tbl

This file contains the interaction matrix for which chemical labels can form an interaction in matching.

- The labels must be identical to labels in chem.defn.
- The `table` flag indicates the beginning of the interaction table.
- Compatible labels are identified with a one, otherwise a zero.

Table 25. Example of chem_match.tbl

```
label   null
label   hydrophobic
label   donor
label   acceptor
label   polar

table
1
1       1
1       0       1
1       0       0       1
1       0       1       1       1
```

chem_screen.tbl

This file contains the interaction matrix for how to scale the contribution of each chemical key when computing the overall similarity.

- The labels must be identical to labels in chem.defn.
- The `table` flag indicates the beginning of the interaction table.
- Each element is a scaling factor.

Table 27. Example of chem_screen.tbl

```
label    null
label    hydrophobic
label    donor
label    acceptor
label    polar

table
0
0        1
0        1        1
0        1        1        1
0        1        1        1        1
```

flex.defn

This file contains labels and definitions for flexible bond identification.

- The `drive_id` field corresponds to a torsion type in the flex_drive.tbl file.
- The `minimize` field is a flag for whether the bond may be minimized.
- Two definition lines must be present. Each definition corresponds to an atom at either end of the bond.

Table 28. Selected entries from flex.defn

```
name          sp3-sp3
drive_id      3
minimize      1

definition    .3 [ 3 H ] [ 3 O.co2 ]
definition    .3 [ 3 H ] [ 3 O.co2 ]
-------------------------------------------
name          sp3-sp2
drive_id      4
minimize      1

definition    .3 [ 3 H ] [ 3 O.co2 ]
definition    .2 [ 2 H ] [ 2 O.co2 ]
-------------------------------------------
name          sp2-sp2
drive_id      2
minimize      0

definition    .2 [ 2 H ] [ 2 O.co2 ]
definition    .2 [ 2 H ] [ 2 O.co2 ]
```

flex_drive.tbl

This file contains torsion positions assigned to each rotatable bond when the `torsion_drive` parameter is used in DOCK.

- The `drive_id` field corresponds to each torsion type.
- The `positions` field specifies the number of torsion angles to sample.
- The `torsions` field specifies the angles that are sampled.

Table 29. Selected entries from flex_drive.defn

```
drive_id        2
positions       2
torsions        0 180
-------------------------------------
drive_id        3
positions       3
torsions        -60 60 180
-------------------------------------
drive_id        4
positions       4
torsions        -90 0 90 180
-------------------------------------
drive_id        6
positions       6
torsions        -150 -90 -30 30 90 150
```

flex_drive.tbl

This file contains torsion positions assigned to each rotatable bond when the torsion_drive parameter is used in DOCK.

- The drive_id field corresponds to each torsion type.
- The positions field specifies the number of torsion angles to sample.
- The torsions field specifies the angles that are sampled.

Table 29. Selected entries from flex_drive.defn

```
drive_id        2
positions       2
torsions        0 180
---------------------------------------------
drive_id        3
positions       3
torsions        -60 60 180
---------------------------------------------
drive_id        4
positions       4
torsions        -90 0 90 180
---------------------------------------------
drive_id        6
positions       6
torsions        -150 -90 -30 30 90 150
```

Sources

compiled by C.Corwin

Available Chemicals Directory

MDL Information Systems, Inc.
14600 Catalina Street
San Leandro, CA 94577
phone (510) 895-1313
fax (510) 352-2870

Brookhaven Protein Data Bank

Protein Data Bank
Chemistry Department, Building 555
Brookhaven National Laboratory
Upton, NY 11973
phone (516) 282-3629
fax (516) 282-5751
e-mail pdb@bnl.gov
gopher://pdb.pdb.bnl.gov/11

Cambridge Crystallographic Database

For industrial users:

The Cambridge Crystallographic Data Centre
12 Union Road
Cambridge
CB2 1EZ
U.K.
phone +44 223 336408
fax +44 223 336033

For US academic users:

Dr. William L. Duax
Medical Foundation of Buffalo
Research Laboratories
73 High Street
Buffalo, NY 14203-1196
phone (716) 856-9600

SYBYL

Tripos Associates
1699 S. Hanley Road, Suite 303
St. Louis, MO 63144-2913
phone (800) 323-2960

The UCSF MIDASPLUS molecular display program

MIDAS Software Distribution
Computer Graphics Laboratory
School of Pharmacy
University of California
San Francisco, CA 94143-0446

More resources

The NIH Molecular Modeling Home Page
http://www.nih.gov/molecular_modeling/mmhome.html

SYBYL

Tripos Associates
1699 S. Hanley Road, Suite 303
St. Louis, MO 63144-2913
phone (800) 323-2960

The UCSF MIDASPLUS molecular display program

MIDAS Software Distribution
Computer Graphics Laboratory
School of Pharmacy
University of California
San Francisco, CA 94143-0446

More resources

The NIH Molecular Modeling Home Page
http://www.nih.gov/molecular_modeling/mmhome.html

```
*          UU/    UU/  CC/    CC/  SS/    SS/  FF/   FF\          *
*        UUUUUUUUU/    CCCCCCC/    SSSSSSS/   FF/     FF\         *
*                                                                 *
*                                                                 *
*      Copyright (C) 1997 Regents of the University of California *
*                     All Rights Reserved.                        *
*                                                                 *
*    This program implements the docking algorithm of I.D. Kuntz, *
*    J Mol Biol 161, 269-288, 1982.                               *
*                                                                 *
*    Versions 1.0 and 1.1 of the dock program were based extensively on *
*    the work of Robert Sheridan, Renee DesJarlais, and Tack Kuntz. *
*                                                                 *
*    Version 2.0, macrodock, is based on the work of Brian Shoichet *
*    and Tack Kuntz, with help from Dale Bodian.                  *
*                                                                 *
*    Version 3.0, chemdock, is based on the work of Elaine Meng, Brian *
*    Shoichet, and Tack Kuntz.                                    *
*                                                                 *
*    Version 3.5 was produced from version 3.0 by Mike Connolly and *
*    Dan Gschwend.                                                *
*                                                                 *
*    Version 4.0 is based on the work of Todd Ewing and Tack Kuntz. *
*                                                                 *
*                                                                 *
\*****************************************************************/

#include <time.h>
#include "define.h"
#include "global.h"
#include "utility.h"
#include "mol.h"
#include "search.h"
#include "dock.h"
#include "label.h"
#include "screen.h"
#include "score.h"
#include "score_dock.h"
#include "flex.h"
#include "match.h"
#include "orient.h"
#include "io.h"
#include "io_grid.h"
#include "io_ligand.h"
#include "mol_prep.h"
#include "parm_dock.h"
#include "rank.h"

GLOBAL global = {0};            /* See global.h for global variables */

main (int argc, char *argv[])
{
/*
* General variables (see header files for globally defined variables
* 6/95 te
*/
  int i;                        /* Counter variables */
  float time = 0.0;             /* Total amount of elapsed cpu time */
  int write_flag;               /* Flag to write molecule when scored */

/*
* Variables for multiple ligands mode
* 6/95 te
*/
  int lig_flag;                 /* Flag for whether a ligand was read */
  int lig_total = 0;            /* Number of ligands read */
  int lig_proc = 0;             /* Number of ligands processed */
  int lig_skip = 0;             /* Number of ligands skipped */

/*
* File pointers used when checking for the presence of control files
* 6/95 te
*/
  FILE *dump = NULL;
  FILE *quit = NULL;

/*
* Data structures containing docking parameters
* 2/96 te
*/
  DOCK dock = {0};              /* Docking data structure */
  ORIENT orient = {0};          /* Orientation data structure */
  SCORE score = {0};            /* Scoring data structure */
  LABEL label = {0};            /* Labeling data structure */

  LIST best_anchors = {0};      /* List of best anchor orientations */
  LIST best_orients = {0};      /* List of best orientations */
  LIST best_ligands = {0};      /* List of best molecules */

  MOLECULE mol_ref = {0};       /* Reference ligand data (from file) */
  MOLECULE mol_init = {0};      /* Initial ligand data */
  MOLECULE mol_conf = {0};      /* Ligand conformation data */
  MOLECULE mol_ori = {0};       /* Oriented ligand data */
  MOLECULE mol_score = {0};     /* Minimized ligand data */
  MOLECULE mol_out = {0};       /* Output ligand data */

/*
* Functions used in the main routine that also reside in this file.
* Other functions are declared in the header files.
* 6/95 te
*/
  void write_program_header (void);
  void set_memory_limit (void);
  float elapsed_time (float *);
  void initialize_performance (DOCK *, SCORE *);
  void report_performance (DOCK *, SCORE *, int);

/*
* Process command line arguments
* 6/95 te
*/
  process_commands (&dock, argc, argv);

/*
* Output program header, set memory ceiling, and initialize clock
* 6/95 te
*/
  write_program_header ();
  set_memory_limit ();

  if (dock.performance_flag)
    initialize_performance (&dock, &score);

  dock.total_time = -elapsed_time (NULL);

/*
* Read in dock parameters
* 6/95 te
*/
  get_parameters (&dock, &orient, &score, &label);

/*
* Initialize best molecule lists
* 12/96 te
*/
  allocate_lists (&score, &best_anchors, dock.rank_anchor_total, FALSE);
  allocate_lists (&score, &best_orients, dock.rank_orient_total, TRUE);
  allocate_lists (&score, &best_ligands, dock.rank_ligand_total, TRUE);

/*
* Open ligand input files
* 6/95 te
*/
  if (!dock.parallel.flag || dock.parallel.server)
    dock.ligand_file = efopen (dock.ligand_file_name, "r", global.outfile);

/*
* Open ligand output files
* 6/95 te
*/
  if (!dock.rank_ligands && !dock.parallel.server)
    for (i = 0; i < SCORE_TOTAL; i++)
      if (score.type[i].flag)
        score.type[i].file =
          efopen (score.type[i].file_name, "w", global.outfile);

  write_flag =
    (dock.write_orients && !dock.rank_orients) ||
    (!dock.multiple_orients && dock.multiple_ligands && !dock.rank_ligands);

/*
* If a restart has been signalled, then read in previous data
* 11/96 te
*/
  if (dock.restart)
  {
    dock.restart = FALSE;
    fprintf (global.outfile,
      "Reading restart information from disk.\n");

    if (read_restartinfo
    (
      &dock,
      &score,
      &best_ligands,
      &lig_total,
      &lig_proc,
      &lig_skip,
      &time
    ))
    {
/*
*     Reset time to what it was in previous run
*     11/95 te
*/
      elapsed_time (&time);
    }

    else
      fprintf (global.outfile,
        "WARNING main: Restart information not found."
        " Normal docking to be performed.\n");

    dock.total_time =
      dock.read_time =
      -elapsed_time (NULL);
  }

/*
* Otherwise, skip initial set of molecules
* 11/96 te
*/
  else
  {
    dock.read_time -= elapsed_time (NULL);

    for
    (
      i = 0;
      (i < dock.initial_skip) &&
        (get_ligand
        (
          &dock, &score, &label,
          &mol_ref, &mol_init,
          FALSE, FALSE, FALSE
        ) != EOF);
      lig_total = lig_skip = ++i
    );
  }

/*
* Loop over all ligands
* 6/95 te
*/
  for (;
    (!lig_total ||
      (dock.multiple_ligands && (lig_total < dock.max_ligands))) &&
    ((lig_flag = get_ligand
      (
        &dock, &score, &label,
        &mol_ref, &mol_init,
        score.flag || label.flex.flag,
        label.chemical.flag,
        label.vdw.flag
      )) != EOF);
    lig_total++)
  {
/*
*   Check to see if any ligands were read from input file
*   8/95 te
*/
    if (lig_flag == TRUE)
    {
      if (global.output_volume == 'v')
      {
        fprintf (global.outfile, "Processing %s\n", mol_ref.info.name);
        fflush (global.outfile);
      }

/*
*     Perform chemical screening functions
*     11/96 te
*/
      if (label.chemical.screen.process_flag)
      {
        dock.screen_time -= elapsed_time (NULL);

        if (check_screen (&orient.match, &label, &mol_ref, lig_proc))
```

```
*          UU/    UU/  CC/    CC/  SS/    SS/  FF/    FF\                   *
*         UUUUUUUUU/    CCCCCCC/    SSSSSSS/   FF/     FF\                  *
*                                                                           *
*                                                                           *
*      Copyright (C) 1997 Regents of the University of California           *
*                         All Rights Reserved.                              *
*                                                                           *
*     This program implements the docking algorithm of I.D. Kuntz,          *
*     J Mol Biol 161, 269-288, 1982.                                        *
*                                                                           *
*     Versions 1.0 and 1.1 of the dock program were based extensively on    *
*     the work of Robert Sheridan, Renee DesJarlais, and Tack Kuntz.        *
*                                                                           *
*     Version 2.0, macrodock, is based on the work of Brian Shoichet        *
*     and Tack Kuntz, with help from Dale Bodian.                           *
*                                                                           *
*     Version 3.0, chemdock, is based on the work of Elaine Meng, Brian     *
*     Shoichet, and Tack Kuntz.                                             *
*                                                                           *
*     Version 3.5 was produced from version 3.0 by Mike Connolly and        *
*     Dan Gschwend.                                                         *
*                                                                           *
*     Version 4.0 is based on the work of Todd Ewing and Tack Kuntz.        *
*                                                                           *
*                                                                           *
\***************************************************************************/

#include <time.h>
#include "define.h"
#include "global.h"
#include "utility.h"
#include "mol.h"
#include "search.h"
#include "dock.h"
#include "label.h"
#include "screen.h"
#include "score.h"
#include "score_dock.h"
#include "flex.h"
#include "match.h"
#include "orient.h"
#include "io.h"
#include "io_grid.h"
#include "io_ligand.h"
#include "mol_prep.h"
#include "parm_dock.h"
#include "rank.h"

GLOBAL global = {0};            /* See global.h for global variables */

main (int argc, char *argv[])
{
/*
* General variables (see header files for globally defined variables
* 6/95 te
*/
  int i;                        /* Counter variables */
  float time = 0.0;             /* Total amount of elapsed cpu time */
  int write_flag;               /* Flag to write molecule when scored */

/*
* Variables for multiple ligands mode
* 6/95 te
*/
  int lig_flag;                 /* Flag for whether a ligand was read */
  int lig_total = 0;            /* Number of ligands read */
  int lig_proc = 0;             /* Number of ligands processed */
  int lig_skip = 0;             /* Number of ligands skipped */

/*
* File pointers used when checking for the presence of control files
* 6/95 te
*/
  FILE *dump = NULL;
  FILE *quit = NULL;

/*
* Data structures containing docking parameters
* 2/96 te
*/
  DOCK dock = {0};              /* Docking data structure */
  ORIENT orient = {0};          /* Orientation data structure */
  SCORE score = {0};            /* Scoring data structure */
  LABEL label = {0};            /* Labeling data structure */

  LIST best_anchors = {0};      /* List of best anchor orientations */
  LIST best_orients = {0};      /* List of best orientations */
  LIST best_ligands = {0};      /* List of best molecules */

  MOLECULE mol_ref = {0};       /* Reference ligand data (from file) */
  MOLECULE mol_init = {0};      /* Initial ligand data */
  MOLECULE mol_conf = {0};      /* Ligand conformation data */
  MOLECULE mol_ori = {0};       /* Oriented ligand data */
  MOLECULE mol_score = {0};     /* Minimized ligand data */
  MOLECULE mol_out = {0};       /* Output ligand data */

/*
* Functions used in the main routine that also reside in this file.
* Other functions are declared in the header files.
* 6/95 te
*/
  void write_program_header (void);
  void set_memory_limit (void);
  float elapsed_time (float *);
  void initialize_performance (DOCK *, SCORE *);
  void report_performance (DOCK *, SCORE *, int);

/*
* Process command line arguments
* 6/95 te
*/
  process_commands (&dock, argc, argv);

/*
* Output program header, set memory ceiling, and initialize clock
* 6/95 te
*/
  write_program_header ();
  set_memory_limit ();

  if (dock.performance_flag)
    initialize_performance (&dock, &score);

  dock.total_time = -elapsed_time (NULL);

/*
* Read in dock parameters
* 6/95 te
*/
  get_parameters (&dock, &orient, &score, &label);

/*
* Initialize best molecule lists
* 12/96 te
*/
  allocate_lists (&score, &best_anchors, dock.rank_anchor_total, FALSE);
  allocate_lists (&score, &best_orients, dock.rank_orient_total, TRUE);
  allocate_lists (&score, &best_ligands, dock.rank_ligand_total, TRUE);

/*
* Open ligand input files
* 6/95 te
*/
  if (!dock.parallel.flag || dock.parallel.server)
    dock.ligand_file = efopen (dock.ligand_file_name, "r", global.outfile);

/*
* Open ligand output files
* 6/95 te
*/
  if (!dock.rank_ligands && !dock.parallel.server)
    for (i = 0; i < SCORE_TOTAL; i++)
      if (score.type[i].flag)
        score.type[i].file =
          efopen (score.type[i].file_name, "w", global.outfile);

  write_flag =
    (dock.write_orients && !dock.rank_orients) ||
    (!dock.multiple_orients && dock.multiple_ligands && !dock.rank_ligands);

/*
* If a restart has been signalled, then read in previous data
* 11/96 te
*/
  if (dock.restart)
  {
    dock.restart = FALSE;
    fprintf (global.outfile,
      "Reading restart information from disk.\n");

    if (read_restartinfo
    (
      &dock,
      &score,
      &best_ligands,
      &lig_total,
      &lig_proc,
      &lig_skip,
      &time
    ))
    {
/*
*     Reset time to what it was in previous run
*     11/95 te
*/
      elapsed_time (&time);
    }

    else
      fprintf (global.outfile,
        "WARNING main: Restart information not found."
        " Normal docking to be performed.\n");

    dock.total_time =
      dock.read_time =
      -elapsed_time (NULL);
  }

/*
* Otherwise, skip initial set of molecules
* 11/96 te
*/
  else
  {
    dock.read_time -= elapsed_time (NULL);

    for
    (
      i = 0;
      (i < dock.initial_skip) &&
        (get_ligand
        (
          &dock, &score, &label,
          &mol_ref, &mol_init,
          FALSE, FALSE, FALSE
        ) != EOF);
      lig_total = lig_skip = ++i
    );
  }

/*
* Loop over all ligands
* 6/95 te
*/
  for (;
    (!lig_total ||
      (dock.multiple_ligands && (lig_total < dock.max_ligands))) &&
    ((lig_flag = get_ligand
      (
        &dock, &score, &label,
        &mol_ref, &mol_init,
        score.flag || label.flex.flag,
        label.chemical.flag,
        label.vdw.flag
      )) != EOF);
    lig_total++)
  {
/*
*   Check to see if any ligands were read from input file
*   8/95 te
*/
    if (lig_flag == TRUE)
    {
      if (global.output_volume == 'v')
      {
        fprintf (global.outfile, "Processing %s\n", mol_ref.info.name);
        fflush (global.outfile);
      }

/*
*     Perform chemical screening functions
*     11/96 te
*/
      if (label.chemical.screen.process_flag)
      {
        dock.screen_time -= elapsed_time (NULL);

        if (check_screen (&orient.match, &label, &mol_ref, lig_proc))
```

```
        {
          fprintf (global.outfile, ".");

          if (lig_proc % 50 == 0)
            fprintf (global.outfile, "| %d mols, %.4g secs\n",
              lig_proc, elapsed_time (NULL));

          fflush (global.outfile);
        }

/*
*       Write/store this molecule if its orientations weren't written out yet.
*       11/96 te
*/
        if (!write_flag)
        {
          if (score.flag)
          {
/*
*           Either store the best orientation(s)
*           3/96 te
*/
            if (dock.rank_ligands)
            {
              inter_lists (&score, &best_orients, &mol_init);
              merge_lists (&score, &best_ligands, &best_orients);
            }

/*
*           Or write the best orientation(s) out to a file
*           3/96 te
*/
            else
              write_topscorers
              (
                &dock,
                &score,
                &best_orients,
                &mol_ref,
                &mol_out
              );
          }

          else if (label.chemical.screen.construct_flag)
            write_ligand
            (
              &dock,
              &score,
              &mol_conf,
              score.type[NONE].file_name,
              score.type[NONE].file
            );
        }

/*
*       Perform database processing functions
*       1/97 te
*/
        if (dock.multiple_ligands)
        {
/*
*         Output list of current top scorers
*         6/95 te
*/
          if (score.flag)
            write_info
            (
              &dock,
              &score,
              &best_ligands,
              lig_total + 1,
              lig_proc,
              lig_skip,
              elapsed_time (NULL)
            );

/*
*         Check to see if this job has been requested to shut down
*         6/95 te
*/
          if
          (
            (dock.rank_ligands || dock.parallel.flag) &&
            (quit = fopen (dock.quit_file_name, "r"))
          )
          {
            fclose (quit);
            if (remove (dock.quit_file_name))
              fprintf (global.outfile,
                "WARNING main: Unable to delete %s.\n", dock.quit_file_name);
          }

/*
*         Check for a request to write current results
*         6/95 te
*/
          if (dock.rank_ligands)
          {
            if (dump = fopen (dock.dump_file_name, "r"))
            {
              fclose (dump);
              if (remove (dock.dump_file_name))
                fprintf (global.outfile,
                  "WARNING main: Unable to delete %s.\n", dock.dump_file_name);
            }

/*
*           Write restart information to disk
*           6/95 te
*/
            if (dump || quit || (lig_proc % dock.restart_interval == 0))
            {
              fprintf (global.outfile,
                "ATTENTION main: Writing restart information to disk.\n");

              if (!write_restartinfo
                (
                  &dock,
                  &score,
                  &best_ligands,
                  lig_total + 1,
                  lig_proc,
                  lig_skip,
                  elapsed_time (NULL)
                ))
                fprintf (global.outfile,
                  "WARNING main: Unable to write restart information.\n");
            }
```

```
            if (dump || quit)
            {
              fprintf (global.outfile,
                "ATTENTION main: Writing current top scorers as requested.\n");

              write_topscorers
              (
                &dock,
                &score,
                &best_ligands,
                &mol_ref,
                &mol_out
              );

              for (i = 0; i < SCORE_TOTAL; i++)
                score.type[i].number_written = 0;
            }

            dump = NULL;
          }

          if (quit)
          {
            fprintf (global.outfile,
              "ATTENTION main: Execution terminating as requested.\n");
            fclose (dock.ligand_file);

/*
*           Shut down parallel client processes, also, if this is a server run
*           10/95 te
*/
            if (dock.parallel.server)
            {
              for (i = 0; i < dock.parallel.client_total; i++)
              {
                sprintf
                  (dock.quit_file_name, "%s.quit",
                  dock.parallel.client_name[i]);
                quit = rfopen (dock.quit_file_name, "w", global.outfile);
                efclose (&quit);
              }
            }

            efclose (&global.outfile);
            exit (EXIT_SUCCESS);
          }
        }
      }

/*
*     If no anchor was found, then record the skip
*     6/97 te
*/
      else
        lig_skip++;
    }

/*
*   If no ligand was read, then record the skip
*   6/97 te
*/
    else
      lig_skip++;

/*
*   Skip set of molecules for each interval
*   9/95 te
*/
    for (i = 0; i < dock.interval_skip; i++)
    {
      if (get_ligand
        (
          &dock, &score, &label,
          &mol_ref, &mol_init,
          FALSE, FALSE, FALSE
        ) != EOF)
      {
        lig_total++;
        lig_skip++;
      }

      else
        break;
    }
  }

  dock.read_time += elapsed_time (NULL);

/*
* Finished reading ligands in from disk, make sure at least one was read
* 6/95 te
*/
  if (lig_total == 0)
  {
    fprintf (global.outfile,
      "Unable to read anything from ligand coordinate file.\n");
    exit (EXIT_FAILURE);
  }


/*
* Shut down client processes, if this is a server run
* 10/95 te
*/
  if (dock.parallel.server)
  {
    for (i = 0; i < dock.parallel.client_total; i++)
    {
      sprintf (dock.quit_file_name, "%s.quit", dock.parallel.client_name[i]);
      quit = rfopen (dock.quit_file_name, "w", global.outfile);
      fclose (quit);
    }
  }

/*
* Write out final results (in rank_ligands mode)
* 6/95 te
*/
  if (dock.rank_ligands)
  {
    if (lig_proc % dock.restart_interval)
    {
      fprintf (global.outfile, "Writing restart information to disk.\n");

      if (!write_restartinfo
        (
          &dock,
          &score,
          &best_ligands,
```

```
        lig_total,
        lig_proc,
        lig_skip,
        elapsed_time (NULL)
      ))
        fprintf (global.outfile,
          "WARNING main: Unable to write restart information.\n");
    }

    fprintf (global.outfile, "Writing top scoring molecules to disk.\n");

    write_topscorers
    (
      &dock,
      &score,
      &best_ligands,
      &mol_ref,
      &mol_out
    );
  }

/*
* Close ligand output files
* 6/95 te
*/
  else
    for (i = 0; i < SCORE_TOTAL; i++)
      if (score.type[i].flag)
        fclose (score.type[i].file);

/*
* Close ligand input files
* 5/97 te
*/
  fclose (dock.ligand_file);

  if (check_file_extension (dock.ligand_file_name, FALSE) == Ptr)
    read_molecule (NULL, NULL, dock.ligand_file_name, NULL, 0);

/*
* Free molecule structures
* 5/97 te
*/
  free_molecule (&mol_ref);
  free_molecule (&mol_init);
  free_molecule (&mol_conf);
  free_molecule (&mol_ori);
  free_molecule (&mol_score);
  free_molecule (&mol_out);

/*
* Free top score lists
* 5/97 te
*/
  free_lists (&score, &best_anchors);
  free_lists (&score, &best_orients);
  free_lists (&score, &best_ligands);

/*
* Free all other arrays
* 5/97 te
*/
  if (label.chemical.screen.flag)
    free_screen (&orient.match, &label);

  if (orient.flag)
    free_orients (&label, &orient);

  free_scores (&label, &score);

  free_labels (&label);

/*
* Report performance
* 1/97 te
*/
  dock.total_time += elapsed_time (NULL);

  if (dock.performance_flag)
    report_performance (&dock, &score, lig_proc);

  fprintf (global.outfile,
    "\nFinished processing molecule%s in %.6g seconds.\n",
    lig_total > 1 ? "s" : "", dock.total_time);

  fclose (global.outfile);

  return (EXIT_SUCCESS);
}

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

Evaluate how much time has elapsed (with checking for wrap-around)
6/95 te

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

float elapsed_time (float *reset_value)
{
  static long clock_previous = 0;
  static long clock_current = 0;
  static float time;

  if (reset_value)
    time = *reset_value;

  clock_previous = clock_current;
  clock_current = clock ();

  if (clock_current < clock_previous)
  {
    time +=
      ((float) (LONG_MAX - clock_previous))
      / ((float) CLOCKS_PER_SEC);

    time +=
      ((float) (clock_current - LONG_MIN))
      / ((float) CLOCKS_PER_SEC);
  }

  else
    time +=
      ((float) (clock_current - clock_previous))
      / ((float) CLOCKS_PER_SEC);

  return time;
}
```

```
/* /////////////////////////////////////////////////////////////////// */

void write_program_header (void)
{
  if (global.outfile != stdout)
    fprintf (global.outfile, "\n"
      "      UUUUUUUUU     CCCCCCC     SSSSSSS    FF/   FFF/ \n"
      "      UU/    UU/  CC/    CC/  SS/    SS/  FF/ FFF/    \n"
      "     UU/    UU/  CC/    CC/  SS/         FFFFF/       \n"
      "    UU/    UU/  CC/    CC/  SS/         FF/ FF\\       \n"
      "   UU/    UU/  CC/    CC/  SS/    SS/  FF/   FF\\      \n"
      " UUUUUUUUU/    CCCCCCC/    SSSSSSS/   FF/     FF\\     \n\n\n");

  else
    fprintf (global.outfile, "\n\n"
      "      UUU?[4mUUUU?[0mUU     C?[4mCCCC?[0mCC     S?[4mSSSS?[0mSS    FF/   FFF/ \n"
      "      UU/    UU/  CC/    CC/  SS/    SS/  FF/ FFF/    \n"
      "     UU/    UU/  CC/    CC/  SS/         FFFFF/       \n"
      "    UU/    UU/  CC/    CC/  SS/         FF/ FF\\       \n"
      "   UU/    UU/  CC/    CC/  SS/    SS/  FF/   FF\\      \n"
      " ?[4mUUUUUUUUU?[0m/    ?[4mCCCCCCC?[0m/    ?[4mSSSSSSS?[0m/   ?[4mFF?[0m/
?[4mFF\\?[0m     \n\n\n");

  fprintf (global.outfile,
    "University of California at San Francisco, DOCK %s\n", DOCK_VERSION);

  fflush (global.outfile);
}

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

void initialize_performance (DOCK *dock, SCORE *score)
{
  dock->total_time = 0;
  dock->read_time = 0;
  dock->screen_time = 0;
  dock->conform_time = 0;
  dock->periph_time = 0;
  dock->orient_time = 0;
  dock->score_time = 0;

  score->time = 0;
  score->minimize.call_total = 0;
  score->minimize.call_sub_total = 0;
  score->minimize.call_min = INT_MAX;
  score->minimize.call_max = INT_MIN;

  score->minimize.vertex_total = 0;
  score->minimize.vertex_min = INT_MAX;
  score->minimize.vertex_max = INT_MIN;

  score->minimize.iteration_total = 0;
  score->minimize.iteration_min = INT_MAX;
  score->minimize.iteration_max = INT_MIN;

  score->minimize.cycle_total = 0;
  score->minimize.cycle_min = INT_MAX;
  score->minimize.cycle_max = INT_MIN;

  score->minimize.delta_total = 0;
  score->minimize.delta_min = FLT_MAX;
  score->minimize.delta_max = FLT_MIN;
}

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

void report_performance (DOCK *dock, SCORE *score, int mol_total)
{
  fprintf (global.outfile,
    "\n____________________Docking_Performance____________________\n");

  dock->other_time = dock->total_time - dock->read_time;
  dock->read_time -= dock->screen_time + dock->conform_time;
  dock->conform_time -= dock->orient_time;
  dock->orient_time -= dock->score_time;

  fprintf
  (
    global.outfile,
    "\n%-45s %8s %8s\n",
    "Procedure timings",
    "time (s)",
    "percent"
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Read",
    dock->read_time,
    dock->read_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Screen",
    dock->screen_time,
    dock->screen_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Orientation Search",
    dock->orient_time,
    dock->orient_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Orientation Score",
    dock->score_time,
    dock->score_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Conformation Anchor",
```

```
          lig_total,
          lig_proc,
          lig_skip,
          elapsed_time (NULL)
        ))
          fprintf (global.outfile,
            "WARNING main: Unable to write restart information.\n");
    }

    fprintf (global.outfile, "Writing top scoring molecules to disk.\n");

    write_topscorers
    (
      &dock,
      &score,
      &best_ligands,
      &mol_ref,
      &mol_out
    );
  }

/*
* Close ligand output files
* 6/95 te
*/
  else
    for (i = 0; i < SCORE_TOTAL; i++)
      if (score.type[i].flag)
        fclose (score.type[i].file);

/*
* Close ligand input files
* 5/97 te
*/
  fclose (dock.ligand_file);

  if (check_file_extension (dock.ligand_file_name, FALSE) == Ptr)
    read_molecule (NULL, NULL, dock.ligand_file_name, NULL, 0);

/*
* Free molecule structures
* 5/97 te
*/
  free_molecule (&mol_ref);
  free_molecule (&mol_init);
  free_molecule (&mol_conf);
  free_molecule (&mol_ori);
  free_molecule (&mol_score);
  free_molecule (&mol_out);

/*
* Free top score lists
* 5/97 te
*/
  free_lists (&score, &best_anchors);
  free_lists (&score, &best_orients);
  free_lists (&score, &best_ligands);

/*
* Free all other arrays
* 5/97 te
*/
  if (label.chemical.screen.flag)
    free_screen (&orient.match, &label);

  if (orient.flag)
    free_orients (&label, &orient);

  free_scores (&label, &score);

  free_labels (&label);

/*
* Report performance
* 1/97 te
*/
  dock.total_time += elapsed_time (NULL);

  if (dock.performance_flag)
    report_performance (&dock, &score, lig_proc);

  fprintf (global.outfile,
    "\nFinished processing molecule%s in %.6g seconds.\n",
    lig_total > 1 ? "s" : "", dock.total_time);

  fclose (global.outfile);

  return (EXIT_SUCCESS);
}


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

Evaluate how much time has elapsed (with checking for wrap-around)
6/95 te

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

float elapsed_time (float *reset_value)
{
  static long clock_previous = 0;
  static long clock_current = 0;
  static float time;

  if (reset_value)
    time = *reset_value;

  clock_previous = clock_current;
  clock_current = clock ();

  if (clock_current < clock_previous)
  {
    time +=
      ((float) (LONG_MAX - clock_previous))
      / ((float) CLOCKS_PER_SEC);

    time +=
      ((float) (clock_current - LONG_MIN))
      / ((float) CLOCKS_PER_SEC);
  }

  else
    time +=
      ((float) (clock_current - clock_previous))
      / ((float) CLOCKS_PER_SEC);

  return time;
}
```

```
/* //////////////////////////////////////////////////////////////////// */

void write_program_header (void)
{
  if (global.outfile != stdout)
    fprintf (global.outfile, "\n"
      "       UUUUUUUUU      CCCCCCC      SSSSSSS     FF/    FFF/ \n"
      "       UU/     UU/  CC/     CC/  SS/     SS/  FF/ FFF/     \n"
      "      UU/     UU/  CC/     CC/  SS/           FFFFF/       \n"
      "     UU/     UU/  CC/     CC/  SS/           FF/ FF\\       \n"
      "    UU/     UU/  CC/     CC/  SS/     SS/  FF/    FF\\      \n"
      "  UUUUUUUUU/     CCCCCCC/     SSSSSSS/    FF/      FF\\     \n\n\n");

  else
    fprintf (global.outfile, "\n\n"
      "       UUU?[4mUUUU?[0mUU      C?[4mCCCC?[0mCC      S?[4mSSSS?[0mSS     FF/    FFF/ \n"
      "       UU/     UU/  CC/     CC/  SS/     SS/  FF/ FFF/     \n"
      "      UU/     UU/  CC/     CC/  SS/           FFFFF/       \n"
      "     UU/     UU/  CC/     CC/  SS/           FF/ FF\\       \n"
      "    UU/     UU/  CC/     CC/  SS/     SS/  FF/    FF\\      \n"
      "  ?[4mUUUUUUUUU?[0m/    ?[4mCCCCCCC?[0m/    ?[4mSSSSSSS?[0m/   ?[4mFF?[0m/
?[4mFF\\?[0m      \n\n\n");

  fprintf (global.outfile,
    "University of California at San Francisco, DOCK %s\n", DOCK_VERSION);

  fflush (global.outfile);
}


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

void initialize_performance (DOCK *dock, SCORE *score)
{
  dock->total_time = 0;
  dock->read_time = 0;
  dock->screen_time = 0;
  dock->conform_time = 0;
  dock->periph_time = 0;
  dock->orient_time = 0;
  dock->score_time = 0;

  score->time = 0;
  score->minimize.call_total = 0;
  score->minimize.call_sub_total = 0;
  score->minimize.call_min = INT_MAX;
  score->minimize.call_max = INT_MIN;

  score->minimize.vertex_total = 0;
  score->minimize.vertex_min = INT_MAX;
  score->minimize.vertex_max = INT_MIN;

  score->minimize.iteration_total = 0;
  score->minimize.iteration_min = INT_MAX;
  score->minimize.iteration_max = INT_MIN;

  score->minimize.cycle_total = 0;
  score->minimize.cycle_min = INT_MAX;
  score->minimize.cycle_max = INT_MIN;

  score->minimize.delta_total = 0;
  score->minimize.delta_min = FLT_MAX;
  score->minimize.delta_max = FLT_MIN;
}


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

void report_performance (DOCK *dock, SCORE *score, int mol_total)
{
  fprintf (global.outfile,
    "\n______________________Docking_Performance______________________\n");

  dock->other_time = dock->total_time - dock->read_time;
  dock->read_time -= dock->screen_time + dock->conform_time;
  dock->conform_time -= dock->orient_time;
  dock->orient_time -= dock->score_time;

  fprintf
  (
    global.outfile,
    "\n%-45s %8s %8s\n",
    "Procedure timings",
    "time (s)",
    "percent"
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Read",
    dock->read_time,
    dock->read_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Screen",
    dock->screen_time,
    dock->screen_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Orientation Search",
    dock->orient_time,
    dock->orient_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Orientation Score",
    dock->score_time,
    dock->score_time / dock->total_time * 100.0
  );

  fprintf
  (
    global.outfile,
    "%-45s %8.2f %8.0f\n",
    "Conformation Anchor",
```

```
int get_layer_conformation
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           layer,
  int           conformer
)
{
  int segment_id;
  int segment;

/*
* If the total conformations is outside the limit, then:
* 1. Assign a random search seed to each segment;
* 2. Initialize the starting segment as always the first
* 3/97 te
*/
  if (molecule->layer[layer].conform_total > label->flex.max_conforms)
  {
    if (conformer >= label->flex.max_conforms)
      return EOF;

    for
    (
      segment_id = 0;
      segment_id < molecule->layer[layer].segment_total;
      segment_id++
    )
    {
      segment = molecule->layer[layer].segment[segment_id];
      molecule->segment[segment].conform_seed = rand ();
      molecule->segment[segment].conform_count = 0;
    }

    segment_id = 0;
  }

/*
* Otherwise, initialize the starting segment as the first or the last
* 3/97 te
*/
  else
  {
    if (conformer == 0)
    {
      segment_id = 0;
      segment = molecule->layer[layer].segment[segment_id];
      molecule->segment[segment].conform_count = 0;
    }

    else
      segment_id = molecule->layer[layer].segment_total - 1;
  }


/*
* Loop until a complete layer conformation is found, or no more possible
* 3/97 te
*/
  for (;;)
  {
    if
    (
      get_segment_conformation
      (
        label,
        score,
        molecule,
        layer,
        segment_id
      ) == TRUE
    )
    {
      if (++segment_id >= molecule->layer[layer].segment_total)
        return TRUE;

      segment = molecule->layer[layer].segment[segment_id];
      molecule->segment[segment].conform_count = 0;
    }

    else if (--segment_id < 0)
      return EOF;
  }
}


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

int get_segment_conformation
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           layer,
  int           segment_id
)
{
  int segment;

  segment = molecule->layer[layer].segment[segment_id];

  if (label->flex.drive_flag == TRUE)
  {
    for (;;)
    {
      if
      (
        update_segment_torsions
        (
          &label->flex,
          molecule,
          segment,
          molecule->segment[segment].conform_count++
        ) != TRUE
      )
        return EOF;

      if (check_segment_conformation
        (label, score, molecule, layer, segment_id)
        == TRUE)
        return TRUE;
    }
  }

  else
  {
    if (molecule->segment[segment].conform_count++ == 0)
      return TRUE;

    else
      return EOF;
  }
}


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

int update_segment_torsions
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule,
  int           segment,
  int           conformer
)
{
  int torsion;
  int flex_id;
  int index;

/*
* Assign new torsion angle and apply it
* 3/97 te
*/
  torsion = molecule->segment[segment].torsion_id;

  if (torsion == NEITHER)
    return (conformer == 0) ? TRUE : EOF;

  flex_id = molecule->torsion[torsion].flex_id;

  if (conformer >= label_flex->member[flex_id].torsion_total)
    return EOF;

  else
  {
    index = (molecule->segment[segment].conform_seed + conformer) %
      label_flex->member[flex_id].torsion_total;

    molecule->torsion[torsion].target_angle =
      label_flex->member[flex_id].torsion[index] / 180.0 * PI;

    molecule->transform.tors_flag = TRUE;

    torsion_transform (molecule, torsion);
    return TRUE;
  }
}


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

Routine to check all inter-segment distances to update distance matrix
and check for clashes

Return values:
        TRUE    segment conformation ok
        FALSE   clash detected
1/97 te

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

int check_segment_conformation
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           li,                     /* Input layer */
  int           sli                     /* Input layer segment_id */
)
{
  int lj;                       /* Comparison layer */
  int slj;                      /* Segment id in layer list */
  int si, sj;                   /* Segment id */
  int asi, asj;                 /* Atom id in segment list */
  int ai, aj;                   /* Atom id */
  float distance;               /* Square of distance between atoms */
  float reference;              /* Cutoff distance allowed */


/*
* Update atom connection info
* 3/97 te
*/
  initialize_near (&score->near, molecule);

  si = molecule->layer[li].segment[sli];

/*
* Loop through atoms in si segment
* 1/97 te
*/
  for (asi = molecule->segment[si].atom_total - 1; asi >= 0; asi--)
  {
    ai = molecule->segment[si].atom[asi];

/*
*   Loop from current layer to all inner layers
*   1/97 te
*/
    for (lj = li; lj >= 0; lj--)
    {
/*
*     Loop through segments in layer
*       For current layer, only consider previous segments
*       For inner layers, consider all segments
*     1/97 te
*/
      for
      (
        slj = (li == lj ? sli - 1 : molecule->layer[lj].segment_total - 1);
        slj >= 0;
        slj--
      )
      {
        sj = molecule->layer[lj].segment[slj];

/*
*       Loop through atoms in current segment
*       1. Check if unflagged atoms clash with atom ai
*       2. Update portions of distance matrix that have changed
*       1/97 te
*/
        for (asj = molecule->segment[sj].atom_total - 1; asj >= 0; asj--)
        {
          aj = molecule->segment[sj].atom[asj];

          distance = square_distance
```

```
            (molecule->coord[ai], molecule->coord[aj]);

          if (!score->near.flag[ai][aj])
          {
            reference = label->flex.clash_overlap *
              (label->vdw.member[molecule->atom[ai].vdw_id].radius +
              label->vdw.member[molecule->atom[aj].vdw_id].radius);

            if (distance < SQR (reference))
              return FALSE;
          }
        } /* End of aaj loop */
      } /* End of alj loop */
    } /* End of lj loop */
  } /* End of ali loop */

  return TRUE;
}


###################################################################################
##################################### GRID.H ######################################
###################################################################################
/*                                                                              */
/*                             Copyright UCSF, 1997                             */
/*                                                                              */

/*
Written by Todd Ewing
9/96
*/

typedef struct grid_struct
{
  int output_molecule;          /* Flag for whether to write receptor */

  int visualize;                /* Write out grids in MIDAS format */

  XYZ box_com;                  /* Center of mass of box enclosing grids */
  XYZ box_dimension;            /* Dimensions of box enclosing grids */

  FILE_NAME in_file_name;       /* Input receptor file name */
  FILE_NAME out_file_name;      /* Output receptor file name */

  FILE *in_file;                /* Input receptor file pointer */
  FILE *out_file;               /* Output receptor file pointer */

  FILE_NAME box_file_name;      /* Grid-enclosing box file */

} GRID;

###################################################################################
##################################### GRID.C ######################################
###################################################################################
/*                                                                              */
/*                             Copyright UCSF, 1997                             */
/*                                                                              */

/*
Written by Todd Ewing
9/96
*/
#include <time.h>
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "label.h"
#include "score.h"
#include "io.h"
#include "io_receptor.h"
#include "io_grid.h"
#include "grid.h"
#include "score_grid.h"
#include "parm_grid.h"

GLOBAL global = {0};

main (int argc, char *argv[])
{
/*
* General variables (see header files for globally defined variables
* 10/95 te
*/
  int i;
  GRID grid = {0};
  SCORE_GRID score_grid = {0};
  SCORE_BUMP score_bump = {0};
  SCORE_CONTACT score_contact = {0};
  SCORE_CHEMICAL score_chemical = {0};
  SCORE_ENERGY score_energy = {0};
  LABEL label = {0};
  MOLECULE receptor;

/*
* Functions used in the main routine
* 10/95 te
*/
  void write_program_header (void);
  void set_memory_limit     (void);

/*
* Process command line arguments
* 10/95 te
*/
  process_commands (argc, argv);

/*
* Output program header and user information
* 10/95 te
*/
  write_program_header ();

/*
* Set ceiling on how memory can be dynamically allocated for this run
* 11/95 te
*/
  set_memory_limit ();

/*
* Read in parameters from input file
* 10/95 te
*/
  if (!get_parameters
    (
      &grid,
      &score_grid,
      &score_bump,
      &score_contact,
      &score_chemical,
      &score_energy,
      &label
    ))
  {
    fprintf (stdout, "Error reading control parameters.\n");
    exit (EXIT_FAILURE);
  }

/*
* Read in vdw definitions
* 10/95 te
*/
  if (label.vdw.flag)
    get_vdw_labels (&label.vdw);

/*
* Read in label definitions
* 10/95 te
*/

  if (label.chemical.flag)
    get_chemical_labels (&label.chemical);

/*
* Read in receptor coordinates
* 10/95 te
*/
  fprintf (global.outfile, "\nReading in coordinates of receptor.\n");
  fflush (global.outfile);
  grid.in_file = efopen (grid.in_file_name, "r", global.outfile);

  if (read_receptor
    (&score_energy, &label, &receptor, grid.in_file_name, grid.in_file,
    label.vdw.flag && label.chemical.flag,
    label.chemical.flag, label.vdw.flag)
    != TRUE)
  {
    fprintf (global.outfile, "Error reading in receptor.\n");
    exit (EXIT_FAILURE);
  }

  fclose (grid.in_file);

/*
* Write out receptor coordinates
* 10/95 te
*/
  if (grid.output_molecule)
  {
    fprintf (global.outfile, "\nWriting out processed receptor.\n");
    fflush (global.outfile);
    grid.out_file = efopen (grid.out_file_name, "w", global.outfile);

    write_molecule
    (
      &receptor,
      grid.in_file_name,
      grid.out_file_name,
      grid.out_file
    );

    fclose (grid.out_file);
  }

  if (score_grid.flag)
  {
/*
*   Read in box coordinates which define the boundaries of the grid
*   10/95 te
*/
    fprintf (global.outfile, "\nReading in grid box information.\n");
    if (!read_box
      (grid.box_file_name, grid.box_com, grid.box_dimension))
    {
      fprintf (global.outfile, "Error reading box file %s.\n",
        grid.box_file_name);
      exit (EXIT_FAILURE);
    }


/*
*   Calculate grid variables
*   10/95 te
*/
    for (i = 0, score_grid.size = 1; i < 3; i++)
    {
      score_grid.origin[i] =
        grid.box_com[i] - grid.box_dimension[i] / 2.0;
      score_grid.span[i] =
        (int) (grid.box_dimension[i] / score_grid.spacing + 1.0) + 1;
      score_grid.size *= score_grid.span[i];
    }

    fprintf (global.outfile, "%-40s: %8d %8d %8d\n",
      "Number of grid points per side [x y z]",
      score_grid.span[0], score_grid.span[1], score_grid.span[2]);

    fprintf (global.outfile, "%-40s: %8d\n",
      "Total number of grid points", score_grid.size);
    fflush (global.outfile);

/*
*   Allocate memory for grids
*   10/95 te
*/
    if (score_bump.flag)
    {
      emalloc
      (
        (void **) &score_bump.grid,
        score_grid.size * sizeof (char),
        "chemgrid arrays",
        global.outfile
      );
      memset (score_bump.grid, -((char) 0), score_grid.size);
    }

    if (score_contact.flag)
      ecalloc
      (
        (void **) &score_contact.grid,
        score_grid.size,
        sizeof (short),
        "chemgrid arrays",
        global.outfile
      );

    if (score_chemical.flag)
    {
```

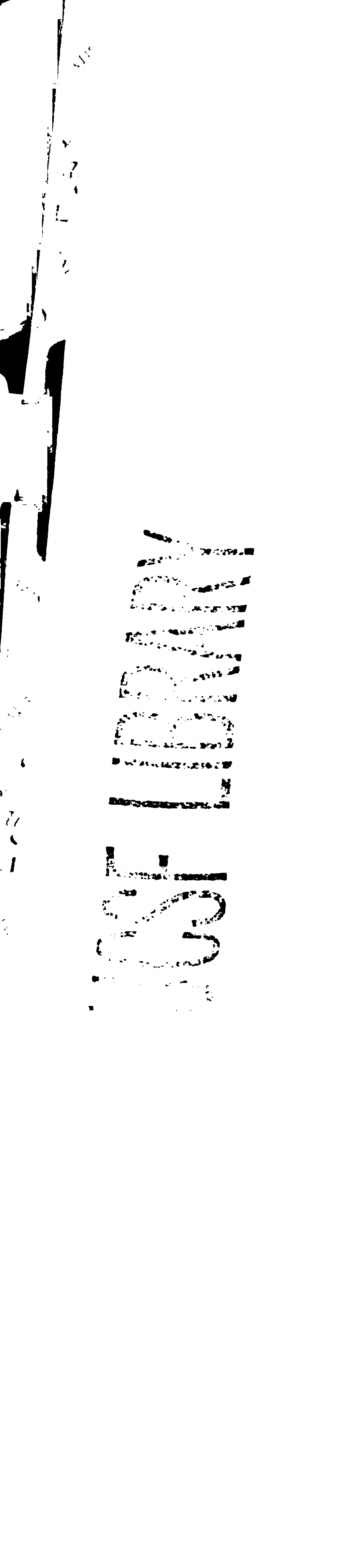
```
    ecalloc
    (
      (void **) &score_chemical.grid,
      label.chemical.total,
      sizeof (float *),
      "chemgrid arrays",
      global.outfile
    );

    for (i = 0; i < label.chemical.total; i++)
      ecalloc
      (
        (void **) &score_chemical.grid[i],
        score_grid size,
        sizeof (float),
        "chemgrid arrays",
        global.outfile
      );
  }

  if (score_energy.flag)
  {
    ecalloc
    (
      (void **) &score_energy.avdw,
      score_grid.size,
      sizeof (float),
      "chemgrid arrays",
      global.outfile
    );

    ecalloc
    (
      (void **) &score_energy.bvdw,
      score_grid size,
      sizeof (float),
      "chemgrid arrays",
      global.outfile
    );

    ecalloc
    (
      (void **) &score_energy.es,
      score_grid.size,
      sizeof (float),
      "chemgrid arrays",
      global.outfile
    );
  }

/*
*   Calculate score grids
*   10/95 te
*/
    fprintf (global.outfile, "\nGenerating scoring grids.\n");
    fflush (global.outfile);

    make_grids
    (
      &score_grid,
      &score_bump,
      &score_contact,
      &score_chemical,
      &score_energy,
      &label,
      &receptor
    );

/*
*   Write out grids
*   10/95 te
*/
    write_grids
    (
      &score_grid,
      &score_bump,
      &score_contact,
      &score_chemical,
      &score_energy,
      &label.chemical
    );
  }

  fprintf (global.outfile, "\nFinished calculation.\n");

  return EXIT_SUCCESS;
}


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

void write_program_header ()
{
  extern GLOBAL global;

  if (global.outfile != stdout)
    fprintf (global.outfile, "\n\n"
      "        UUUUUUU    CCCCCCCC    SSSSSSS/  FFFFFFFFF  \n"
      "      UU/    UU/  CC/     CC/     SS/      FF/     FF/ \n"
      "     UU/          CCCCCCCC/      SS/      FF/     FF/  \n"
      "    UU/  UUUU/  CC/ CC\\        SS/      FF/     FF/   \n"
      "   UU/     UU/  CC/    CC\\      SS/      FF/     FF/    \n"
      "   UUUUUUU/    CC/      CC\\  SSSSSS/  FFFFFFFFF/       \n\n\n");

  else
    fprintf (global.outfile, "\n\n"
      "          U?[4mUUUU?[0mUU    CC?[4mCCCC?[0mCC    ?[4mSS?[0mSSS?[4mSS?[0m/
FFF?[4mFFFF?[0mFF  \n"
      "       UU/     UU/  CC/     CC/      SS/      FF/     FF/ \n"
      "      UU/           CCCCCCCC/       SS/      FF/     FF/  \n"
      "     UU/  ?[4mU?[0mUUU/  CC/ CC\\          SS/      FF/      FF/   \n"
      "    UU/     UU/  CC/    CC\\       SS/      FF/     FF/     \n"
      "    ?[4mUUUUUUUU?[0m/   ?[4mCC?[0m/      ?[4mCC\\?[0m  ?[4mSSSSSS?[0m/
?[4mFFFFFFFFF?[0m/       \n\n\n");

  fprintf (global.outfile,
    "University of California at San Francisco, DOCK %s\n", DOCK_VERSION);

  fflush (global.outfile);
}
##############################################################################
###################################### IO.H ##################################
##############################################################################
/*                                                                          */
/*                         Copyright UCSF, 1997                             */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

enum FILE_FORMAT {Unknown, Mol2, Pdb, Xpdb, Ptr, Sph};

enum FILE_FORMAT check_file_extension (FILE_NAME file_name, int report_error);

int read_molecule
(
  MOLECULE  *mol_orig,
  MOLECULE  *mol_init,
  FILE_NAME in_file_name,
  FILE      *in_file,
  int       read_source
);

int write_molecule
(
  MOLECULE  *molecule,
  FILE_NAME in_file_name,
  FILE_NAME out_file_name,
  FILE      *out_file
);

##############################################################################
###################################### IO.C ##################################
##############################################################################
/*                                                                    */
/*                       Copyright UCSF, 1997                         */
/*                                                                    */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "io.h"
#include "io_mol2.h"
#include "io_pdb.h"
#include "io_ptr.h"
#include "io_sph.h"
#include "transform.h"
#include "rotrans.h"

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

enum FILE_FORMAT check_file_extension
(
  FILE_NAME file_name,
  int report_error
)
{
  enum FILE_FORMAT format;
  char *extension;

  format = Unknown;

  if (strchr (file_name, '.'))
  {
    extension = strrchr (file_name, '.') + 1;

    if (!strcmp (extension, "mol2"))
      format = Mol2;

    else if (!strcmp (extension, "pdb"))
      format = Pdb;

    else if (!strcmp (extension, "xpdb"))
      format = Xpdb;

    else if (!strcmp (extension, "ptr"))
      format = Ptr;

    else if (!strcmp (extension, "sph"))
      format = Sph;
  }

  if ((format == Unknown) && report_error)
  {
    fprintf (global.outfile,
      "WARNING check_file_extension: \n"
      "        Coordinate files must have a recognized file extension.\n"
      "        Recognized file extensions are: mol2, pdb, xpdb, ptr, sph\n");
  }

  return format;
}


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

int read_molecule
(
  MOLECULE *mol_ref,
  MOLECULE *mol_init,
  FILE_NAME in_file_name,
  FILE *in_file,
  int read_source
)
{
  enum FILE_FORMAT format;
  int return_value;

/*
* Determine format of input file
* 2/96 te
*/
  format = check_file_extension (in_file_name, TRUE);

  switch (format)
  {
    case Mol2:
      return_value =
        read_mol2
        (
          mol_ref,
          in_file_name,
          in_file
        );

      break;

    case Pdb:
    case Xpdb:
```

```
      return_value =
        read_pdb
        (
          mol_ref,
          in_file_name,
          in_file
        );

      break;

    case Ptr:
      return_value =
        read_ptr
        (
          mol_ref,
          mol_init,
          in_file_name,
          in_file,
          read_source
        );

      break;

    case Sph:
      return_value =
        read_sph
        (
          mol_ref,
          in_file_name,
          in_file
        );

      break;


    default:
      return FALSE;
  }

/*
* Initialize accessory molecule information
* 10/96 te
*/
  if ((return_value == TRUE) && (format != Ptr) && (mol_ref != NULL))
  {
    atom_neighbors (mol_ref);

    center_of_mass
    (
      mol_ref->coord,
      mol_ref->total.atoms,
      mol_ref->transform.com
    );

    if (mol_init != NULL)
      copy_molecule (mol_init, mol_ref);
  }

  return return_value;
}

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

int write_molecule
(
  MOLECULE *molecule,
  FILE_NAME in_file_name,
  FILE_NAME out_file_name,
  FILE *out_file
)
{
  enum FILE_FORMAT format;

/*
* Determine format of output file
* 2/96 te
*/
  format = check_file_extension (out_file_name, TRUE);

  switch (format)
  {
    case Mol2:
      return
        write_mol2
        (
          molecule,
          out_file
        );

    case Pdb:
    case Xpdb:
      return
        write_pdb
        (
          molecule,
          out_file_name,
          out_file
        );

    case Ptr:
      return
        write_ptr
        (
          molecule,
          in_file_name,
          out_file
        );

    case Sph:
      return
        write_sph
        (
          molecule,
          out_file
        );

    default:
      return FALSE;
  }
}

##############################################################################
################################### IO_GRID.H ################################
##############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

void read_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

void write_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

void free_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

int read_box
(
  FILE_NAME file_name,
  XYZ center_of_mass,
  XYZ dimension
);

##############################################################################
################################### IO_GRID.C ################################
##############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "io_grid.h"
#include "io_pdb.h"

void read_grids
(
  SCORE_GRID            *grid,
  SCORE_BUMP            *bump,
  SCORE_CONTACT         *contact,
  SCORE_CHEMICAL        *chemical,
  SCORE_ENERGY          *energy,
  LABEL_CHEMICAL        *label_chemical
)
{
  STRING100 grid_file_name;
  FILE *grid_file = NULL;
  int i, j, jj, grid_size_check;
  int input_var;
  STRING20 *name = NULL;
  int gridnum;
  int *grid2label = NULL;

  void readgrid_ (char *, int *, float *, int [], float [],
    float [], float [], float [], char []);

  grid->init_flag = TRUE;

/*
* Check if old grids requested
* 3/96 te
*/
  if (grid->version < 3.99)
  {
    emalloc
    (
      (void **) &bump->grid,
      grid->size * sizeof (char),
      "bump array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->avdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->bvdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->es,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    if (strlen (grid->file_prefix) < sizeof (FILE_NAME))
      grid->file_prefix[strlen (grid->file_prefix)] = ' ';

    else
      exit (fprintf (global.outfile,
```

```
      return_value =
        read_pdb
        (
          mol_ref,
          in_file_name,
          in_file
        );

      break;

    case Ptr:
      return_value =
        read_ptr
        (
          mol_ref,
          mol_init,
          in_file_name,
          in_file,
          read_source
        );

      break;

    case Sph:
      return_value =
        read_sph
        (
          mol_ref,
          in_file_name,
          in_file
        );

      break;


    default:
      return FALSE;
  }

/*
* Initialize accessory molecule information
* 10/96 te
*/
  if ((return_value == TRUE) && (format != Ptr) && (mol_ref != NULL))
  {
    atom_neighbors (mol_ref);

    center_of_mass
    (
      mol_ref->coord,
      mol_ref->total.atoms,
      mol_ref->transform.com
    );

    if (mol_init != NULL)
      copy_molecule (mol_init, mol_ref);
  }

  return return_value;
}

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

int write_molecule
(
  MOLECULE *molecule,
  FILE_NAME in_file_name,
  FILE_NAME out_file_name,
  FILE *out_file
)
{
  enum FILE_FORMAT format;

/*
* Determine format of output file
* 2/96 te
*/
  format = check_file_extension (out_file_name, TRUE);

  switch (format)
  {
    case Mol2:
      return
        write_mol2
        (
          molecule,
          out_file
        );

    case Pdb:
    case Xpdb:
      return
        write_pdb
        (
          molecule,
          out_file_name,
          out_file
        );

    case Ptr:
      return
        write_ptr
        (
          molecule,
          in_file_name,
          out_file
        );

    case Sph:
      return
        write_sph
        (
          molecule,
          out_file
        );

    default:
      return FALSE;
  }
}

################################################################################
################################### IO_GRID.H ##################################
################################################################################
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

void read_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

void write_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

void free_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL_CHEMICAL *
);

int read_box
(
  FILE_NAME file_name,
  XYZ center_of_mass,
  XYZ dimension
);

################################################################################
################################### IO_GRID.C ##################################
################################################################################
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "io_grid.h"
#include "io_pdb.h"

void read_grids
(
  SCORE_GRID          *grid,
  SCORE_BUMP          *bump,
  SCORE_CONTACT       *contact,
  SCORE_CHEMICAL      *chemical,
  SCORE_ENERGY        *energy,
  LABEL_CHEMICAL      *label_chemical
)
{
  STRING100 grid_file_name;
  FILE *grid_file = NULL;
  int i, j, jj, grid_size_check;
  int input_var;
  STRING20 *name = NULL;
  int gridnum;
  int *grid2label = NULL;

  void readgrid_ (char *, int *, float *, int [], float [],
    float [], float [], float [], char []);

  grid->init_flag = TRUE;

/*
* Check if old grids requested
* 3/96 te
*/
  if (grid->version < 3.99)
  {
    emalloc
    (
      (void **) &bump->grid,
      grid->size * sizeof (char),
      "bump array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->avdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->bvdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->es,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    if (strlen (grid->file_prefix) < sizeof (FILE_NAME))
      grid->file_prefix[strlen (grid->file_prefix)] = ' ';

    else
      exit (fprintf (global.outfile,
```

```
      energy->atom_model == 'a' ? "an all" : "a united");

    emalloc
    (
      (void **) &energy->avdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->bvdw,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    emalloc
    (
      (void **) &energy->es,
      grid->size * sizeof (float),
      "energy grid array",
      global.outfile
    );

    fprintf (global.outfile, "  Reading attractive VDW energy grid.\n");
    fflush (global.outfile);
    efread
    (
      energy->bvdw,
      sizeof (float),
      grid->size,
      grid_file
    );

    fprintf (global.outfile, "  Reading repulsive VDW energy grid.\n");
    fflush (global.outfile);
    efread
    (
      energy->avdw,
      sizeof (float),
      grid->size,
      grid_file
    );

    fprintf (global.outfile, "  Reading electrostatic energy grid.\n");
    fflush (global.outfile);
    efread
    (
      energy->es,
      sizeof (float),
      grid->size,
      grid_file
    );

    fclose (grid_file);
  }

  else
  {
    energy->atom_model = 'a';
    energy->attractive_exponent = 6;
    energy->repulsive_exponent = 12;
  }
}


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

void write_grids
(
  SCORE_GRID          *grid,
  SCORE_BUMP          *bump,
  SCORE_CONTACT       *contact,
  SCORE_CHEMICAL      *chemical,
  SCORE_ENERGY        *energy,
  LABEL_CHEMICAL      *label_chemical
)
{
  int i;
  STRING100 grid_file_name;
  FILE *grid_file;

/*
* Write out bump grid
* 6/95 te
*/
  strcat (strcpy (grid_file_name, grid->file_prefix), ".bmp");
  fprintf (global.outfile, "Writing general grid info to %s\n", grid_file_name);
  grid_file = efopen (grid_file_name, "w", global.outfile);

  efwrite (&grid->size, sizeof (int), 1, grid_file);
  efwrite (&grid->spacing, sizeof (float), 1, grid_file);
  efwrite (grid->origin, sizeof (float), 3, grid_file);
  efwrite (grid->span, sizeof (int), 3, grid_file);

  if (bump->flag || contact->flag || chemical->flag)
  {
    fprintf (global.outfile, "Writing bump grid to %s\n", grid_file_name);
    fflush (global.outfile);

    efwrite
    (
      bump->grid,
      sizeof (char),
      grid->size,
      grid_file
    );
  }

  fclose (grid_file);

/*
* Write out contact grid
* 6/95 te
*/
  if (contact->flag)
  {
    strcat (strcpy (grid_file_name, grid->file_prefix), ".cnt");
    fprintf (global.outfile, "Writing contact grid to %s\n", grid_file_name);
    grid_file = efopen (grid_file_name, "w", global.outfile);
    fflush (global.outfile);

    efwrite (&grid->size, sizeof (int), 1, grid_file);
    efwrite
    (
      contact->grid,
      sizeof (short),
      grid->size,
      grid_file
    );

    fclose (grid_file);
  }

/*
* Write out chemical grids
* 12/96 te
*/
  if (chemical->flag)
  {
    strcat (strcpy (grid_file_name, grid->file_prefix), ".chm");
    fprintf (global.outfile, "Writing chemical grids to %s\n", grid_file_name);
    grid_file = efopen (grid_file_name, "w", global.outfile);
    fflush (global.outfile);

    efwrite (&grid->size, sizeof (int), 1, grid_file);
    efwrite (&label_chemical->total, sizeof (int), 1, grid_file);

    for (i = 0; i < label_chemical->total; i++)
      efwrite (label_chemical->member[i].name, sizeof (STRING20), 1, grid_file);

    for (i = 0; i < label_chemical->total; i++)
    {
      fprintf (global.outfile, "  Writing %s chemical grid\n",
        label_chemical->member[i].name);
      fflush (global.outfile);

      efwrite
      (
        chemical->grid[i],
        sizeof (float),
        grid->size,
        grid_file
      );
    }

    fclose (grid_file);
  }

/*
* Write out energy grids
* 6/95 te
*/
  if (energy->flag)
  {
    strcat (strcpy (grid_file_name, grid->file_prefix), ".nrg");
    fprintf (global.outfile, "Writing energy grids to %s\n", grid_file_name);
    grid_file = efopen (grid_file_name, "w", global.outfile);
    fflush (global.outfile);

    efwrite (&grid->size, sizeof (int), 1, grid_file);
    efwrite (&energy->atom_model, sizeof (int), 1, grid_file);
    efwrite (&energy->attractive_exponent, sizeof (int), 1, grid_file);
    efwrite (&energy->repulsive_exponent, sizeof (int), 1, grid_file);

    fprintf (global.outfile, "  Writing attractive VDW energy grid\n");
    fflush (global.outfile);
    efwrite
    (
      energy->bvdw,
      sizeof (float),
      grid->size,
      grid_file
    );

    fprintf (global.outfile, "  Writing repulsive VDW energy grid\n");
    fflush (global.outfile);
    efwrite
    (
      energy->avdw,
      sizeof (float),
      grid->size,
      grid_file
    );

    fprintf (global.outfile, "  Writing electrostatic energy grid\n");
    fflush (global.outfile);
    efwrite
    (
      energy->es,
      sizeof (float),
      grid->size,
      grid_file
    );

    fclose (grid_file);
  }
}

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

int read_box (char *filename,
              XYZ com,
              XYZ dimension)
{
  STRING80 line;
  FILE *file;

/*
* Open file
* 6/95 te
*/
  file = efopen (filename, "rb", global.outfile);

/*
* Read in data
* 6/95 te
*/
  fgets (line, 80, file);
  fgets (line, 80, file);

  if (sscanf (&line[25], "%f %f %f", &com[0], &com[1], &com[2]) != 3)
  {
    fprintf (global.outfile, "Error reading center of mass from box file.\n");
    return FALSE;
  }

  fprintf (global.outfile, "%-40s: %8.3f %8.3f %8.3f\n",
    "Box center of mass", com[0], com[1], com[2]);

  fgets (line, 80, file);

  if (sscanf (&line[29], "%f %f %f",
    &dimension[0], &dimension[1], &dimension[2]) != 3)
  {
    fprintf (global.outfile, "Error reading dimensions from box file.\n");
```

```
    return FALSE;
  }

  fprintf (global.outfile, "%-40s: %8.3f %8.3f %8.3f\n",
    "Box dimensions", dimension[0], dimension[1], dimension[2]);

  fclose (file);
  return TRUE;
}


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

void free_grids
(
  SCORE_GRID            *grid,
  SCORE_BUMP            *bump,
  SCORE_CONTACT         *contact,
  SCORE_CHEMICAL        *chemical,
  SCORE_ENERGY          *energy,
  LABEL_CHEMICAL        *label_chemical
)
{
  int i;

  if (grid->init_flag)
    free_receptor_grid (grid);

  if (bump->grid)
    efree ((void **) &bump->grid);

  if (contact->flag)
    efree ((void **) &contact->grid);

  if (chemical->flag)
  {
    for (i = 0; i < label_chemical->total; i++)
      efree ((void **) &chemical->grid[i]);

    efree ((void **) &chemical->grid);
  }

  if (energy->flag || chemical->flag)
  {
    efree ((void **) &energy->avdw);
    efree ((void **) &energy->bvdw);
    efree ((void **) &energy->es);
  }
}

**********************************************************************
********************************** IO_LIGAND.H ***********************
**********************************************************************
/*                                                                  */
/*                      Copyright UCSF, 1997                        */
/*                                                                  */

/*
Written by Todd Ewing
10/95
*/

/* Routines used by dock to read and write ligand data */

int get_ligand
(
  DOCK          *dock,
  SCORE         *score,
  LABEL         *label,
  MOLECULE      *lig_orig,
  MOLECULE      *lig_init,
  int           need_bonds,
  int           chem_label,
  int           vdw_label
);

int write_ligand
(
  DOCK          *dock,
  SCORE         *score,
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file
);

void write_ligand_info
(
  DOCK          *dock,
  SCORE         *score,
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file
);

**********************************************************************
********************************** IO_LIGAND.C ***********************
**********************************************************************
/*                                                                  */
/*                      Copyright UCSF, 1997                        */
/*                                                                  */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "score_dock.h"
#include "io.h"
#include "mol_prep.h"
#include "io_ligand.h"
#include "transform.h"
#include "rotrans.h"

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

int get_ligand
(
  DOCK *dock,
  SCORE *score,
  LABEL *label,
  MOLECULE *lig_ref,
  MOLECULE *lig_init,
  int need_bonds,
  int chemical_label,
  int vdw_label
)
{
  int return_value;
  FILE_NAME ready_file_name;
  FILE *ready_file, *quit_file;

/*
* Perform parallel client operations, if necessary
* 10/95 te
*/
  if (dock->parallel.flag && !dock->parallel.server)
  {
/*
*   If the ligand file has already been open, then close it now
*   11/96 te
*/
    if (dock->ligand_file != NULL)
    {
      fclose (dock->ligand_file);
      remove (dock->ligand_file_name);
    }

/*
*   Announce that we are ready to accept a job
*   10/95 te
*/
    sprintf (dock->ligand_file_name, "%s_%s.ptr",
      dock->parallel.server_name,
      dock->parallel.client_name[0]);
    sprintf (ready_file_name, "%s.ready", dock->parallel.client_name[0]);
    ready_file = rfopen (ready_file_name, "w", global.outfile);
    fclose (ready_file);

/*
*   Wait until the ready file has been removed by the server
*   10/95 te
*/
    while (ready_file = fopen (ready_file_name, "r"))
    {
      fclose (ready_file);

      if (quit_file = fopen (dock->quit_file_name, "r"))
      {
        fclose (quit_file);
        remove (ready_file_name);
        remove (dock->ligand_file_name);
        return EOF;
      }
    }

    dock->ligand_file = rfopen (dock->ligand_file_name, "r", global.outfile);
  }

/*
* Reset molecule data structure
* 2/96 te
*/
  reset_molecule (lig_ref);
  lig_ref->info.file_position = ftell (dock->ligand_file);

/*
* Read in coordinates of molecule
* 6/95 te
*/
  return_value =
    read_molecule
    (
      lig_ref,
      lig_init,
      dock->ligand_file_name,
      dock->ligand_file,
      !label->chemical.screen.process_flag
    );

  if (return_value != TRUE)
    return return_value;

/*
* Prepare molecule for docking
* 6/95 te
*/
  if (!label->chemical.screen.process_flag &&
    ((return_value =
      prepare_molecule
      (
        lig_init,
        dock->ligand_file_name,
        dock->ligand_file,
        label,
        score->energy.atom_model,
        need_bonds,
        chemical_label,
        vdw_label
      )) != TRUE))
    return return_value;

/*
* Check number of heavy atoms
* 6/96 te
*/
  if (dock->multiple_ligands && vdw_label &&
    ((lig_init->transform.heavy_total > dock->max_heavies) ||
    (lig_init->transform.heavy_total < dock->min_heavies)))
  {
    if (global.output_volume != 't')
      fprintf (global.outfile, "Skipped %s (%d heavy atoms).\n",
        lig_init->info.name, lig_init->transform.heavy_total);

    return NULL;
  }

  return TRUE;
}


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

int write_ligand
(
  DOCK *dock,
  SCORE *score,
  MOLECULE *molecule,
  FILE_NAME input_molecule_file_name,
  FILE *molecule_file
)
```



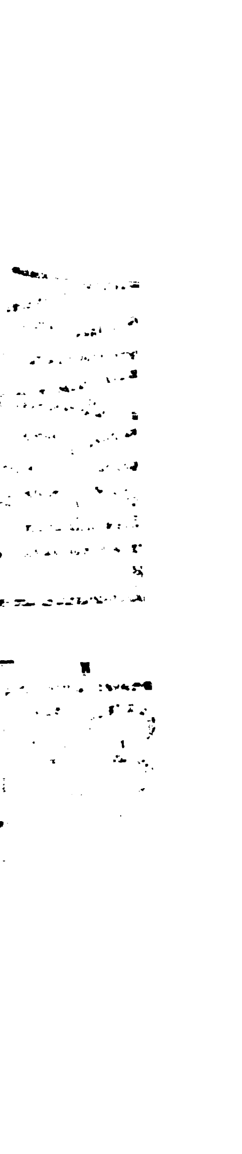
```c
    return FALSE;
  }

  fprintf (global.outfile, "%-40s: %8.3f %8.3f %8.3f\n",
    "Box dimensions", dimension[0], dimension[1], dimension[2]);

  fclose (file);
  return TRUE;
}


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

void free_grids
(
  SCORE_GRID          *grid,
  SCORE_BUMP          *bump,
  SCORE_CONTACT       *contact,
  SCORE_CHEMICAL      *chemical,
  SCORE_ENERGY        *energy,
  LABEL_CHEMICAL      *label_chemical
)
{
  int i;

  if (grid->init_flag)
    free_receptor_grid (grid);

  if (bump->grid)
    efree ((void **) &bump->grid);

  if (contact->flag)
    efree ((void **) &contact->grid);

  if (chemical->flag)
  {
    for (i = 0; i < label_chemical->total; i++)
      efree ((void **) &chemical->grid[i]);

    efree ((void **) &chemical->grid);
  }

  if (energy->flag || chemical->flag)
  {
    efree ((void **) &energy->avdw);
    efree ((void **) &energy->bvdw);
    efree ((void **) &energy->es);
  }
}

##############################################################################
################################### IO_LIGAND.H ##############################
##############################################################################
/*                                                                          */
/*                         Copyright UCSF, 1997                             */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used by dock to read and write ligand data */

int get_ligand
(
  DOCK          *dock,
  SCORE         *score,
  LABEL         *label,
  MOLECULE      *lig_orig,
  MOLECULE      *lig_init,
  int           need_bonds,
  int           chem_label,
  int           vdw_label
);

int write_ligand
(
  DOCK          *dock,
  SCORE         *score,
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file
);

void write_ligand_info
(
  DOCK          *dock,
  SCORE         *score,
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file
);

##############################################################################
################################### IO_LIGAND.C ##############################
##############################################################################
/*                                                                          */
/*                         Copyright UCSF, 1997                             */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "score_dock.h"
#include "io.h"
#include "mol_prep.h"
#include "io_ligand.h"
#include "transform.h"
#include "rotrans.h"

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

int get_ligand
(
  DOCK *dock,
  SCORE *score,
  LABEL *label,
  MOLECULE *lig_ref,
  MOLECULE *lig_init,
  int need_bonds,
  int chemical_label,
  int vdw_label
)
{
  int return_value;
  FILE_NAME ready_file_name;
  FILE *ready_file, *quit_file;

/*
* Perform parallel client operations, if necessary
* 10/95 te
*/
  if (dock->parallel.flag && !dock->parallel.server)
  {
/*
*   If the ligand file has already been open, then close it now
*   11/96 te
*/
    if (dock->ligand_file != NULL)
    {
      fclose (dock->ligand_file);
      remove (dock->ligand_file_name);
    }

/*
*   Announce that we are ready to accept a job
*   10/95 te
*/
    sprintf (dock->ligand_file_name, "%s_%s.ptr",
      dock->parallel.server_name,
      dock->parallel.client_name[0]);
    sprintf (ready_file_name, "%s.ready", dock->parallel.client_name[0]);
    ready_file = rfopen (ready_file_name, "w", global.outfile);
    fclose (ready_file);

/*
*   Wait until the ready file has been removed by the server
*   10/95 te
*/
    while (ready_file = fopen (ready_file_name, "r"))
    {
      fclose (ready_file);

      if (quit_file = fopen (dock->quit_file_name, "r"))
      {
        fclose (quit_file);
        remove (ready_file_name);
        remove (dock->ligand_file_name);
        return EOF;
      }
    }

    dock->ligand_file = rfopen (dock->ligand_file_name, "r", global.outfile);
  }

/*
* Reset molecule data structure
* 2/96 te
*/
  reset_molecule (lig_ref);
  lig_ref->info.file_position = ftell (dock->ligand_file);

/*
* Read in coordinates of molecule
* 6/95 te
*/
  return_value =
    read_molecule
    (
      lig_ref,
      lig_init,
      dock->ligand_file_name,
      dock->ligand_file,
      !label->chemical.screen.process_flag
    );

  if (return_value != TRUE)
    return return_value;

/*
* Prepare molecule for docking
* 6/95 te
*/
  if (!label->chemical.screen.process_flag &&
    ((return_value =
      prepare_molecule
      (
        lig_init,
        dock->ligand_file_name,
        dock->ligand_file,
        label,
        score->energy.atom_model,
        need_bonds,
        chemical_label,
        vdw_label
      )) != TRUE))
    return return_value;

/*
* Check number of heavy atoms
* 6/96 te
*/
  if (dock->multiple_ligands && vdw_label &&
    ((lig_init->transform.heavy_total > dock->max_heavies) ||
    (lig_init->transform.heavy_total < dock->min_heavies)))
  {
    if (global.output_volume != 't')
      fprintf (global.outfile, "Skipped %s (%d heavy atoms).\n",
        lig_init->info.name, lig_init->transform.heavy_total);

    return NULL;
  }

  return TRUE;
}


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

int write_ligand
(
  DOCK *dock,
  SCORE *score,
  MOLECULE *molecule,
  FILE_NAME input_molecule_file_name,
  FILE *molecule_file
)
```

```
      " intramolecular component", molecule->score.intra.total);
    fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
      " intermolecular component", molecule->score.inter.total);
  }

  if ((molecule->score.type == CHEMICAL) || (molecule->score.type == ENERGY))
  {
    fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
      " van der Waals component",
      molecule->score.intra.vdw + molecule->score.inter.vdw);
    fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
      " electrostatic component",
      molecule->score.intra.electro + molecule->score.inter.electro);
  }
 }

 if (dock->multiple_conforms ||
   dock->multiple_orients ||
   score->type[molecule->score.type].minimize)
   fprintf (molecule_file, "%s %-40s : %10.2f\n",
     flag, "RMSD from input orientation (A)", molecule->transform.rmsd);
}


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

void write_trailer
(
  DOCK *dock,
  SCORE *score,
  MOLECULE *molecule,
  FILE *molecule_file
)
{
/*
* Move back one byte to overwrite the previous new line character
* 2/96 te
*/
  fseek (molecule_file, -7, SEEK_END);

/*
* Output score information
* 6/95 te
*/
  if (score->bump.flag)
    fprintf (molecule_file, " <BMP> %d", molecule->score.bumpcount);

  if (molecule->score.total != INITIAL_SCORE)
  {
    if (molecule->score.type == NONE)
      fprintf (molecule_file, " <SCORE> %.2f", molecule->score.total);

    else if (molecule->score.type == CONTACT)
      fprintf (molecule_file, " <CNT> %.2f", molecule->score.total);

    else if (molecule->score.type == CHEMICAL)
      fprintf (molecule_file, " <CHM> %.2f", molecule->score.total);

    else if (molecule->score.type == ENERGY)
      fprintf (molecule_file, " <NRG> %.2f", molecule->score.total);

    else if (molecule->score.type == RMSD)
      fprintf (molecule_file, " <RMS> %.2f", molecule->score.total);

    if (score->inter_flag && score->intra_flag)
    {
      fprintf (molecule_file, " <INTRA> %.2f", molecule->score.intra.total);
      fprintf (molecule_file, " <INTER> %.2f", molecule->score.inter.total);
    }

    if ((molecule->score.type == CHEMICAL) || (molecule->score.type == ENERGY))
    {
      fprintf (molecule_file, " <VDW> %.2f",
        molecule->score.intra.vdw + molecule->score.inter.vdw);
      fprintf (molecule_file, " <ELECTRO> %.2f",
        molecule->score.intra.electro + molecule->score.inter.electro);
    }
  }

  if (dock->multiple_orients || score->type[molecule->score.type].minimize)
    fprintf (molecule_file, " <RMSD> %.2f", molecule->transform.rmsd);

  fprintf (molecule_file, " <END>\n");
}
```

```
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●● IO_MOL2.H ●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
/*                                                                         */
/*                         Copyright UCSF, 1997                            */
/*                                                                         */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

int read_mol2
(
  MOLECULE  *molecule,
  FILE_NAME molecule_file_name,
  FILE      *molecule_file
);

int write_mol2
(
  MOLECULE  *molecule,
  FILE      *molecule_file
);
```

```
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●● IO_MOL2.C ●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●
/*                                                                         */
/*                         Copyright UCSF, 1997                            */
/*                                                                         */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
```

```
#include "io_mol2.h"

int read_mol2
(
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file
)
{
  int i, j;                     /* Iteration variables */
  int token_error = FALSE;      /* Flag for error reading tokens */
  char *line = NULL;            /* Line of data */
  char *token;                  /* Pointer to identify each token */
  int return_value = TRUE;      /* Value to return upon completion */

/*
* Find the next molecule data record
* 6/96 te
*/
  if (find_record (&line, "@<TRIPOS>MOLECULE", molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

/*
* Record molecule name.  If no name, then record default name.
* 10/96 te
*/
  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  for (token = strtok (white_line (line), " "); token;
    token = strtok (NULL, " "))
  {
    if (molecule->info.name)
      vstrcat (&molecule->info.name, " ");
    vstrcat (&molecule->info.name, token);
  }

  if (!molecule->info.name || !strcmp (molecule->info.name, ""))
    vstrcpy (&molecule->info.name, "*****");

/*
* Record the total molecule components and types
* 10/96 te
*/
  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  if (token = strtok (white_line (line), " "))
    molecule->total.atoms = atoi (token);

  else
  {
    fprintf (global.outfile,
      "WARNING read_mol2: incorrect MOLECULE record for %s in %s\n",
      molecule->info.name, molecule_file_name);
    return_value = NULL;
    goto terminate;
  }

  if (token = strtok (NULL, " "))
  {
    molecule->total.bonds = atoi (token);

    if (token = strtok (NULL, " "))
    {
      molecule->total.substs = atoi (token);

      token = strtok (NULL, " ");
        /* Skip number features data record */

      if (token = strtok (NULL, " "))
        molecule->total.sets = atoi (token);

      else
        molecule->total.sets = 0;
    }

    else
    {
      molecule->total.substs = 0;
      molecule->total.sets = 0;
    }
  }

  else
  {
    molecule->total.bonds = 0;
    molecule->total.substs = 0;
    molecule->total.sets = 0;
  }

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  vstrcpy (&molecule->info.molecule_type, strtok (white_line (line), " "));

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  vstrcpy (&molecule->info.charge_type, strtok (white_line (line), " "));

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  if (strncmp (line, "@<TRIPOS>", 9))
  {
    vstrcpy (&molecule->info.status_bits, strtok (white_line (line), " "));

/*
*   Record molecule comment.  If no comment was read,
*   then record default comment.
```

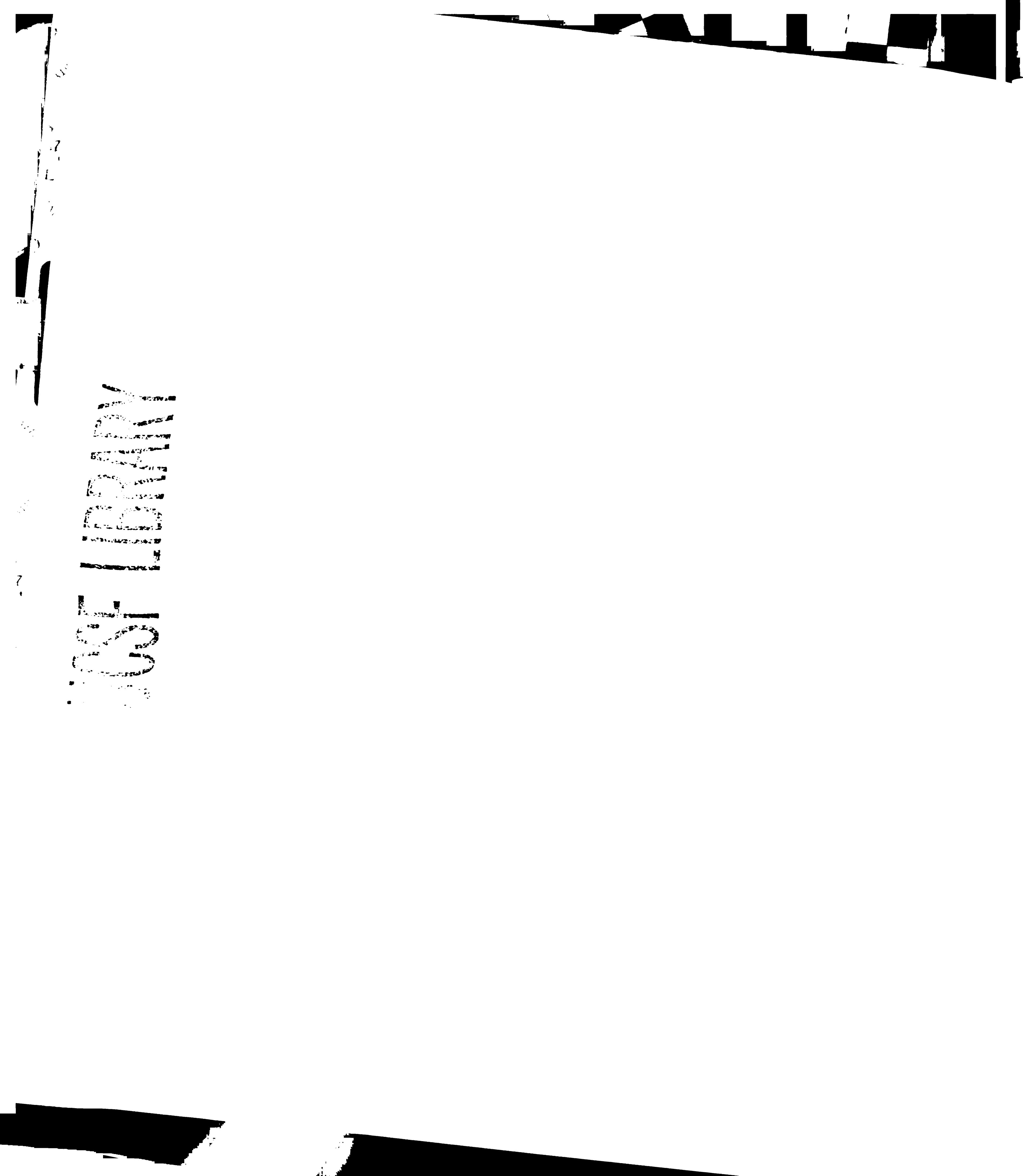


```
        " intramolecular component", molecule->score.intra.total);
      fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
        " intermolecular component", molecule->score.inter.total);
    }

    if ((molecule->score.type == CHEMICAL) || (molecule->score.type == ENERGY))
    {
      fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
        " van der Waals component",
        molecule->score.intra.vdw + molecule->score.inter.vdw);
      fprintf (molecule_file, "%s %-40s   %10.2f\n", flag,
        " electrostatic component",
        molecule->score.intra.electro + molecule->score.inter.electro);
    }
  }

  if (dock->multiple_conforms ||
    dock->multiple_orients ||
    score->type[molecule->score.type].minimize)
    fprintf (molecule_file, "%s %-40s : %10.2f\n",
      flag, "RMSD from input orientation (A)", molecule->transform.rmsd);
}


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

void write_trailer
(
  DOCK *dock,
  SCORE *score,
  MOLECULE *molecule,
  FILE *molecule_file
)
{
/*
* Move back one byte to overwrite the previous new line character
* 2/96 te
*/
  fseek (molecule_file, -7, SEEK_END);

/*
* Output score information
* 6/95 te
*/
  if (score->bump.flag)
    fprintf (molecule_file, " <BMP> %d", molecule->score.bumpcount);

  if (molecule->score.total != INITIAL_SCORE)
  {
    if (molecule->score.type == NONE)
      fprintf (molecule_file, " <SCORE> %.2f", molecule->score.total);

    else if (molecule->score.type == CONTACT)
      fprintf (molecule_file, " <CNT> %.2f", molecule->score.total);

    else if (molecule->score.type == CHEMICAL)
      fprintf (molecule_file, " <CHM> %.2f", molecule->score.total);

    else if (molecule->score.type == ENERGY)
      fprintf (molecule_file, " <NRG> %.2f", molecule->score.total);

    else if (molecule->score.type == RMSD)
      fprintf (molecule_file, " <RMS> %.2f", molecule->score.total);

    if (score->inter_flag && score->intra_flag)
    {
      fprintf (molecule_file, " <INTRA> %.2f", molecule->score.intra.total);
      fprintf (molecule_file, " <INTER> %.2f", molecule->score.inter.total);
    }

    if ((molecule->score.type == CHEMICAL) || (molecule->score.type == ENERGY))
    {
      fprintf (molecule_file, " <VDW> %.2f",
        molecule->score.intra.vdw + molecule->score.inter.vdw);
      fprintf (molecule_file, " <ELECTRO> %.2f",
        molecule->score.intra.electro + molecule->score.inter.electro);
    }
  }

  if (dock->multiple_orients || score->type[molecule->score.type].minimize)
    fprintf (molecule_file, " <RMSD> %.2f", molecule->transform.rmsd);

  fprintf (molecule_file, " <END>\n");
}


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ IO_MOL2.H @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

int read_mol2
(
  MOLECULE  *molecule,
  FILE_NAME molecule_file_name,
  FILE      *molecule_file
);

int write_mol2
(
  MOLECULE  *molecule,
  FILE      *molecule_file
);

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ IO_MOL2.C @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "io_mol2.h"

int read_mol2
(
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file
)
{
  int i, j;                      /* Iteration variables */
  int token_error = FALSE;       /* Flag for error reading tokens */
  char *line = NULL;             /* Line of data */
  char *token;                   /* Pointer to identify each token */
  int return_value = TRUE;       /* Value to return upon completion */

/*
* Find the next molecule data record
* 6/96 te
*/
  if (find_record (&line, "@<TRIPOS>MOLECULE", molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

/*
* Record molecule name.  If no name, then record default name.
* 10/96 te
*/
  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  for (token = strtok (white_line (line), " "); token;
    token = strtok (NULL, " "))
  {
    if (molecule->info.name)
      vstrcat (&molecule->info.name, " ");
    vstrcat (&molecule->info.name, token);
  }

  if (!molecule->info.name || !strcmp (molecule->info.name, ""))
    vstrcpy (&molecule->info.name, "*****");

/*
* Record the total molecule components and types
* 10/96 te
*/
  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  if (token = strtok (white_line (line), " "))
    molecule->total.atoms = atoi (token);

  else
  {
    fprintf (global.outfile,
      "WARNING read_mol2: incorrect MOLECULE record for %s in %s\n",
      molecule->info.name, molecule_file_name);
    return_value = NULL;
    goto terminate;
  }

  if (token = strtok (NULL, " "))
  {
    molecule->total.bonds = atoi (token);

    if (token = strtok (NULL, " "))
    {
      molecule->total.subsets = atoi (token);

      token = strtok (NULL, " ");
        /* Skip number features data record */

      if (token = strtok (NULL, " "))
        molecule->total.sets = atoi (token);

      else
        molecule->total.sets = 0;
    }

    else
    {
      molecule->total.subsets = 0;
      molecule->total.sets = 0;
    }
  }

  else
  {
    molecule->total.bonds = 0;
    molecule->total.subsets = 0;
    molecule->total.sets = 0;
  }

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  vstrcpy (&molecule->info.molecule_type, strtok (white_line (line), " "));

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  vstrcpy (&molecule->info.charge_type, strtok (white_line (line), " "));

  if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

  if (strncmp (line, "@<TRIPOS>", 9))
  {
    vstrcpy (&molecule->info.status_bits, strtok (white_line (line), " "));

/*
*   Record molecule comment.  If no comment was read,
*   then record default comment.
```

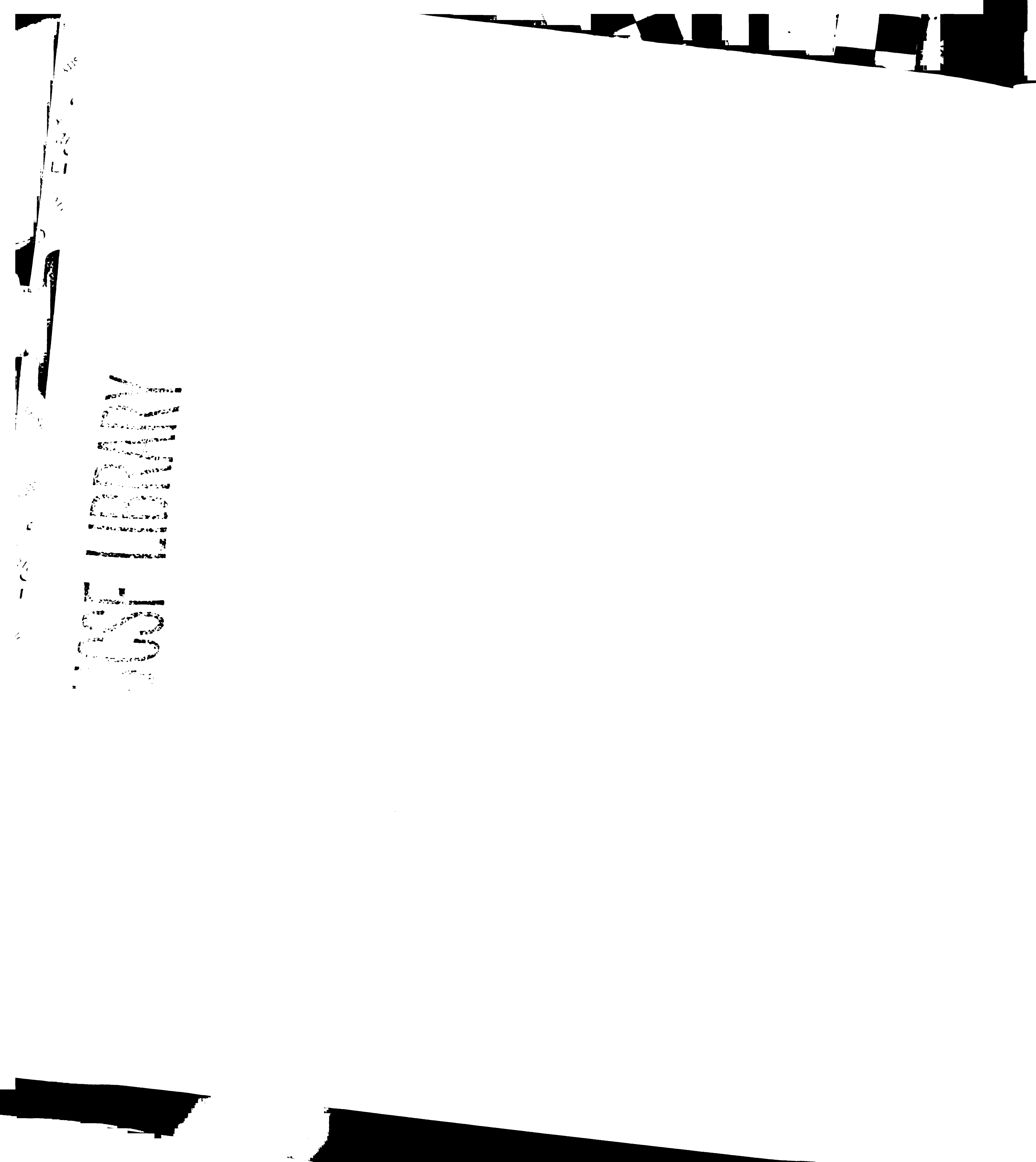
```
*   10/96 te
*/
    if (vfgets (&line, molecule_file) == NULL)
  {
    return_value = EOF;
    goto terminate;
  }

    for (token = strtok (white_line (line), " "); token;
      token = strtok (NULL, " "))
    {
      if (molecule->info.comment)
        vstrcat (&molecule->info.comment, " ");
      vstrcat (&molecule->info.comment, token);
    }

    if (!molecule->info.comment || !strcmp (molecule->info.comment, ""))
      vstrcpy (&molecule->info.comment, "****");
  }

  else
  {
    vstrcpy (&molecule->info.status_bits, "****");
    vstrcpy (&molecule->info.comment, "****");
    fseek (molecule_file, (long) -strlen (line), SEEK_CUR);
  }

/*
* Allocate space for molecule components
* 2/96 te
*/
  reallocate_atoms (molecule);
  reallocate_bonds (molecule);
  reallocate_substs (molecule);
  reallocate_sets (molecule);

/*
* Read in atoms
* 6/95 te
*/
  if (find_record (&line, "@<TRIPOS>ATOM", molecule_file) == NULL)
  {
    fprintf (global.outfile,
      "WARNING read_mol2: Unable to find ATOM record for %s in %s\n",
      molecule->info.name, molecule_file_name);
    return_value = NULL;
    goto terminate;
  }

  for (i = 0; i < molecule->total.atoms; i++)
  {
    if (vfgets (&line, molecule_file) == NULL)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: Incomplete ATOM record for %s in %s\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }

/*
*   Parse each data line into space-separated tokens and check that each
*   token is found.
*   9/95 te
*/
    molecule->atom[i].number = i + 1;

    if (token = strtok (white_line (line), " "));
      /* Skip the atom id field */
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      vstrcpy (&molecule->atom[i].name, token);
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      molecule->coord[i][0] = atof (token);
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      molecule->coord[i][1] = atof (token);
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      molecule->coord[i][2] = atof (token);
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      vstrcpy (&molecule->atom[i].type, token);
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      molecule->atom[i].subst_id = atoi (token) - 1;
    else
      token_error = TRUE;

    if ((molecule->atom[i].subst_id < 0) ||
      (molecule->atom[i].subst_id >= molecule->total.substs))
    {
      if (molecule->total.substs == 1)
        molecule->atom[i].subst_id = 0;

      else
      {
        fprintf (global.outfile,
          "WARNING read_mol2: "
          "Improper ATOM record substructure id for %s in %s.\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }
    }

    if (!token_error && (token = strtok (NULL, " ")));
      /* Skip the substructure name field */
    else
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
      molecule->atom[i].charge = atof (token);
    else
      token_error = TRUE;

    if (token_error)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: Error reading ATOM record for %s in %s.\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }
  }

/*
* Read in bonds
* 6/95 te
*/
  if (molecule->total.bonds > 0)
  {
    if (find_record (&line, "@<TRIPOS>BOND", molecule_file) == NULL)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: Unable to fine BOND record for %s in %s\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }

    for (i = 0; i < molecule->total.bonds; i++)
    {
      if (vfgets (&line, molecule_file) == NULL)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Incomplete BOND record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }

/*
*     Parse each data line into space-separated tokens and check that each
*     token is found.
*     9/95 te
*/
      if (token = strtok (white_line (line), " "));
        /* Skip the bond id field */
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        molecule->bond[i].origin = atoi (token) - 1;
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        molecule->bond[i].target = atoi (token) - 1;
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        vstrcpy (&molecule->bond[i].type, strip_char (token, '\n'));
      else
        token_error = TRUE;

      if (token_error)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Error in BOND record of %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }

      if ((molecule->bond[i].origin >= molecule->total.atoms) ||
        (molecule->bond[i].origin < 0) ||
        (molecule->bond[i].target >= molecule->total.atoms) ||
        (molecule->bond[i].target < 0))
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Improper BOND origin/target for %s in %s.\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }
    }
  }

/*
* Read in substructure information
* 9/95 te
*/
  if (molecule->total.substs > 0)
  {
    if (find_record (&line, "@<TRIPOS>SUBSTRUCTURE", molecule_file) == NULL)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: No SUBSTRUCTURE record for %s in %s\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }

    for (i = 0; i < molecule->total.substs; i++)
    {
      if (vfgets (&line, molecule_file) == NULL)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Incomplete SUBSTRUCTURE record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }
/*
*     Parse each data line into space-separated tokens and check that each
*     token is found.
*     9/95 te
*/
      molecule->subst[i].number = i + 1;

      if (token = strtok (white_line (line), " "));
        /* Skip the substructure id field */
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        vstrcpy (&molecule->subst[i].name, token);
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        molecule->subst[i].root_atom = atoi (token) - 1;
      else
```

```
      token_error = TRUE;

    if (!token_error && (token = strtok (NULL, " ")))
    {
      vstrcpy (&molecule->subst[i].type, strip_char (token, '\n'));

      if (token = strtok (NULL, " "))
      {
        molecule->subst[i].dict_type = atoi (token) - 1;

        if (token = strtok (NULL, " "))
        {
          vstrcpy (&molecule->subst[i].chain, strip_char (token, '\n'));

          if (token = strtok (NULL, " "))
          {
            vstrcpy (&molecule->subst[i].sub_type, strip_char (token, '\n'));

            if (token = strtok (NULL, " "))
            {
              molecule->subst[i].inter_bonds = atoi (token);

              if (token = strtok (NULL, " "))
                vstrcpy (&molecule->subst[i].status, strip_char (token, '\n'));
            }
          }
        }
      }
    }

    if (token_error)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: Error in SUBSTRUCTURE record for %s in %s\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }
  }

/*
*   Update substructure information if more that one substructure is present.
*   (Assume molecule is a macromolecule.)  Extract out the residue number
*   from the substructure name only if the beginning of the name is the
*   same as the sub_type.
*   10/95 te
*/
    if (molecule->total.substs > 1)
    {
      for (i = 0, token_error = TRUE; i < molecule->total.substs; i++)
        if ((strlen (molecule->subst[i].name) <= 3) ||
          !isdigit (molecule->subst[i].name[3]) ||
          (atoi (&molecule->subst[i].name[3]) <= 0))
          token_error = FALSE;

      if (token_error)
        for (i = 0; i < molecule->total.substs; i++)
          molecule->subst[i].number = atoi
            (&molecule->subst[i].name[3]);
    }
  }

  else
  {
    molecule->total.substs = 1;
    reallocate_substs (molecule);

    molecule->subst[0].number = 1;
    vstrcpy (&molecule->subst[0].name, "****");
    molecule->subst[0].root_atom = 0;

    for (i = 0; i < molecule->total.atoms; i++)
      molecule->atom[i].subst_id = 0;
  }

/*
* Read in set information
* 10/96 te
*/
  if (molecule->total.sets > 0)
  {
    token_error = FALSE;

    if (find_record (&line, "@<TRIPOS>SET", molecule_file) == NULL)
    {
      fprintf (global.outfile,
        "WARNING read_mol2: No SET record for %s in %s\n",
        molecule->info.name, molecule_file_name);
      return_value = NULL;
      goto terminate;
    }

    for (i = 0; i < molecule->total.sets; i++)
    {
      if (vfgets (&line, molecule_file) == NULL)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Incomplete SET record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }

      if (token = strtok (white_line (line), " "))
        vstrcpy (&molecule->set[i].name, token);
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        vstrcpy (&molecule->set[i].type, token);
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
        vstrcpy (&molecule->set[i].obj_type, token);
      else
        token_error = TRUE;

      if (!token_error && (token = strtok (NULL, " ")))
      {
        vstrcpy (&molecule->set[i].sub_type, token);

        if (token = strtok (NULL, " "))
        {
          vstrcpy (&molecule->set[i].status, token);

          while (token = strtok (NULL, " "))
          {
            if (molecule->set[i].comment)
              vstrcat (&molecule->set[i].comment, " ");
            vstrcat (&molecule->set[i].comment, token);
          }
        }
      }

      if (token_error)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Error in SET record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }

      if (vfgets (&line, molecule_file) == NULL)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Incomplete SET record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }

      if (token = strtok (white_line (line), " "))
        molecule->set[i].member_total = atoi (token);
      else
        token_error = TRUE;

      emalloc
      (
        (void **) &molecule->set[i].member,
        molecule->set[i].member_total * sizeof (int),
        "molecule set members",
        global.outfile
      );

      for (j = 0; !token_error && (j < molecule->set[i].member_total); j++)
      {
        if (token = strtok (NULL, " "))
        {
          if (strcmp (token, "\\"))
            molecule->set[i].member[j] = atoi (token) - 1;

          else if ((vfgets (&line, molecule_file) == NULL) ||
            ((token = strtok (white_line (line), " ")) == NULL))
            token_error = TRUE;
        }

        else
          token_error = TRUE;
      }

      if (token_error)
      {
        fprintf (global.outfile,
          "WARNING read_mol2: Error in SET record for %s in %s\n",
          molecule->info.name, molecule_file_name);
        return_value = NULL;
        goto terminate;
      }
    }
  }

/*
* Upon termination, free up space
* 11/96 te
*/
  terminate:

  efree ((void **) &line);

  return return_value;
}


/* ============================================================ */

int write_mol2
(
  MOLECULE *molecule,
  FILE *molecule_file
)
{
  int i, j;                     /* Iteration variables */
  STRING20 subst_name;          /* String to construct substructure name */

  fprintf (molecule_file, "\n");
  fprintf (molecule_file, "@<TRIPOS>MOLECULE\n");
  fprintf (molecule_file, "%s\n",
    molecule->info.name ? molecule->info.name : "****");
  fprintf (molecule_file, "%-5d %-5d %-5d %-5d %-5d\n",
    molecule->total.atoms,
    molecule->total.bonds,
    molecule->total.substs, 0,
    molecule->total.sets);

  fprintf (molecule_file, "%s\n",
    molecule->info.molecule_type ? molecule->info.molecule_type : "****");
  fprintf (molecule_file, "%s\n",
    molecule->info.charge_type ? molecule->info.charge_type : "****");

  if (molecule->info.status_bits)
  {
    fprintf (molecule_file, "%s\n", molecule->info.status_bits);
    fprintf (molecule_file, "%s\n",
      molecule->info.comment ? molecule->info.comment : "****");
  }

/*
* Print out atoms
* 6/95 te
*/
  fprintf (molecule_file, "@<TRIPOS>ATOM\n");
  for (i = 0; i < molecule->total.atoms; i++)
  {
    if ((molecule->total.substs > 0) &&
      (molecule->atom[i].subst_id < molecule->total.substs) &&
      molecule->subst[molecule->atom[i].subst_id].name &&
      strcmp (molecule->subst[molecule->atom[i].subst_id].name, "****"))
      sprintf
        (subst_name, "%s", molecule->subst[molecule->atom[i].subst_id].name);
    else
      sprintf (subst_name, "<%d>", molecule->atom[i].subst_id + 1);

    fprintf (molecule_file, "%-5d %-5s %9.4f %9.4f %9.4f %-5s %3d %4s %8.4f\n",
      i + 1,
      molecule->atom[i].name ? molecule->atom[i].name : "****",
      molecule->coord[i][0],
```

```
      molecule->coord[i][1],
      molecule->coord[i][2],
      molecule->atom[i].type ? molecule->atom[i].type : "****",
      molecule->atom[i].subst_id + 1, subst_name,
      molecule->atom[i].charge);

    fflush (molecule_file);
  }

/*
* Print out bonds
* 6/95 te
*/
  fprintf (molecule_file, "@<TRIPOS>BOND\n");
  for (i = 0; i < molecule->total.bonds; i++)
  {
    fprintf (molecule_file, "%-5d %-5d %-5d %s\n",
      i + 1, molecule->bond[i].origin + 1, molecule->bond[i].target + 1,
      molecule->bond[i].type ? molecule->bond[i].type : "****");
  }

/*
* Print out substructures
* 6/95 te
*/
  if (molecule->total.substs > 0)
  {
    fprintf (molecule_file, "@<TRIPOS>SUBSTRUCTURE\n");
    for (i = 0; i < molecule->total.substs; i++)
    {
      fprintf (molecule_file, "%-3d %-6s %-5d",
        i + 1, molecule->subst[i].name, molecule->subst[i].root_atom + 1);
      if (molecule->subst[i].type)
      {
        fprintf (molecule_file, " %-10s", molecule->subst[i].type);
        fprintf (molecule_file, " %-3d", molecule->subst[i].dict_type + 1);
        if (molecule->subst[i].chain)
        {
          fprintf (molecule_file, " %-5s", molecule->subst[i].chain);
          if (molecule->subst[i].sub_type)
          {
            fprintf (molecule_file, " %-5s", molecule->subst[i].sub_type);
            if (molecule->subst[i].inter_bonds)
            {
              fprintf (molecule_file, " %ld", molecule->subst[i].inter_bonds);
              if (molecule->subst[i].status)
                fprintf (molecule_file, " %s", molecule->subst[i].status);
            }
          }
        }
      }

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

/*
* Print out sets
* 10/96 te
*/
  if (molecule->total.sets > 0)
  {
    fprintf (molecule_file, "@<TRIPOS>SET\n");

    for (i = 0; i < molecule->total.sets; i++)
    {
      fprintf (molecule_file, "%s %s %s",
        molecule->set[i].name,
        molecule->set[i].type,
        molecule->set[i].obj_type);

      if (molecule->set[i].sub_type)
      {
        fprintf (molecule_file, " %s", molecule->set[i].sub_type);

        if (molecule->set[i].status)
        {
          fprintf (molecule_file, " %s", molecule->set[i].status);

          if (molecule->set[i].comment)
            fprintf (molecule_file, " %s", molecule->set[i].comment);
        }
      }

      fprintf (molecule_file, "\n%d", molecule->set[i].member_total);

      for (j = 0; j < molecule->set[i].member_total; j++)
        fprintf (molecule_file, " %d", molecule->set[i].member[j] + 1);

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

  fprintf (molecule_file, "\n\n");

  return TRUE;
}

###############################################################################
################################### IO_PDB.H ##################################
###############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

int read_pdb
(
  MOLECULE  *molecule,
  FILE_NAME molecule_file_name,
  FILE      *molecule_file
);

int write_pdb
(
  MOLECULE  *molecule,
  FILE_NAME molecule_file_name,
  FILE      *molecule_file
);

###############################################################################
################################### IO_PDB.C ##################################
###############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "io_pdb.h"

typedef struct link_atom
{
  ATOM atom;
  XYZ coord;
  struct link_atom *previous, *next;
} LINK_ATOM;

typedef struct link_bond
{
  BOND bond;
  struct link_bond *previous, *next;
} LINK_BOND;

typedef struct link_subst
{
  SUBST subst;
  struct link_subst *previous, *next;
} LINK_SUBST;


int read_pdb
(
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file
)
{
  int i, j, origin, target;
  char buff[200];
  STRING20 temp, subst_name;
  int subst_number;
  int found_atom;
  int new_substructure;
  LINK_ATOM *first_atom, *previous_atom, *current_atom;
  LINK_BOND *first_bond, *previous_bond, *current_bond;
  LINK_SUBST *first_subst, *previous_subst, *current_subst;

  void assign_atom_type (ATOM *);

/*
* Initialize molecule info
* 6/95 te
*/
  vstrcpy (&molecule->info.name, "****");
  vstrcpy (&molecule->info.status_bits, "****");
  vstrcpy (&molecule->info.comment, "****");

/*
* Read in molecule info if found before ATOM records
* 6/95 te
*/
  if (!fgets (buff, 199, molecule_file))
    return EOF;

  while (strncmp (buff, "ATOM", 4) && strncmp (buff, "HETATM", 6))
  {
    if (!strncmp (buff, "HEADER", 6))
      sscanf (buff, "%*s %s", molecule->info.name);

    if (!strncmp (buff, "COMPND", 6))
    {
      for (i = 6; (i < strlen (buff)) && isspace (buff[i]); i++);
      vstrcpy (&molecule->info.comment, strip_newline (&buff[i]));
      if (strchr (molecule->info.comment, ' '))
        strcpy (strchr (molecule->info.comment, ' '), "");
    }

    if (!fgets (buff, 199, molecule_file))
      return EOF;
  }

/*
* Initialize linked lists and counters
* 6/95 te
*/
  first_atom = current_atom = previous_atom = NULL;
  first_subst = current_subst = previous_subst = NULL;
  first_bond = current_bond = previous_bond = NULL;

/*
* Read in atom information and store in a linked list
* 6/95 te
*/
  while (!strncmp (buff, "ATOM", 4) || !strncmp (buff, "HETATM", 6))
  {
/*
*   Allocate memory for this ATOM link
*   6/95 te
*/
    previous_atom = current_atom;
    current_atom = NULL;

    ecalloc
    (
      (void **) &current_atom,
      1,
      sizeof (LINK_ATOM),
      "linked atom list",
      global.outfile
    );

/*
*   Update pointers between ATOM links, initialize first link
*   6/95 te
*/
    current_atom->previous = previous_atom;

    if (previous_atom)
      previous_atom->next = current_atom;

    else
      first_atom = current_atom;

/*
*   Deposit atom info in this ATOM link
```



```
*   6/95 te
*/
    memset (temp, 0, sizeof (STRING20));
    current_atom->atom.number = atoi (strncpy (temp, &buff[6], 5));

    sscanf (&buff[12], "%s", temp);
    vstrcpy (&current_atom->atom.name, temp);

    memset (temp, 0, sizeof (STRING20));
    strncpy (subst_name, &buff[17], 3);
    subst_name[3] = 0;

    memset (temp, 0, sizeof (STRING20));
    subst_number = atoi (strncpy (temp, &buff[22], 4));

    memset (temp, 0, sizeof (STRING20));
    current_atom->coord[0] = atof (strncpy (temp, &buff[30], 8));

    memset (temp, 0, sizeof (STRING20));
    current_atom->coord[1] = atof (strncpy (temp, &buff[38], 8));

    memset (temp, 0, sizeof (STRING20)).
    current_atom->coord[2] = atof (strncpy (temp, &buff[46], 8));

    if (!strcmp (strrchr (molecule_file_name, '.'), ".xpdb"))
    {
      memset (temp, 0, sizeof (STRING20));
      current_atom->atom.charge = atof (strncpy (temp, &buff[55], 8));

      memset (temp, 0, sizeof (STRING20));
      strncpy (current_atom->atom.type, &buff[71], 5);
    }

    else
    {
      current_atom->atom.charge = 0.0;
      assign_atom_type (&current_atom->atom);
    }

/*
*   Store substructure information if the current residue is new
*   6/95 te
*/
    if (!previous_atom || !current_subst ||
      (subst_number != current_subst->subst.number))
    {
/*
*     Check to see if this substructure has been seen before
*     6/95 te
*/
      for (i = 0, current_subst = first_subst, new_substructure = TRUE;
        current_subst != NULL;
        i++, previous_subst = current_subst,
        current_subst = current_subst->next)
      {
        if (current_subst->subst.number == subst_number)
        {
          new_substructure = FALSE;
          current_atom->atom.subst_id = i;
        }
      }

/*
*     If the current atom is a member of a new residue, then add a new link
*     10/95 te
*/
      if (new_substructure)
      {
        current_subst = NULL;

        ecalloc
        (
          (void **) &current_subst,
          1,
          sizeof (LINK_SUBST),
          "linked subst list",
          global.outfile
        );

/*
*       Update pointers between SUBST links, initialize first link
*       6/95 te
*/
        current_subst->previous = previous_subst;

        if (previous_subst)
          previous_subst->next = current_subst;
        else
          first_subst = current_subst;

/*
*       Deposit substructure info in this SUBST link
*       6/95 te
*/
        current_subst->subst.number = subst_number;
        vstrcpy (&current_subst->subst.name, subst_name);
        current_subst->subst.root_atom = molecule->total.atoms;

        current_atom->atom.subst_id = molecule->total.substs++;
      }

/*
*     Otherwise, set current subst pointer to the last link added
*     10/95 te
*/
      else
        current_subst = previous_subst;
    }

    else
      current_atom->atom.subst_id = molecule->total.substs - 1;

/*
*   Update atom info
*   10/95 te
*/
    molecule->total.atoms++;

/*
*   Read in next line of file
*   6/95 te
*/
    if (!fgets (buff, 199, molecule_file))
      break;
  }

/*
* Advance to TER record, but stop if CONECT records are found
* 6/95 te
*/
  while (strncmp (buff, "TER", 3))
  {
    if (!strncmp (buff, "CONECT", 6))
      break;

    if (!fgets (buff, 199, molecule_file))
      break;
  }

/*
* Read in connectivity information
* 6/95 te
*/
  while (!strncmp (buff, "CONECT", 6))
  {
    memset (temp, 0, sizeof (STRING20));
    origin = atoi (strncpy (temp, &buff[6], 5));

/*
*   Scan the data in the CONECT record
*   6/95 te
*/
    for (i = 1; target = atoi (strncpy (temp, &buff[i * 5 + 6], 5)); i++)
    {
      if (target > origin)
      {
/*
*       Allocate memory for this BOND link
*       6/95 te
*/
        previous_bond = current_bond;
        current_bond = NULL;

        ecalloc
        (
          (void **) &current_bond,
          1,
          sizeof (LINK_BOND),
          "linked bond list",
          global.outfile
        );

/*
*       Update pointers between BOND links, initialize first link
*       6/95 te
*/
        current_bond->previous = previous_bond;

        if (previous_bond)
          previous_bond->next = current_bond;

        else
          first_bond = current_bond;

/*
*       Deposit bond info in this BOND link
*       6/95 te
*/
        current_bond->bond.origin = origin;
        current_bond->bond.target = target;
        vstrcpy (&current_bond->bond.type, "1");
        molecule->total.bonds++;
      }
    }

/*
*   Read in next line of file
*   6/95 te
*/
    if (!fgets (buff, 199, molecule_file))
      break;
  }

/*
* Advance to TER record
* 6/95 te
*/
  while (strncmp (buff, "TER", 3))
    if (!fgets (buff, 199, molecule_file))
      break;

/*
* Allocate space for molecule components
* 2/96 te
*/
  reallocate_atoms (molecule);
  reallocate_bonds (molecule);
  reallocate_substs (molecule);

/*
* Copy atom info into molecule structure
* 6/95 te
*/
  for (i = 0, current_atom = first_atom;
    (i < molecule->total.atoms) && (current_atom != NULL);
    i++, current_atom = current_atom->next)
  {
    copy_atom (&molecule->atom[i], &current_atom->atom);
    copy_coord (molecule->coord[i], current_atom->coord);
  }

  if (i != molecule->total.atoms)
    exit (fprintf (global outfile,
      "ERROR read_pdb: Error in atom linked list for %s in %s.\n",
      molecule->info.name, molecule_file_name));

/*
* Copy bond info into molecule structure
* 6/95 te
*/
  for (i = 0, current_bond = first_bond;
    (i < molecule->total.bonds) && (current_bond != NULL);
    i++, current_bond = current_bond->next)
  {
    copy_bond (&molecule->bond[i], &current_bond->bond);

/*
*   Convert bond reference from atom number to position in atom array
*   10/95 te
*/
    for (j = 0, found_atom = FALSE; j < molecule->total.atoms; j++)
    {
      if (molecule->bond[i].origin == molecule->atom[j].number)
      {
        molecule->bond[i].origin = j;
        found_atom = TRUE;
      }
    }
```

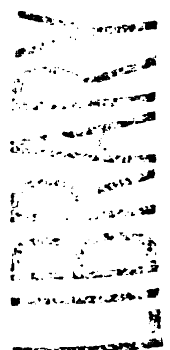


```c
    if (!found_atom)
      exit (fprintf (global.outfile,
        "ERROR read_pdb: Unknown atom in CONECT record for %s in %s\n",
        molecule->info.name, molecule_file_name));

    for (j = 0, found_atom = FALSE; j < molecule->total.atoms; j++)
    {
      if (molecule->bond[i].target == molecule->atom[j].number)
      {
        molecule->bond[i].target = j;
        found_atom = TRUE;
      }
    }

    if (!found_atom)
      exit (fprintf (global.outfile,
        "ERROR read_pdb: Unknown atom in CONECT record for %s in %s\n",
        molecule->info.name, molecule_file_name));
  }

  if (i != molecule->total.bonds)
    exit (fprintf (global.outfile,
      "ERROR read_pdb: Error in bond linked list for %s in %s\n",
      molecule->info.name, molecule_file_name));

/*
* Copy substructure info into molecule structure
* 6/95 te
*/
  for (i = 0, current_subst = first_subst;
    (i < molecule->total.substs) && (current_subst != NULL);
    i++, current_subst = current_subst->next)
    copy_subst (&molecule->subst[i], &current_subst->subst);

  if (i != molecule->total.substs)
    exit (fprintf (global.outfile,
      "ERROR read_pdb: Error in subst linked list for %s in %s\n",
      molecule->info.name, molecule_file_name));

/*
* Update substructure information if more than one substructure read
* (Assume molecule is a macromolecule.)
* 10/95 te
*/
  if (molecule->total.substs > 1)
  {
    for (i = 0; i < molecule->total.substs; i++)
    {
      vstrcpy (&molecule->subst[i] type, "RESIDUE");
      molecule->subst[i] dict_type = 0;
      vstrcpy (&molecule->subst[i].chain, "A");
      vstrcpy (&molecule->subst[i].sub_type, molecule->subst[i].name);

      sprintf (temp, "%d", molecule->subst[i].number);
      vstrcat (&molecule->subst[i].name, temp);
    }
  }

/*
* Free up memory allocated to linked lists
* 6/95 te
*/
  for (previous_atom = first_atom;
    previous_atom != NULL;
    previous_atom = current_atom)
  {
    current_atom = previous_atom->next;
    efree ((void **) &previous_atom);
  }

  for (previous_bond = first_bond;
    previous_bond != NULL;
    previous_bond = current_bond)
  {
    current_bond = previous_bond->next;
    efree ((void **) &previous_bond);
  }

  for (previous_subst = first_subst;
    previous_subst != NULL;
    previous_subst = current_subst)
  {
    current_subst = previous_subst->next;
    efree ((void **) &previous_subst);
  }

  return TRUE;
}


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

int write_pdb
(
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file
)
{
  int i, j;
  int atom;
  int segment;
  int neighbor;
  STRING80 line;

  fprintf (molecule_file, "HEADER     %s\n", molecule->info.name);
  fprintf (molecule_file, "COMPND     %s\n", molecule->info.comment);
  fprintf (molecule_file, "AUTHOR     Generated by %s version %s\n",
    global.executable, DOCK_VERSION);

/*
* Write out atoms
* 12/96 te
*/
  for (atom = 0; atom < molecule->total.atoms; atom++)
  {
    if
    (
      (molecule->total.layers > 0) &&
      (
        ((segment = molecule->atom[atom].segment_id) != NEITHER) &&
        (molecule->segment[segment].active_flag == FALSE)
      )
    )
      continue;

    memset (line, 0, sizeof (STRING80));
    sprintf (&line[0], "%4s", "ATOM");
    sprintf (&line[6], "%5d", molecule->atom[atom].number);
    sprintf (&line[12], "%-4s",
      molecule->atom[atom].name ? molecule->atom[atom].name : "UNK");
    sprintf (&line[17], "%3.3s",
      (molecule->total substs > molecule->atom[atom].subst_id) &&
      molecule->subst[molecule->atom[atom].subst_id].name &&
        strcmp (molecule->subst[molecule->atom[atom].subst_id].name, "****") ?
        molecule->subst[molecule->atom[atom].subst_id].name : "UNK");
    sprintf (&line[22], "%4d", molecule->total.substs > 1 ?
      molecule->subst[molecule->atom[atom].subst_id].number :
      molecule->info output_id);
    sprintf (&line[30], "%8.3f", molecule->coord[atom][0]);
    sprintf (&line[38], "%8.3f", molecule->coord[atom][1]);
    sprintf (&line[46], "%8.3f", molecule->coord[atom][2]);

/*
*   If extended pdb requested, then print out charge and atom type
*   6/95 te
*/
    if (!strcmp (strrchr (molecule_file_name, '.') + 1, "xpdb"))
    {
      sprintf (&line[55], "%10.4f", molecule->atom[atom].charge);
      sprintf (&line[71], "%-5s", molecule->atom[atom].type);
    }

/*
*   Replace null characters with spaces
*   6/95 te
*/
    for (i = 0; i < sizeof (STRING80); i++)
      if (line[i] == '\0') line[i] = ' ';
    line[sizeof (STRING80) - 2] = '\0';

    fprintf (molecule_file, "%s\n", line);
  }

/*
* Write out connectivity
* 12/96 te
*/
  if (molecule->total.bonds)
  {
    for (atom = 0; atom < molecule->total.atoms; atom++)
    {
      if
      (
        (molecule->total.layers > 0) &&
        (
          ((segment = molecule->atom[atom].segment_id) == NEITHER) ||
          (molecule->segment[segment].active_flag == FALSE)
        )
      )
        continue;

      memset (line, 0, sizeof (STRING80));
      sprintf (&line[0], "%-5s", "CONECT");
      sprintf (&line[6], "%5d", molecule->atom[atom].number);

      for (i = j = 0; i < molecule->atom[atom].neighbor_total; i++)
      {
        neighbor = molecule->atom[atom].neighbor[i].id;

        if
        (
          (molecule->total.layers > 0) &&
          (
            ((segment = molecule->atom[neighbor].segment_id) == NEITHER) ||
            (molecule->segment[segment].active_flag == FALSE)
          )
        )
          continue;

        sprintf (&line[j++ * 5 + 11], "%5d",
          molecule->atom[neighbor].number);
      }

/*
*     Replace null characters with spaces
*     6/95 te
*/
      for (i = 0; i < sizeof (STRING80); i++)
        if (line[i] == '\0') line[i] = ' ';
      line[sizeof (STRING80) - 2] = '\0';

      fprintf (molecule_file, "%s\n", line);
    }
  }

  fprintf (molecule_file, "TER\n");

  return TRUE;
}


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

void assign_atom_type (ATOM *atom)
{
  int i;
  STRING5 atom_name[13] =
    {"C", "N", "O", "S", "P", "H", "F", "CL", "BR", "SI", "I", "DU", "LP"};
  STRING5 atom_type[13] =
    {"C.3", "N.3", "O.3", "S.3", "P.3", "H", "F", "Cl", "Br", "Si", "I", "Du", "LP"};

  for (i = 0; i < 13; i++)
    if (strstr (atom->name, atom_name[i]))
      vstrcpy (&atom->type, atom_type[i]);
}

*****************************************************************************
*********************************** IO_PTR.H ********************************
*****************************************************************************
/*                                                                        */
/*                          Copyright UCSF, 1997                          */
/*                                                                        */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

int read_ptr
(
  MOLECULE  *mol_ref,
  MOLECULE  *mol_init,
  FILE_NAME in_file_name,
  FILE      *in_file,
```

```
  int       read_source
);

int write_ptr
(
  MOLECULE  *molecule,
  FILE_NAME in_file_name,
  FILE      *out_file
);

###############################################################################
################################### IO_PTR.C ##################################
###############################################################################
/*                                                                           */
/*                         Copyright UCSF, 1997                              */
/*                                                                           */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "io.h"
#include "io_ptr.h"
#include "vector.h"
#include "transform.h"
#include "rotrans.h"


int read_ptr
(
  MOLECULE *mol_ref,
  MOLECULE *mol_init,
  FILE_NAME in_file_name,
  FILE *in_file,
  int read_source
)
{
  int i, j;
  int read_flag = FALSE;
  char *line = NULL;
  char *token;
  STRING20 field;
  FILE *source_file = NULL;
  static FILE_NAME previous_file_name;
  static FILE *previous_file = NULL;
  MOLECULE temporary = {0};

  reset_molecule (&temporary);

/*
* Check if termination requested
* 5/97 te
*/
  if ((mol_ref == NULL) && (mol_init == NULL))
  {
    if (previous_file != NULL)
      efclose (&previous_file);

    return EOF;
  }

/*
* Read in line of data from file
* 6/96 te
*/
  if (!vfgets (&line, in_file))
    return EOF;

/*
* Process each data item in line, one at a time
* 10/96 te
*/
  for (token = strtok (white_line (line), " "); token;
    token = strtok (NULL, " "))
  {
/*
*   Read in file name containing source molecule
*   10/96 te
*/
    if (!strcmp (token, "<FILE>"))
    {
      if (token = strtok (NULL, " "))
        vstrcpy (&mol_ref->info.source_file, token);

      else
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <FILE> data in %s\n", in_file_name));
    }

/*
*   Get file position
*   10/96 te
*/
    else if (!strcmp (token, "<FPOS>"))
    {
      if (token = strtok (NULL, " "))
        mol_ref->info.source_position = atol (token);

      else
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <FPOS> data in %s\n", in_file_name));
    }

/*
*   If the source file is not to be read (chemical screen run),
*   then be sure to read in the molecule name, description and screen keys
*   11/96 te
*/
    if (!read_source)
    {
/*
*     Read in file name containing source molecule
*     11/96 te
*/
      if (!strcmp (token, "<NAME>"))
      {
        if (token = strtok (NULL, " "))
          vstrcpy (&mol_ref->info.name, token);

        else
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <NAME> data in %s\n", in_file_name));
      }

/*
*     Read in file name containing source molecule
*     11/96 te
*/
      else if (!strcmp (token, "<DESCR>"))
      {
        if (token = strtok (NULL, " "))
          vstrcpy (&mol_ref->info.comment, token);

        else
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <DESCR> data in %s\n",
            in_file_name));
      }

/*
*     Check if chemical key information is present
*     10/96 te
*/
      else if (!strcmp (token, "<KEY>"))
      {
        if (token = strtok (NULL, " "))
          mol_ref->total.keys = atoi (token);

        else
          exit (fprintf (global.outfile,
            "* * * Missing <KEY> data in pointer file.\n"));

        reallocate_keys (mol_ref);

        if (!(token = strtok (NULL, " ")) || strcmp (token, "<KFOLD>"))
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <KFOLD> field in %s\n",
            in_file_name));

        if (token = strtok (NULL, " "))
          mol_ref->transform.fold_flag = atoi (token);

        else
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <KFOLD> data in %s\n",
            in_file_name));

        if (mol_ref->transform.fold_flag != TRUE)
        {
          mol_ref->total.atoms =
            mol_ref->transform.heavy_total =
            mol_ref->total.keys;
          reallocate_atoms (mol_ref);
        }

        for (i = 0; i < mol_ref->total.keys; i++)
        {
          sprintf (field, "<KI%d>", i + 1);

          if (!(token = strtok (NULL, " ")) || strcmp (token, field))
            exit (fprintf (global.outfile,
              "ERROR read_ptr: Missing %s field in %s\n",
              field, in_file_name));

          if (token = strtok (NULL, " "))
          {
            if (mol_ref->transform.fold_flag != TRUE)
              mol_ref->atom[i].chem_id = atoi (token);
          }

          else
            exit (fprintf (global.outfile,
              "ERROR read_ptr: Missing %s data in %s\n",
              field, in_file_name));

          for (j = i; j < mol_ref->total.keys; j++)
          {
            sprintf (field, "<KJ%d>", j + 1);

            if (!(token = strtok (NULL, " ")) || strcmp (token, field))
              exit (fprintf (global.outfile,
                "ERROR read_ptr: Missing %s field in %s\n",
                field, in_file_name));

            if (token = strtok (NULL, " "))
              mol_ref->key[i][j].count = atoi (token);

            else
              exit (fprintf (global.outfile,
                "ERROR read_ptr: Missing %s data in %s\n",
                field, in_file_name));

            if (token = strtok (NULL, " "))
              mol_ref->key[i][j].distance = strtoul (token, NULL, 16);

            else
              exit (fprintf (global.outfile,
                "ERROR read_ptr: Missing %s data in %s\n",
                field, in_file_name));
          }
        }
      }

      continue;
    }

/*
*   Skip other initialization info if no data structure was passed to hold it
*   10/96 te
*/
    if (mol_init == NULL)
      continue;

/*
*   Check if rot/trans information is present
*   10/96 te
*/
    else if (!strcmp (token, "<TRANS>"))
    {
      temporary.transform.flag = TRUE;
      temporary.transform.trans_flag = TRUE;

      for (i = 0; i < 3; i++)
      {
        if (token = strtok (NULL, " "))
          temporary.transform.translate[i] = atof (token);

        else
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <TRANS> data in %s\n",
            in_file_name));
      }
    }
```

```
    else if (!strcmp (token, "<ROT>"))
    {
      temporary.transform.flag = TRUE;
      temporary.transform.rot_flag = TRUE;

      for (i = 0; i < 3; i++)
      {
        if (token = strtok (NULL, " "))
          temporary.transform.rotate[i] = atof (token);

        else
          exit (fprintf (global.outfile,
            "ERROR read_ptr: Missing <ROT> data in %s\n",
            in_file_name));
      }
    }

/*
*   Check if torsion information is present
*   10/96 te
*/
    else if (!strcmp (token, "<TORS>"))
    {
      temporary.transform.flag = TRUE;

      if (token = strtok (NULL, " "))
        temporary.total.torsions = atoi (token);

      else
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <TORS> data in %s\n",
          in_file_name));

      if (!(token = strtok (NULL, " ")) || strcmp (token, "<TANCHOR>"))
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <TANCHOR> field in %s\n",
          in_file_name));

      if (token = strtok (NULL, " "))
        temporary.transform.anchor_atom = atoi (token) - 1;

      else
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <TANCHOR> data in %s\n",
          in_file_name));

      if (temporary.transform.anchor_atom < 0)
        temporary.total.torsions = 0;

      if (temporary.total.torsions > 0)
      {
        temporary.transform.tors_flag = TRUE;
        reallocate_torsions (&temporary);

        for (i = 0; i < temporary.total.torsions; i++)
        {
          sprintf (field, "<T%d>", i + 1);

          if (!(token = strtok (NULL, " ")) || strcmp (token, field))
            exit (fprintf (global.outfile,
              "ERROR read_ptr: Missing %s field in %s\n",
              field, in_file_name)),

          if (token = strtok (NULL, " "))
            temporary.torsion[i].bond_id = atoi (token) - 1;

          else
            exit (fprintf (global.outfile,
              "ERROR read_ptr: Missing %s data in %s\n",
              field, in_file_name));

          if (token = strtok (NULL, " "))
            temporary.torsion[i].target_angle = atof (token) * PI / 180.0;

          else
            exit (fprintf (global.outfile,
              "ERROR read_ptr: Missing %s data in %s\n",
              field, in_file_name));
        }
      }
    }

/*
*   Check if reflection information is present
*   10/96 te
*/
    else if (!strcmp (token, "<REFL>"))
    {
      if (token = strtok (NULL, " "))
        temporary.transform.refl_flag = atoi (token);

      else
        exit (fprintf (global.outfile,
          "ERROR read_ptr: Missing <REFL> data in %s\n",
          in_file_name));

      if (temporary.transform.refl_flag == TRUE)
        temporary.transform.flag = TRUE;
    }
  }

  efree ((void **) &line);

/*
* Check if molecule file and position were read
* 10/96 te
*/
  if (!strcmp (mol_ref->info.source_file, "") ||
    (mol_ref->info.source_position < 0))
    exit (fprintf (global.outfile,
      "ERROR read_ptr: Missing <FILE> and/or <FPOS> field in %s\n",
      in_file_name));

  if (read_source)
  {
/*
*   Check to see if the current file is still open
*   6/96 te
*/
    if (!strcmp (mol_ref->info.source_file, previous_file_name))
      source_file = previous_file;

    else
    {
      source_file = efopen (mol_ref->info.source_file, "r", global.outfile);
      strcpy (previous_file_name, mol_ref->info.source_file);

      if (previous_file != NULL)
        efclose (&previous_file);

      previous_file = source_file;
    }

/*
*   Read in molecule
*   10/96 te
*/
    fseek (source_file, mol_ref->info.source_position, SEEK_SET);

    read_flag = read_molecule
      (mol_ref, mol_init, mol_ref->info.source_file, source_file, FALSE);

    if ((read_flag != TRUE) || (mol_init == NULL))
      return read_flag;

    if (temporary.transform.flag == TRUE)
    {
      copy_transform (mol_init, &temporary);

      if (temporary.total.torsions > 0)
      {
        copy_torsions (mol_init, &temporary);
        revise_atom_neighbors (mol_init);
        get_torsion_neighbors (mol_init);
      }

      center_of_mass
      (
        mol_init->coord,
        mol_init->total.atoms,
        mol_init->transform.com
      );

/*
*     Generate initial coordinates based on transformation specified in input
*     7/96 te
*/
      transform_molecule (mol_init, mol_ref);
    }
  }

  else if (mol_init != NULL)
    copy_molecule (mol_init, mol_ref);

  free_molecule (&temporary);
  return TRUE;
}


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

int write_ptr
(
  MOLECULE *molecule,
  FILE_NAME in_file_name,
  FILE *out_file
)
{
  int i, j;
  char *word = NULL;

/*
* Output molecule identifiers
* 10/96 te
*/
  fprintf (out_file, "<ID> %d", molecule->info.output_id);

  if (molecule->info.input_id != NEITHER)
    fprintf (out_file, " <SRCID> %d", molecule->info.input_id);

  fprintf (out_file, " <NAME> %s",
    subst_char (vstrcpy (&word, molecule->info.name), ' ', '_'));
  fprintf (out_file, " <DESCR> %s",
    subst_char (vstrcpy (&word, molecule->info.comment), ' ', '_'));

  efree ((void **) &word);

/*
* Output source file name and position
* 6/95 te
*/
  if (molecule->info.source_file)
  {
    fprintf (out_file, " <FILE> %s", molecule->info.source_file);
    fprintf (out_file, " <FPOS> %d", molecule->info.source_position);
  }

  else
  {
    fprintf (out_file, " <FILE> %s", in_file_name);
    fprintf (out_file, " <FPOS> %d", molecule->info.file_position);
  }

/*
* Output chemical keys
* 1/97 te
*/
  if (molecule->total.keys > 0)
  {
    fprintf (out_file, " <KEY> %d", molecule->total.keys);
    fprintf (out_file, " <KFOLD> %d", molecule->transform.fold_flag);

/*
*   Write out each chemical key
*   11/96 te
*/
    for (i = 0; i < molecule->total.keys; i++)
    {
      fprintf (out_file, " <KI%d>", i + 1);

      if (molecule->transform.fold_flag == TRUE)
        fprintf (out_file, " %d", i);

      else
        fprintf (out_file, " %d", molecule->atom[i].chem_id);

      for (j = i; j < molecule->total.keys; j++)
      {
        fprintf (out_file, " <KJ%d>", j + 1);
        fprintf (out_file, " %d", molecule->key[i][j].count);
        fprintf (out_file, " %lx", molecule->key[i][j].distance);
      }
    }

    fprintf (out_file, " <END>\n");
    return TRUE;
  }
```

```
/*
* If a transformation has occurred, then output rotation/translation info
* 10/96 te
*/
  if (molecule->transform.trans_flag == TRUE)
  {
    fprintf (out_file, " <TRANS>");

    for (i = 0; i < 3; i++)
      fprintf (out_file, " %.6g", molecule->transform.translate[i]);
  }

  if (molecule->transform.rot_flag == TRUE)
  {
    fprintf (out_file, " <ROT>");

    for (i = 0; i < 3; i++)
      fprintf (out_file, " %.6g", molecule->transform.rotate[i]);
  }

/*
* Signal flexible bond rotation
* 10/96 te
*/
  if ((molecule->transform.tors_flag != NEITHER) &&
    (molecule->transform.anchor_atom != NEITHER))
  {
    fprintf (out_file, " <TORS> %d", molecule->total.torsions);
    fprintf (out_file, " <TANCHOR> %d", molecule->transform.anchor_atom + 1);

/*
*   Write out each rotatable bond and angle.
*   10/96 te
*/
    for (i = 0; i < molecule->total.torsions; i++)
    {
      fprintf (out_file, " <T%d>", i + 1);
      fprintf (out_file, " %d", molecule->torsion[i].bond_id + 1);
      fprintf (out_file, " %.6g",
        molecule->torsion[i].current_angle / PI * 180.0);
    }
  }

/*
* Write out reflection state
* 10/96 te
  fprintf (out_file, " <HEAVY> %d", molecule->transform.heavy_total);
*/
  if (molecule->transform.refl_flag != NEITHER)
    fprintf (out_file, " <REFL> %d", molecule->transform.refl_flag);

  fprintf (out_file, " <END>\n");
  return TRUE;
}

##############################################################################
################################# IO_RECEPTOR.H ##############################
##############################################################################
/*                                                                          */
/*                        Copyright UCSF, 1997                              */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used by dock to read receptor data */

int read_receptor
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file,
  int           need_bonds,
  int           label_chemical,
  int           label_vdw
);


##############################################################################
################################# IO_RECEPTOR.C ##############################
##############################################################################
/*                                                                          */
/*                        Copyright UCSF, 1997                              */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "io.h"
#include "mol_prep.h"

MOLECULE temporary = {0};

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

int read_receptor
(
  SCORE_ENERGY *energy,
  LABEL *label,
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file,
  int need_bonds,
  int label_chemical,
  int label_vdw
)
{
  int i, j, return_value;
  int root_atom, root_subst_id, neighbor_subst_id;
  int *old2new = NULL;
  STRING80 line;
  float residue_charge, molecule_charge;
  int *order = NULL;

  int compare_substs ();
  int compare_bonds ();

/*
* Read in coordinates of molecule
* 2/96 te
*/
  return_value =
    read_molecule
    (
      &temporary,
      molecule,
      molecule_file_name,
      molecule_file,
      TRUE
    );

  free_molecule (&temporary);

  if (return_value != TRUE)
    return return_value;

/*
* Prepare molecule for grid calculation
* 2/96 te
*/
  if ((return_value =
    prepare_molecule
    (
      molecule,
      molecule_file_name,
      molecule_file,
      label,
      energy->atom_model,
      need_bonds,
      label_chemical,
      label_vdw
    )) != TRUE)
    return return_value;

/*
* If capping groups were added by SYBYL, then merge them with the substructure
* to which they are attached.
* 10/95 te
*/
  for (i = 0; i < molecule->total.substs; i++)
    if ((strstr (molecule->subst[i].sub_type, "AMN")) ||
      (strstr (molecule->subst[i].sub_type, "CXL")))
    {
      root_atom = molecule->subst[i].root_atom;

      root_subst_id = neighbor_subst_id = molecule->atom[root_atom].subst_id;

      for (j = 0;
        (neighbor_subst_id == root_subst_id) &&
        (j < molecule->atom[root_atom].neighbor_total); j++)
      {
        neighbor_subst_id =
          molecule->atom[molecule->atom[root_atom].neighbor[j].id].subst_id;
      }

      if (neighbor_subst_id != root_subst_id)
      {
        fprintf (global.outfile, "Merging %s %d cap residue with %s %d residue.\n",
          molecule->subst[root_subst_id].name, root_subst_id + 1,
          molecule->subst[neighbor_subst_id].name, neighbor_subst_id + 1);

/*
*       Copy over merged residue info to root atom in cap
*/
        molecule->atom[root_atom].subst_id = neighbor_subst_id;

/*
*       Copy over merged residue info to other atoms in cap
*/
        for (j = 0; j < molecule->atom[root_atom].neighbor_total; j++)
          if (molecule->atom[molecule->atom[root_atom].neighbor[j].id].subst_id
            == root_subst_id)
            molecule->atom[molecule->atom[root_atom].neighbor[j].id].subst_id =
              neighbor_subst_id;

/*
*       Edit neighbor substructure info
*/
        molecule->subst[neighbor_subst_id].inter_bonds = 1;
      }

      else
        exit (fprintf (global.outfile, "Unable to merge substructure %s %d.\n",
          molecule->subst[i].name, i + 1));
    }

/*
* Remove caps from atom and subst records
*/
  for (i = molecule->total.substs - 1; i >= 0; i--)
    if ((strstr (molecule->subst[i].sub_type, "AMN")) ||
      (strstr (molecule->subst[i].sub_type, "CXL")))
    {
/*
*     Reorder substructure ids stored in atom records
*/
      for (j = 0; j < molecule->total.atoms; j++)
        if (molecule->atom[j].subst_id > i)
          molecule->atom[j].subst_id--;

/*
*     Delete cap residues from substructure list
*/
      for (j = i; j < molecule->total.substs - 1; j++)
        molecule->subst[j] = molecule->subst[j + 1];
      molecule->total.substs--;
    }

/*
* Allocate space for use while shuffling molecular components
* 10/95 te
*/
  temporary.max = molecule->max;
  allocate_molecule (&temporary);
  copy_substs (&temporary, molecule);

/*
* Reshuffle substructures so that they are in ascending residue number order
* 10/95 te
*/
  ecalloc
  (
    (void **) &old2new,
    molecule->total.substs,
    sizeof (int),
    "old2new",
```



```
    global.outfile
  );

  emalloc
  (
    (void **) &order,
    molecule->total.subsets * sizeof (int),
    "order",
    global.outfile
  );

  for (i = 0; i < molecule->total.subsets; i++)
    order[i] = i;

  qsort (order, molecule->total.subsets, sizeof (int), compare_subsets);

  for (i = 0; i < molecule->total.subsets; i++)
  {
    old2new[order[i]] = i;

    copy_subset
      (&molecule->subset[i], &temporary.subset[order[i]]);
  }

  for (i = 0; i < molecule->total.atoms; i++)
    molecule->atom[i].subset_id = old2new[molecule->atom[i].subset_id];

  efree ((void **) &order);
  efree ((void **) &old2new);

/*
* Reshuffle atoms so that they are also in ascending residue number order.
* Also, report any charged residues.
* 10/95 te
*/
  ecalloc
  (
    (void **) &old2new,
    molecule->max.atoms,
    sizeof (int),
    "old2new",
    global.outfile
  );

  for (i = temporary.total.atoms = 0, molecule_charge = 0.0;
    i < molecule->total.subsets; i++)
  {
    for (j = 0, residue_charge = 0.0; j < molecule->total.atoms; j++)
      if (molecule->atom[j].subset_id == i)
      {
        old2new[j] = temporary.total.atoms;
        residue_charge += molecule->atom[j].charge;

        copy_atom
          (&temporary.atom[temporary.total.atoms],
          &molecule->atom[j]);

        temporary.atom[temporary.total.atoms].number =
          temporary.total.atoms + 1;

        copy_coord
          (temporary.coord[temporary.total.atoms],
          molecule->coord[j]);

        temporary.total.atoms++;
      }

    if (ABS (residue_charge) > 0.00001)
    {
      sprintf (line, "CHARGED RESIDUE %s", molecule->subset[i].name);
      fprintf (global.outfile, "%-40s: %8.3f\n", line, residue_charge);

      molecule_charge += residue_charge;
    }
  }

  sprintf (line, "Total charge on %s", molecule->info.name);
  fprintf (global.outfile, "\n%-40s: %8.3f\n", line, molecule_charge);

  copy_atoms (molecule, &temporary);

/*
* Update bond records to new atom numbering
* 10/95 te
*/
  for (i = 0; i < molecule->total.bonds; i++)
  {
    molecule->bond[i].origin = old2new[molecule->bond[i].origin];
    molecule->bond[i].target = old2new[molecule->bond[i].target];
  }

/*
* Update substructure records to new atom numbering
* 10/95 te
*/
  for (i = 0; i < molecule->total.subsets; i++)
    molecule->subset[i].root_atom = old2new[molecule->subset[i].root_atom];

  free_molecule (&temporary);
  efree ((void **) &old2new);

/*
* Reorder bond records also
* 10/95 te
*/
  for (i = 0; i < molecule->total.bonds; i++)
    if (molecule->bond[i].origin > molecule->bond[i].target)
    {
      j = molecule->bond[i].origin;
      molecule->bond[i].origin = molecule->bond[i].target;
      molecule->bond[i].target = j;
    }

  qsort (molecule->bond, molecule->total.bonds, sizeof (BOND), compare_bonds);

  atom_neighbors (molecule);

  return TRUE;
}

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

int compare_subsets (int *subset1, int *subset2)
{
  extern MOLECULE temporary;

  if (temporary.subset[*subset1].chain && temporary.subset[*subset2].chain)
  {
    if (!strcmp
      (temporary.subset[*subset1].chain, temporary.subset[*subset2].chain))
    {
      if (temporary.subset[*subset1].number >
        temporary.subset[*subset2].number)
        return 1;

      else
        return -1;
    }

    else
      return strcmp
        (temporary.subset[*subset1].chain, temporary.subset[*subset2].chain);
  }

  else
  {
    if (temporary.subset[*subset1].number >
      temporary.subset[*subset2].number)
      return 1;

    else
      return -1;
  }
}

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

int compare_bonds (BOND *bond1, BOND *bond2)
{
  if (bond1->origin == bond2->origin)
  {
    if (bond1->target > bond2->target)
      return 1;

    else
      return -1;
  }

  else if (bond1->origin > bond2->origin)
    return 1;

  else
    return -1;
}
```

```
##############################################################################
#################################### IO_SPH.H ################################
##############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

/* Routines used to read and write molecule data */

int read_sph
(
  MOLECULE  *molecule,
  FILE_NAME molecule_file_name,
  FILE      *molecule_file
);

int write_sph
(
  MOLECULE  *molecule,
  FILE      *molecule_file
);

##############################################################################
#################################### IO_SPH.C ################################
##############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "io_sph.h"

int read_sph
(
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file
)
{
  int i;
  int flag;
  char *line = NULL;
  int return_value = TRUE;
  STRING20 temp;

  if (find_record (&line, "cluster", molecule_file) == NULL)
    return EOF;

  vstrcpy (&molecule->info.name, "site_points");
  vstrcpy (&molecule->info.comment, molecule_file_name);

  memset (temp, 0, sizeof (STRING20));
  molecule->total.atoms = atoi (strncpy (temp, &line[45], 5));

/*
* Allocate space for molecule components
* 2/96 te
*/
  reallocate_atoms (molecule);

/*
* Store site point information
* 6/95 te
*/
  for (i = 0; i < molecule->total.atoms; i++)
  {
/*
*   Read in line of file
*   6/95 te
*/
```

SF LIBRARY

```
    if (vfgets (&line, molecule_file) == NULL)
      exit (fprintf (global.outfile,
        "ERROR read_sph. Insufficient sphere records in %s\n",
        molecule_file_name));

    if (strlen (line) < 52)
      exit (fprintf (global.outfile,
        "ERROR read_sph: Sphere file data line too short in %s\n%s",
        molecule_file_name, line));

    molecule->atom[i].number = i + 1;

    memset (temp, 0, sizeof (STRING20));
    sscanf (&line[0], "%s", temp);
    vstrcpy (&molecule->atom[i].name, temp);

    vstrcpy (&molecule->atom[i].type, "Du");

    memset (temp, 0, sizeof (STRING20));
    molecule->coord[i][0] = atof (strncpy (temp, &line[5], 10));

    memset (temp, 0, sizeof (STRING20));
    molecule->coord[i][1] = atof (strncpy (temp, &line[15], 10));

    memset (temp, 0, sizeof (STRING20));
    molecule->coord[i][2] = atof (strncpy (temp, &line[25], 10));

    memset (temp, 0, sizeof (STRING20));
    molecule->atom[i].charge = atof (strncpy (temp, &line[35], 8));

    memset (temp, 0, sizeof (STRING20));
    molecule->atom[i].subst_id = atoi (strncpy (temp, &line[48], 2)) - 1;

    memset (temp, 0, sizeof (STRING20));
    molecule->atom[i].chem_id = atoi (strncpy (temp, &line[50], 3));
  }

/*
* Convert zero-value critical id numbers to one above the maximum
* 10/95 te
*/
  for (i = 0, flag = FALSE; i < molecule->total.atoms; i++)
    if (molecule->atom[i].subst_id + 1 > molecule->total.substs)
      molecule->total.substs = molecule->atom[i].subst_id + 1;

  for (i = 0, flag = FALSE; i < molecule->total.atoms; i++)
    if (molecule->atom[i].subst_id < 0)
    {
      flag = TRUE;
      molecule->atom[i].subst_id = molecule->total.substs;
    }

  if (flag)
    molecule->total.substs++;

/*
* Allocate substructure records
* 10/95 te
*/
  reallocate_substs (molecule);

  for (i = 0; i < molecule->total.substs; i++)
  {
    molecule->subst[i].number = i + 1;
    vstrcpy (&molecule->subst[i].name, "SPH");
  }

  efree ((void **) &line);
  return return_value;
}


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

int write_sph
(
  MOLECULE *molecule,
  FILE *molecule_file
)
{
  int i, j;
  STRING80 line;

/*
* Write header
* 6/95 te
*/
  fprintf (molecule_file, "cluster %5d   ", molecule->info.output_id);
  fprintf (molecule_file, "number of spheres in cluster %5d\n",
    molecule->total.atoms);

/*
* Write out atoms
* 6/95 te
*/
  for (i = 0; i < molecule->total.atoms; i++)
  {
/*
*   Print out site point info
*   6/95 te
*/
    memset (line, 0, sizeof (STRING80));

    if (molecule->atom[i].name)
      sprintf (&line[0], "%5d", i + 1);

    else
      sprintf (&line[0], "%5d", i + 1);

    sprintf (&line[5], "%10.5f", molecule->coord[i][0]);
    sprintf (&line[15], "%10.5f", molecule->coord[i][1]);
    sprintf (&line[25], "%10.5f", molecule->coord[i][2]);
    sprintf (&line[35], "%8.3f", molecule->atom[i].charge);
    sprintf (&line[48], "%2d", molecule->atom[i].subst_id + 1);
    sprintf (&line[50], "%3d", molecule->atom[i].chem_id);

/*
*   Replace null characters with spaces
*   6/95 te
*/
    for (j = 0; j < sizeof (STRING80); j++)
      if (line[j] == '\0') line[j] = ' ';
    line[sizeof (STRING80) - 2] = '\0';

    fprintf (molecule_file, "%s\n", line);
  }

  return TRUE;
}
```

```
********************************************************************************
************************************ LABEL.H ***********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/
#include "label_node.h"
#include "label_vdw.h"
#include "label_chem.h"
#include "label_flex.h"

/* Structures to store atom labelling definitions */

typedef struct label_struct
{
  LABEL_VDW vdw;
  LABEL_CHEMICAL chemical;
  LABEL_FLEX flex;

  FILE_NAME definition_path;

} LABEL;


void free_labels (LABEL *);
********************************************************************************
************************************ LABEL.C ***********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

#include "define.h"
#include "mol.h"
#include "label.h"

void free_labels (LABEL *label)
{
  free_vdw_labels (&label->vdw);
  free_chemical_labels (&label->chemical);
  free_flex_labels (&label->flex);
}

********************************************************************************
********************************* LABEL_CHEM.H *********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

/* Structures to store atom labelling definitions */

typedef struct chemical_member_struct
{
  STRING20        name;                 /* Member name */
  NODE            *definition;          /* Member definitions */
  int             definition_total;     /* Number of definitions */

  float           radius;               /* Interaction radius */
  float           tolerance;            /* Interaction tolerance */

} CHEMICAL_MEMBER;

typedef struct chemical_screen_struct
{
  int             flag;                 /* Flag for chemical screen */
  int             process_flag;         /* Flag to screen ligands */
  int             construct_flag;       /* Flag for keying database */
  int             pharmaco_flag;        /* Flag for pharmacophore search */
  int             similar_flag;         /* Flag for similarity search */
  float           dissimilar_maximum;   /* Similarity cutoff */
  int             fold_flag;            /* Flag for folding label keys */
  float           distance_minimum;     /* Minimum distance of interest */
  float           distance_maximum;     /* Maximum distance of interest */
  float           distance_interval;    /* Distance interval between keys */
  int             interval_total;       /* Number of distance keys */

} CHEMICAL_SCREEN;

typedef struct label_chemical_struct
{
  int             flag;                 /* Flag for chemical labeling */
  int             init_flag;            /* Flag for chem label initialization */
  CHEMICAL_MEMBER *member;              /* Label members */
  int             total;                /* Number of members */
  NODE            *definition;          /* Label definitions */

  CHEMICAL_SCREEN screen;               /* Chemical screen parameters */

  float           **match_table;        /* Match compatibility table */
  float           **score_table;        /* Score weight table */
  float           **screen_table;       /* Screen weight table */

  FILE_NAME       file_name;            /* File containing label data */
  FILE_NAME       match_file_name;      /* File with match table */
  FILE_NAME       score_file_name;      /* File with score table */
  FILE_NAME       screen_file_name;     /* File with screen table */

} LABEL_CHEMICAL;

/* Routines to manipulate atom label structures */

void get_chemical_labels        (LABEL_CHEMICAL *);
void free_chemical_labels       (LABEL_CHEMICAL *);
int  assign_chemical_labels     (LABEL_CHEMICAL *, MOLECULE *);
void read_chemical_labels       (LABEL_CHEMICAL *, MOLECULE *,
                                FILE_NAME, FILE *);
void get_table                  (LABEL_CHEMICAL *, FILE_NAME, float ***);
void free_table                 (LABEL_CHEMICAL *, float ***);

********************************************************************************
********************************* LABEL_CHEM.C *********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */
```

```c
/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "label_node.h"
#include "label_chem.h"

/* ================================================================ */
void get_chemical_labels (LABEL_CHEMICAL *label_chemical)
{
  int i, definition_total, definition_count;
  int values_read;
  STRING100 line;
  FILE *chemical_file;

  label_chemical->init_flag = TRUE;

  chemical_file = efopen (label_chemical->file_name, "r", global.outfile);

/*
* Count up the number of chemical label declarations and definitions,
* then allocate memory
* 6/95 te
*/
  label_chemical->total = 1;
  definition_count = 0;
  while (fgets (line, 100, chemical_file) != NULL)
  {
    if (!strncmp (line, "name", 4))
      label_chemical->total++;
    if (!strncmp (line, "definition", 10))
      definition_count++;
  }
  rewind (chemical_file);
  definition_total = definition_count;

  ecalloc
  (
    (void **) &label_chemical->member,
    label_chemical->total,
    sizeof (CHEMICAL_MEMBER),
    "chemical labels",
    global.outfile
  );

  ecalloc
  (
    (void **) &label_chemical->definition,
    definition_count,
    sizeof (NODE),
    "chemical label definitions",
    global.outfile
  );

/*
* Read in the atom label definitions
* 6/95 te
*/
  strcpy (label_chemical->member[0].name, "null");
  label_chemical->member[0].definition_total = 0;
  label_chemical->member[0].definition = &label_chemical->definition[0];
  strcpy (label_chemical->member[0].definition[0].type, "*");
  label_chemical->member[0].radius = label_chemical->member[0].tolerance = 0;

  label_chemical->total = 1;
  definition_count = definition_total = 0;
  while (fgets (line, 100, chemical_file) != NULL)
  {
/*
*   Process chemical label declaration
*   6/95 te
*/
    if (!strncmp (line, "name", 4))
    {
      label_chemical->member[label_chemical->total - 1].definition_total =
        definition_total;
      label_chemical->member[label_chemical->total].definition =
        &label_chemical->definition[definition_count];
      definition_total = 0;

      if (sscanf
        (line, "%*s %s", label_chemical->member[label_chemical->total].name) <
        1)
        exit (fprintf (global.outfile,
          "ERROR get_chemical_labels: Incomplete label declaration in %s\n",
          label_chemical->file_name));
/*
*     Convert label_chemical->member name to lowercase
*     6/95 te
*/
      for (i = 0; i < strlen
        (label_chemical->member[label_chemical->total].name); i++)
        label_chemical->member[label_chemical->total].name[i] =
          (char) tolower
            (label_chemical->member[label_chemical->total].name[i]);

/*
*     Make sure that "null" label was not in input
*     6/95 te
*/
      if (!strncmp
        (label_chemical->member[label_chemical->total].name, "null", 7))
        exit (fprintf (global.outfile,
          "ERROR get_chemical_labels: <null> specified if %s\n",
          label_chemical->file_name));

      label_chemical->total++;
    }

/*
*   Process label_chemical->member radius
*   10/95 te
*/
    else if (!strncmp (line, "radius", 5))
    {
      values_read = sscanf (line, "%*s %f %f",
        &label_chemical->member[label_chemical->total - 1].radius,
        &label_chemical->member[label_chemical->total - 1].tolerance);

      if (values_read < 1)
        exit (fprintf (global.outfile,
          "ERROR get_chemical_labels: Incomplete radius specification in %s\n",
          label_chemical->file_name));

      if (values_read < 2)
        label_chemical->member[label_chemical->total - 1].tolerance = 0.0;
    }

/*
*   Process label_chemical->member definition
*   6/95 te
*/
    else if (!strncmp (line, "definition", 10))
    {
      strtok (white_line (line), " ");

      if (!assign_node (&label_chemical->definition[definition_count],
        TRUE))
        exit (fprintf (global.outfile,
          "ERROR get_chemical_labels: Improper label definition in %s\n",
          label_chemical->file_name));

      label_chemical->definition[definition_count].weight = 1.0;
      definition_count++;
      definition_total++;
    }

    else if (!strncmp (line, "weight", 6))
      if (sscanf (line, "%*s %f",
        &label_chemical->definition[definition_count - 1].weight) < 1)
        label_chemical->definition[definition_count - 1].weight = 1.0;
  }

/*
* Update last label_chemical->member info also
* 6/95 te
*/
  label_chemical->member[label_chemical->total - 1].definition_total =
    definition_total;

  efclose (&chemical_file);


/*
* Print out the label_chemical->members and their definitions
* 6/95 te

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "\n____Chemical_Label_Definitions____\n\n");
    for (i = 0; i < label_chemical->total; i++)
    {
      MFPUTC ('_', 40);
      fprintf (global.outfile, "\n");
      fprintf (global.outfile, "%-20s%s\n", "name",
        label_chemical->member[i].name);
      fprintf (global.outfile, "%-20s%-7.2f", "radius",
        label_chemical->member[i].radius);
      if (label_chemical->member[i].tolerance)
        fprintf (global.outfile, "%-7.2f", label_chemical->member[i].tolerance);
      fprintf (global.outfile, "\n");

      for (j = 0; j < label_chemical->member[i].definition_total; j++)
      {
        fprintf (global.outfile, "%-20s", "definition");
        print_node (&label_chemical->member[i].definition[j], 0);
        fprintf (global.outfile, "\n");
        fprintf (global.outfile, "%-20s%-7.2f\n", "weight",
            label_chemical->member[i].definition[j].weight);
        fprintf (global.outfile, "\n");
      }
      fprintf (global.outfile, "\n");
    }
    fprintf (global.outfile, "\n\n");
  }
*/
}


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

void free_chemical_labels (LABEL_CHEMICAL *label_chemical)
{
  int i, j;

  for (i = 0; i < label_chemical->total; i++)
    for (j = 0; j < label_chemical->member[i].definition_total; j++)
      free_node (&label_chemical->member[i].definition[j]);

  efree ((void **) &label_chemical->member);
  efree ((void **) &label_chemical->definition);
}


/* ================================================================ */

void get_table
(
  LABEL_CHEMICAL        *label_chemical,
  FILE_NAME             table_file_name,
  float                 ***table
)
{
  int i, j;
  int continue_loop;
  STRING20 receptor_axis = "RECEPTOR", value;
  FILE *table_file;
  STRING100 line;
  STRING20 *table_label = NULL;
  int label_count, label_match, *label_conversion = NULL;
  char *token, *token_arg;

  if (!label_chemical->init_flag)
    get_chemical_labels (label_chemical);

/*
* Allocate memory (and set elements to zero) for chemical matching table
* 6/95 te
*/
  ecalloc
  (
    (void **) table,
    label_chemical->total,
    sizeof (float *),
    "chemical table",
    global.outfile
  );

  for (i = 0; i < label_chemical->total; i++)
    ecalloc
    (
      (void **) &(*table)[i],
      label_chemical->total,
      sizeof (float),
```

UCSF LIBRARY

```
      "chemical table",
      global.outfile
    );

/*
* Read in interaction table
* 10/95 te
*/
  table_file = efopen (table_file_name, "r", global.outfile);

  for (label_count = 0; fgets (line, 100, table_file);)
    if (!strncmp (line, "label", 5))
      label_count++;

  rewind (table_file);

  emalloc
  (
    (void **) &label_conversion,
    label_count * sizeof (int),
    "label conversion array",
    global.outfile
  );

  ecalloc
  (
    (void **) &table_label,
    label_count,
    sizeof (STRING20),
    "label name array",
    global.outfile
  );

  for (label_count = 0; fgets (line, 100, table_file);)
    if (!strncmp (line, "label", 5))
    {
      if (sscanf (line, "%*s %s", table_label[label_count]))
      {
        for (i = 0, label_match = FALSE; i < label_chemical->total; i++)
        {
          if (!strcmp
            (table_label[label_count], label_chemical->member[i].name))
          {
            label_conversion[label_count] = i;
            label_match = TRUE;
          }
        }

        if (label_match)
          label_count++;

        else
          exit (fprintf (global.outfile,
            "ERROR get_table: %s not in definition file %s\n",
            table_label[label_count], table_file_name));
      }

      else
        exit (fprintf (global.outfile,
          "ERROR get_table. Empty line in definition file %s\n",
          table_file_name));
    }

  rewind (table_file);

  while (strncmp (line, "table", 5))
    if (!fgets (line, 100, table_file))
      exit (fprintf (global.outfile,
        "ERROR get_table: No table found in %s\n", table_file_name));

  for (i = 0; i < label_count; i++)
  {
    if (!fgets (line, 100, table_file))
      exit (fprintf (global.outfile,
        "ERROR get_table: Table empty in %s\n", table_file_name));

/*
*   Replace tab characters in input line with spaces
*   6/95 te
*/
    white_line (line);

    for (j = 0, token_arg = line; j <= i; j++, token_arg = NULL)
    {
      if (token = strtok (token_arg, " "))
        (*table)[label_conversion[i]][label_conversion[j]] =
          (*table)[label_conversion[j]][label_conversion[i]] =
            atof (token);

      else
        exit (fprintf (global.outfile,
          "ERROR get_table: Insufficient table entries in %s\n",
          table_file_name));
    }
  }

  fclose (table_file);

  efree ((void *) &label_conversion);
  efree ((void *) &table_label);

/*
* Print out match table
* 6/95 te
*/

  fprintf (global.outfile, "\n___________Chemical_Table___________\n\n");


  fprintf (global.outfile, "    _");
  for (i = 0; i < label_chemical->total; i++)
    fprintf (global.outfile, "%6s", "______");
  fprintf (global.outfile, "_\n");

  fprintf (global.outfile, "  | ");
  for (i = 0; i < label_chemical->total; i++)
    fprintf (global.outfile, "%6s", "");
  fprintf (global.outfile, " |  LIGAND\n");

  for (i = 0; i < label_chemical->total; i++)
  {
    fprintf (global.outfile, "  | ");
    for (j = 0; j < label_chemical->total; j++)
    {
      sprintf (value, "%6.2f", (*table)[i][j]);

      if (value[5] == '0')
      {
        value[5] = ' ';

        if (value[4] == '0')
        {
          value[4] = ' ';
          value[3] = ' ';

          if (atoi (value) == 0)
            strcpy (value, "");
        }
      }

      fprintf (global.outfile, "%6s", value);
    }

    fprintf (global.outfile, " |  %s\n", label_chemical->member[i].name);
  }

  fprintf (global.outfile, "  |_");
  for (i = 0; i < label_chemical->total; i++)
    fprintf (global.outfile, "%6s", "______");
  fprintf (global.outfile, "_|\n\n");

  for (i = 0, continue_loop = TRUE;
    (i < sizeof (STRING20)) && continue_loop; i++)
  {
    if (receptor_axis[i])
    {
      fprintf (global.outfile, "  %c", receptor_axis[i]);
      continue_loop = TRUE;
    }
    else
      fprintf (global.outfile, "%3s", "");

    for (j = continue_loop = 0; j < label_chemical->total; j++)
    {
      if (label_chemical->member[j].name[i])
      {
        fprintf (global.outfile, "   %c  ", label_chemical->member[j].name[i]);
        continue_loop = TRUE;
      }
      else
        fprintf (global.outfile, "%6s", "");
    }
    fprintf (global.outfile, "\n");
  }

  fprintf (global.outfile, "\n\n");
}


/* ============================================================== */

void free_table
(
  LABEL_CHEMICAL        *label_chemical,
  float                 ***table
)
{
  int i;

  for (i = 0; i < label_chemical->total; i++)
    efree ((void **) &(*table)[i]);

  efree ((void **) table);
}


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

int assign_chemical_labels
(
  LABEL_CHEMICAL *label_chemical,
  MOLECULE *molecule
)
{
  int i, j, k;

  if (!label_chemical->init_flag)
    get_chemical_labels (label_chemical);

  molecule->info.assign_chem = TRUE;

  for (i = 0; i < molecule->total.atoms; i++)
    molecule->atom[i].chem_id = 0;

  for (i = 0; i < molecule->total.atoms; i++)
    for (j = 1; j < label_chemical->total; j++)
      for (k = 0; k < label_chemical->member[j].definition_total; k++)
        if (check_atom
          (molecule, i, &label_chemical->member[j].definition[k]))
          molecule->atom[i].chem_id = j;

/*
* Print out chemical label assignments
* 10/95 te
*/
  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "____Chemical_Assignments____\n");
    fprintf (global.outfile, "%s:\n\n", molecule->info.name);
    for (i = 0; i < molecule->total.atoms; i++)
      fprintf (global.outfile, "%6d %4s %5s   %-20s\n", i + 1,
        molecule->atom[i].name,
        molecule->atom[i].type,
        label_chemical->member[molecule->atom[i].chem_id].name);
    fprintf (global.outfile, "\n\n");
  }

  return TRUE;
}


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

void read_chemical_labels
(
  LABEL_CHEMICAL *label_chemical,
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file
)
{
  int i, j;
  long file_position;
  int label_count, label_match, *label_conversion = NULL;
  char buff[200];

/*
* Record current file position
* 2/96 te
```

```
*/
  file_position = ftell (molecule_file);
  rewind (molecule_file);

/*
* Read in header line of file
* 6/95 te
*/
  if (!fgets (buff, 199, molecule_file))
    exit (fprintf (global.outfile,
      "ERROR read_chemical_labels: No data in SPH file %s\n",
      molecule_file_name));

  if (!strncmp (buff, "cluster", 7))
    exit (fprintf (global.outfile,
      "ERROR read_chemical_labels: No chemical labels are in SPH file %s\n",
      molecule_file_name));

/*
* Construct list to convert chemical labels read here to proper
* chem_id values
* 6/95 te
*/
  ecalloc
  (
    (void **) &label_conversion,
    label_chemical->total,
    sizeof (int),
    "chemical label conversion list",
    global.outfile
  );

  for (i = 1; i < label_chemical->total; i++)
    label_conversion[i] = -1;

  if (!fgets (buff, 199, molecule_file))
    exit (fprintf (global.outfile,
      "ERROR read_chemical_labels: No chemical labels are in SPH file %s\n",
      molecule_file_name));

  label_count = 0;
  while (strncmp (buff, "cluster", 7))
  {
    label_match = FALSE;

    for (i = 0; i < label_chemical->total; i++)
      if (!strncmp
        (buff, label_chemical->member[i].name,
        strlen (label_chemical->member[i].name)))
      {
        label_conversion[i] = ++label_count;
        label_match = TRUE;
        break;
      }

    if (!label_match)
      exit (fprintf (global.outfile,
        "ERROR read_chemical_labels: No definition for %s in SPH file %s\n",
        buff, molecule_file_name));

    if (!fgets (buff, 199, molecule_file))
      exit (fprintf (global.outfile,
        "ERROR read_chemical_labels: Insufficient labels in SPH file %s\n",
        molecule_file_name));
  }

  for (i = 0; i < molecule->total.atoms; i++)
  {
/*
*   Convert chemical label integer
*   6/95 te
*/
    if (label_chemical->total && label_conversion)
    {
      if (molecule->atom[i].chem_id > label_count + 1)
        exit (fprintf (global.outfile,
          "ERROR read_chemical_labels: "
          "Improper chemical label for entry %s in %s\n",
          molecule->atom[i].name, molecule_file_name));

      for (j = 0; j < label_chemical->total; j++)
        if (label_conversion[j] == molecule->atom[i].chem_id)
        {
          molecule->atom[i].chem_id = j;
          break;
        }
    }
  }

  efree ((void **) &label_conversion);

/*
* Return to original file position
* 2/96 te
*/
  fseek (molecule_file, file_position, SEEK_SET);
}
```

```
##############################################################################
################################ LABEL_FLEX.H ################################
##############################################################################
```


```
/*                                                                       */
/*                       Copyright UCSF, 1997                            */
/*                                                                       */
```

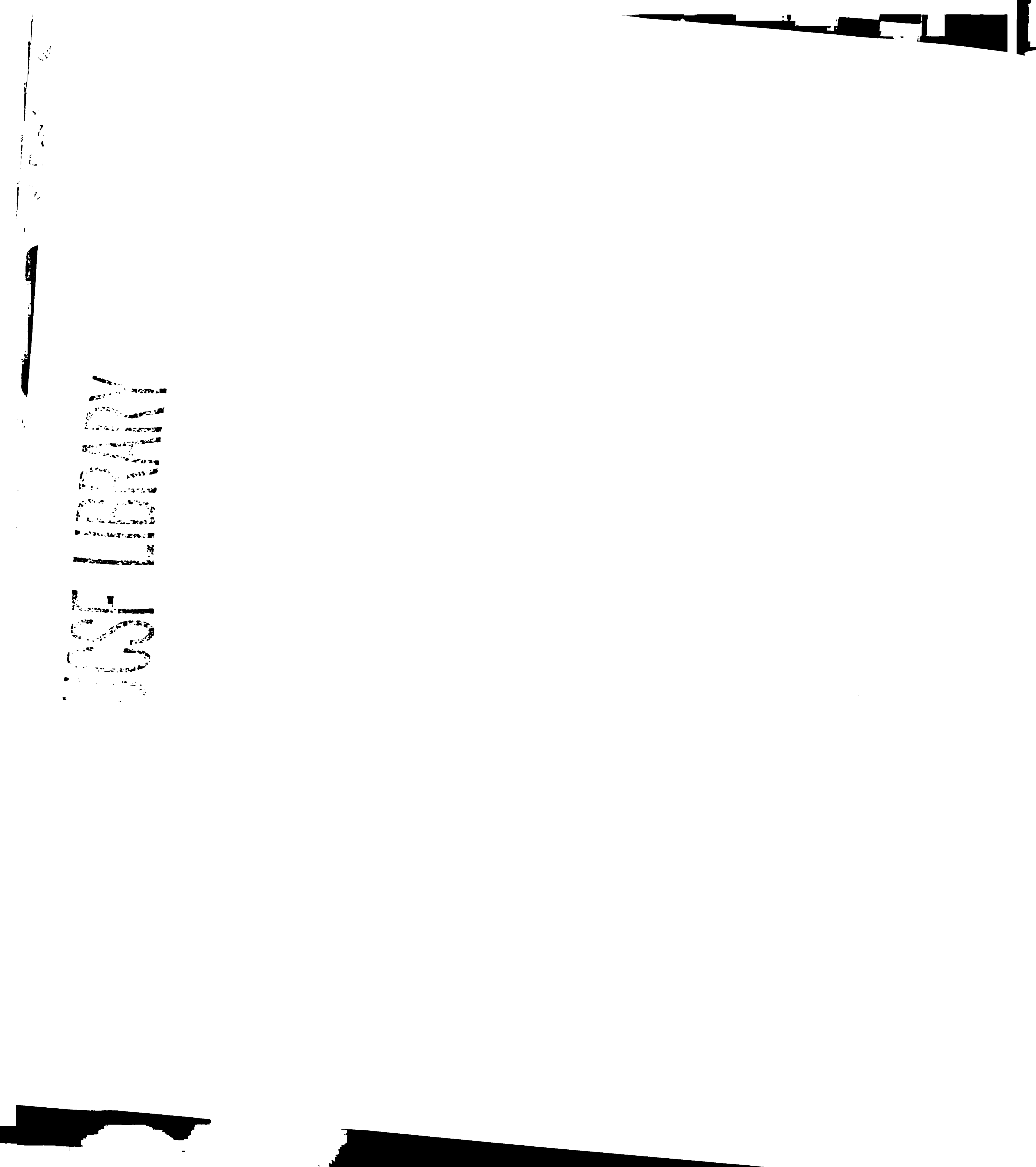

```
/*
Written by Todd Ewing
10/96
*/

/* Structures to store atom labelling definitions */

typedef struct flex_member_struct
{
  STRING20      name;                   /* Member name */
  NODE          *definition;            /* Member definitions */
  int           definition_total;       /* Number of definitions */

  int           drive_id;               /* Torsion search label */
  int           torsion_total;          /* Number of torsion positions */
  float         *torsion;               /* Torsion values */
  int           minimize_flag;          /* Flag for rotation during minim'n */
  float         penalty;                /* Penalty value (unused) */

} FLEX_MEMBER;

typedef struct label_flex_struct
{
  int           flag;                   /* Flag for flex labeling */
  int           init_flag;              /* Flag for flex label initialization */

  int           drive_flag;             /* Flag for torsion driver search */
  float         clash_overlap;          /* Clash vdw overlap threshold */
  int           max_conforms;           /* Maximum conformations per layer */

  int           minimize_flag;          /* Flag for torsional minimization */

  int           anchor_flag;            /* Flag for anchor search */
  int           multiple_anchors;       /* Flag for multiple anchors */
  int           anchor_size;            /* Minimum size of anchor segment */

  int           periph_flag;            /* Flag for peripheral search */
  int           periph_seeds;           /* Number of seed layer conformations */
  int           minimize_anchor_flag;   /* Flag to minimize anchor */
  int           reminimize_layers;      /* Previous layers to reminimize */
  int           reminimize_anchor_flag; /* Flag to reminimize anchor */
  int           reminimize_ligand_flag; /* Flag to reminimize ligand */

  int           max_torsions;           /* Maximum torsions per molecule */

  FLEX_MEMBER   *member;                /* Label members */
  int           total;                  /* Number of label members */
  NODE          *definition;            /* Member definitions */
  FILE_NAME     file_name;              /* File containing flex data */
  FILE_NAME     search_file_name;       /* File containing search data */

} LABEL_FLEX;

/* Routines to manipulate atom label structures */

int get_flex_labels (LABEL_FLEX *flex);

void assign_flex_labels
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule
);

void free_flex_labels (LABEL_FLEX *label_flex);

int get_flex_search (LABEL_FLEX *);

int check_peripheral_torsion
(
  MOLECULE      *molecule,
  int           torsion_id
);

int get_anchor
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *mol_init,
  MOLECULE      *mol_anch,
  int           anchor
);

void get_segments (MOLECULE *mol_init);

int get_anchor_segment
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule,
  int           anchor_count
);

void get_layers
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *mol_anch
);

void get_single_anchor
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *mol_anch
);

void initialize_segments
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *mol_anch
);

void initialize_anchor_layer
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule
);

void initialize_layer
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule,
  int           layer
);
```

```
##############################################################################
################################ LABEL_FLEX.C ################################
##############################################################################
```


```
/*                                                                       */
/*                       Copyright UCSF, 1997                            */
/*                                                                       */
```


```
/*
Written by Todd Ewing
12/96
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label_node.h"
#include "label_vdw.h"
#include "label_flex.h"
#include "transform.h"


/* ===================================================================== */
int get_flex_labels (LABEL_FLEX *label_flex)
{
  int i, definition_total, definition_count;
  STRING100 line;
  FILE *flex_file;

  label_flex->init_flag = TRUE;
```

```
  flex_file = efopen (label_flex->file_name, "r", global.outfile);

/*
* Count up the number of flex label declarations and definitions,
* then allocate memory
* 6/95 te
*/
  for (label_flex->total = definition_count = 0; fgets (line, 100, flex_file);)
  {
    if (!strncmp (line, "name", 4))
      label_flex->total++;
    if (!strncmp (line, "definition", 10))
      definition_count++;
  }

  rewind (flex_file);

  definition_total = definition_count;

  ecalloc
  (
    (void **) &label_flex->member,
    label_flex->total + 1,
    sizeof (FLEX_MEMBER),
    "flexible bond labels",
    global.outfile
  );

  ecalloc
  (
    (void **) &label_flex->definition,
    definition_total,
    sizeof (NODE),
    "flexible bond label definitions",
    global.outfile
  );

  strcpy (label_flex->member[0].name, "null");

/*
* Read in the atom label definitions
* 6/95 te
*/
  for (label_flex->total = 1, definition_count = definition_total = 0;
    fgets (line, 100, flex_file);)
  {
/*
*   Process flexible label declaration
*   6/95 te
*/
    if (!strncmp (line, "name", 4))
    {
      if (label_flex->total > 1)
      {
        if (label_flex->member[label_flex->total - 1].drive_id == NEITHER)
          exit (fprintf (global.outfile,
            "ERROR get_flex_labels: Missing drive_id parameter in %s\n",
            label_flex->file_name));

        else if (label_flex->member[label_flex->total - 1].minimize_flag ==
          NEITHER)
          exit (fprintf (global.outfile,
            "ERROR get_flex_labels: Missing minimize parameter in %s\n",
            label_flex->file_name));

        else if (definition_total != 2)
          exit (fprintf (global.outfile,
            "ERROR get_flex_labels: Incorrect number of definitions in %s\n",
            label_flex->file_name));
      }

      label_flex->member[label_flex->total - 1].definition_total =
        definition_total;
      label_flex->member[label_flex->total].definition =
        &label_flex->definition[definition_count];
      definition_total = 0;

      if (sscanf
        (line, "%*s %s", label_flex->member[label_flex->total].name) < 1)
      {
        fprintf (global.outfile,
          "Incomplete label_flex->member declaration.\n");
        exit (EXIT_FAILURE);
      }

/*
*     Convert flex label name to lowercase
*     6/95 te
*/
      for (i = 0; i < strlen (label_flex->member[label_flex->total].name); i++)
        label_flex->member[label_flex->total].name[i] =
          (char) tolower (label_flex->member[label_flex->total].name[i]);

      label_flex->total++;

      label_flex->member[label_flex->total - 1].drive_id = NEITHER;
      label_flex->member[label_flex->total - 1].minimize_flag = NEITHER;
    }

/*
*   Process label_flex->member search identifier
*   10/95 te
*/
    else if (!strncmp (line, "drive_id", 6))
      sscanf
        (line, "%*s %d", &label_flex->member[label_flex->total - 1].drive_id);

/*
*   Process flex label minimize flag
*   10/95 te
*/
    else if (!strncmp (line, "minimize", 8))
      sscanf
        (line, "%*s %d",
        &label_flex->member[label_flex->total - 1].minimize_flag);

/*
*   Process flex label definition
*   6/95 te
*/
    else if (!strncmp (line, "definition", 10))
    {
      strtok (white_line (line), " ");

      if (!assign_node
        (&label_flex->definition[definition_count], TRUE))
      {
        fprintf (global.outfile, "Error assigning flex label definitions \n");
        exit (EXIT_FAILURE);
      }

      definition_count++;
      definition_total++;
    }
  }

/*
* Update last flex label info also
* 6/95 te
*/
  label_flex->member[label_flex->total - 1].definition_total =
    definition_total;

  efclose (&flex_file);

/*
* Assign search torsion parameters if requested
* 10/96 te
*/
  if (label_flex->drive_flag)
    get_flex_search (label_flex);

/*
* Print out the flex label and their definitions
* 6/95 te

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "\n____Flexible_Bond_Label_Definitions____\n\n");
    for (i = 0; i < label_flex->total; i++)
    {
      fprintf (global.outfile, "\n");
      fprintf (global.outfile, "%-20s%s\n", "name",
        label_flex->member[i].name);
      fprintf (global.outfile, "%-20s%d\n", "drive_id",
        label_flex->member[i].drive_id);
      fprintf (global.outfile, "%-20s%d\n", "minimize",
        label_flex->member[i].minimize_flag);

      fprintf (global.outfile, "%-20s%d\n", "positions",
        label_flex->member[i].torsion_total);

      fprintf (global.outfile, "%-20s", "torsions");
      for (j = 0; j < label_flex->member[i].torsion_total; j++)
        fprintf (global.outfile, "%g ", label_flex->member[i].torsion[j]);

      fprintf (global.outfile, "\n");
      for (j = 0; j < label_flex->member[i].definition_total; j++)
      {
        fprintf (global.outfile, "%-20s", "definition");
        print_node (&label_flex->member[i].definition[j], 0);
        fprintf (global.outfile, "\n");
      }

      fprintf (global.outfile, "\n");
    }

    fprintf (global.outfile, "\n\n");
  }
*/

  return TRUE;
}


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

void free_flex_labels (LABEL_FLEX *label_flex)
{
  int i, j;

  for (i = 0; i < label_flex->total; i++)
  {
    efree ((void **) &label_flex->member[i].torsion);

    for (j = 0; j < label_flex->member[i].definition_total; j++)
      free_node (&label_flex->member[i].definition[j]);
  }

  efree ((void **) &label_flex->member);
  efree ((void **) &label_flex->definition);
}


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

Program to read in torsion.defn
11/95 Yax Sun
10/96 edited by Todd Ewing

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

int get_flex_search (LABEL_FLEX *label_flex)
{
  int i, j;                     /* Counter variables */
  STRING100 line;               /* String used to compose output */
  char *token;
  FILE  *search_file;

  int   drive_id;
  int   torsion_total;
  float *torsion = NULL;        /* temporarily hold the torsion values */

  search_file = efopen (label_flex->search_file_name, "r", global.outfile);

  while (fgets (line, 100, search_file))
  {
    token = strtok (white_line (line), " ");

    if (!strcmp (token, "drive_id"))
    {
      if (token = strtok (NULL, " "))
        drive_id = atoi (token);

      else
        exit (fprintf (global.outfile,
          "ERROR get_flex_search: Search_id value not specified in %s\n",
          label_flex->search_file_name));

      if (!fgets (line, 100, search_file) ||
        !(token = strtok (white_line (line), " ")) ||
        strcmp (token, "positions"))
        exit (fprintf (global.outfile,
          "ERROR get_flex_search: Positions field doesn't follow Id in %s\n",
          label_flex->search_file_name));

      if (token = strtok (NULL, " "))
        torsion_total = atoi (token);
```



```
      else
        exit (fprintf (global.outfile,
          "ERROR get_flex_search: Postions value not specified in %s\n",
          label_flex->search_file_name));

      ecalloc
      (
        (void **) &torsion,
        torsion_total,
        sizeof (float),
        "torsion sample values",
        global.outfile
      );

      if (!fgets (line, 100, search_file) ||
        !(token = strtok (white_line (line), " ")) ||
        strcmp (token, "torsions"))
        exit (fprintf (global.outfile,
          "ERROR get_flex_search: Torsions doesn't follow Positions in %s\n",
          label_flex->search_file_name));

      for (i = 0; i < torsion_total; i++)
      {
        if (token = strtok (NULL, " "))
          torsion[i] = atof (token);

        else
          exit (fprintf (global.outfile,
            "ERROR get_flex_search. Insufficient number of torsions in %s\n",
            label_flex->search_file_name));
      }

      for (i = 0; i < label_flex->total; i++)
      {
        if (label_flex->member[i].drive_id == drive_id)
        {
          label_flex->member[i].torsion_total = torsion_total;

          ecalloc
          (
            (void **) &label_flex->member[i].torsion,
            label_flex->member[i].torsion_total,
            sizeof (float),
            "stored torsion sample values",
            global.outfile
          );

          for (j = 0; j < label_flex->member[i].torsion_total; j++)
            label_flex->member[i].torsion[j] = torsion[j];

/*
          fprintf (global.outfile,
            "search %d, %s\n", drive_id, label_flex->member[i].name);
*/
        }
      }

      torsion_total = 0;
      efree ((void **) &torsion);
    }
  }

  for (i = 1; i < label_flex->total; i++)
    if (label_flex->member[i].torsion_total < 1)
      exit (fprintf (global outfile,
        "ERROR get_flex_search: Missing torsion parameters in %s\n",
        label_flex->search_file_name));

  fclose (search_file);
  return TRUE;
}


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

void assign_flex_labels
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule
)
{
  int i, j;
  int bond_id, bond_found;
  int flex_id;
  int conf_total;
  static MOLECULE temporary = {0};

  void detect_rings (MOLECULE *);

  reset_molecule (&temporary);

  if (!label_flex->init_flag)
    get_flex_labels (label_flex);

/*
* Make a copy of the original set of torsion bonds
* 10/96 te
*/
  copy_torsions (&temporary, molecule);

/*
* Flag all ring bonds
* 10/96 te
*/
  detect_rings (molecule);

/*
* Identify all bonds with flex labels
* 10/96 te
*/
  for (i = molecule->total.torsions = 0; i < molecule->total.bonds; i++)
  {
    molecule->bond[i].flex_id = 0;

    if (molecule->bond[i].ring_flag == TRUE)
      continue;

    if ((molecule->atom[molecule->bond[i].origin].neighbor_total <= 1) ||
      (molecule->atom[molecule->bond[i].target].neighbor_total <= 1))
      continue;

    for (j = 1; j < label_flex->total; j++)
    {
      if
      (
        check_atom
          (molecule, molecule->bond[i].origin,
          &label_flex->member[j].definition[0]) &&
        check_atom
          (molecule, molecule->bond[i].target,
          &label_flex->member[j].definition[1])
      )
        molecule->bond[i].flex_id = j;

      else if
      (
        check_atom
          (molecule, molecule->bond[i].origin,
          &label_flex->member[j].definition[1]) &&
        check_atom
          (molecule, molecule->bond[i].target,
          &label_flex->member[j].definition[0])
      )
        molecule->bond[i].flex_id = j;
    }

    if (molecule->bond[i].flex_id)
      molecule->total.torsions++;
  }

/*
* Check for sufficient space to store new and old torsions
* 10/96 te
*/
  if ((molecule->total.torsions + temporary.total.torsions) >
    molecule->max.torsions)
  {
    molecule->total.torsions += temporary.total.torsions;
    reallocate_torsions (molecule);
  }

  else
    reset_torsions (molecule);

/*
* Store flexible torsions
* 10/96 te
*/
  for (i = molecule->total.torsions = 0; i < molecule->total.bonds; i++)
    if (molecule->bond[i].flex_id)
    {
      molecule->torsion[molecule->total.torsions].bond_id = i;
      molecule->torsion[molecule->total.torsions].flex_id =
        molecule->bond[i].flex_id;
      molecule->total.torsions++;
    }

/*
* Append original torsions
* 10/96 te
*/
  for (i = 0; i < temporary.total.torsions; i++)
  {
    for (j = 0, bond_found = FALSE; j < molecule->total.torsions; j++)
      if (temporary.torsion[i].bond_id == molecule->torsion[j].bond_id)
      {
        molecule->torsion[j].target_angle = temporary.torsion[i].target_angle;
        bond_found = TRUE;
      }

    if (!bond_found)
      copy_torsion
        (&molecule->torsion[molecule->total.torsions++],
        &temporary.torsion[i]);
  }

/*
* Compute current torsion angles and conformation total
* 10/96 te
*/
  get_torsion_neighbors (molecule);

  for (i = 0, conf_total = 1; i < molecule->total.torsions; i++)
  {
    molecule->torsion[i].current_angle =
      molecule->torsion[i].target_angle =
      compute_torsion (molecule, i);

    molecule->torsion[i].periph_flag =
      check_peripheral_torsion (molecule, i);

    bond_id = molecule->torsion[i].bond_id;

    if (label_flex->drive_flag)
    {
      flex_id = molecule->bond[bond_id].flex_id;

      if (flex_id > 0)
      {
        if (label_flex->member[flex_id].torsion_total < 1)
          exit (fprintf (global.outfile,
            "ERROR assign_flex_labels: bond with no positions detected\n"));

        if (conf_total < INT_MAX / label_flex->member[flex_id].torsion_total)
          conf_total *= label_flex->member[flex_id].torsion_total;

        else
          conf_total = INT_MAX;
      }
    }
  }

/*
* Print out flex label assignments
* 10/95 te
*/
  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "____Flexible_Bond_Label_Assignments____\n");
    fprintf (global.outfile, "%s:\n\n", molecule->info.name);
    fprintf (global.outfile,
      "%-4s %-4s || %-4s | %-4s | %-4s | %-4s || %7s %2s %s\n",
      "tors", "bond", "orin", "ori", "trg", "trgn", "angle", "ps", "type");
    fprintf (global.outfile,
      "----------------------------------------"
      "---------------------------\n");

    for (i = 0; i < molecule->total.torsions; i++)
    {
      bond_id = molecule->torsion[i].bond_id;

      fprintf
      (
        global.outfile,
        "%-4d %-4d || %-4s | %-4s | %-4s | %-4s || %7.2f %2d %s\n",
        i + 1, bond_id + 1,
        molecule->atom[molecule->torsion[i].origin_neighbor].name,
```

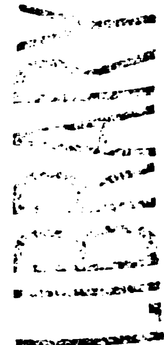
USF LIBRARY

```
        molecule->atom[molecule->torsion[i].origin].name,
        molecule->atom[molecule->torsion[i].target].name,
        molecule->atom[molecule->torsion[i].target_neighbor].name,
        molecule->torsion[i].current_angle / PI * 180,
        label_flex->member[molecule->bond[bond_id].flex_id].torsion_total,
        molecule->bond[bond_id].flex_id ?
          label_flex->member[molecule->bond[bond_id].flex_id].name :
          "INFLEXIBLE"
      );
    }

    if (label_flex->drive_flag)
      fprintf (global.outfile, "\nTotal conformations for %s %d\n",
        molecule->info.name, conf_total);
  }
}


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

Subroutine to identify all bonds in rings.

11/96 te

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

void detect_rings (MOLECULE *molecule)
{
  int i, j;

  int next_atom_level (int, int, int, MOLECULE *);

/*
* Reset atom flags and bond ring flags
* 4/97 te
*/
  for (i = 0; i < molecule->total.atoms; i++)
    molecule->atom[i].flag = 0;

  for (i = 0; i < molecule->total.bonds; i++)
    molecule->bond[i].ring_flag = 0;

/*
* Identify ring bonds
* 4/97 te
*/
  next_atom_level (1, 0, 0, molecule);

/*
* Also identify bonds specified in RIGID set, if present
* 4/97 te
*/
  for (i = 0; i < molecule->total.sets; i++)
  {
    if ((molecule->set[i].name != NULL) &&
      !strcmp (molecule->set[i].name, "RIGID"))
    {
      if ((molecule->set[i].type != NULL) &&
        !strcmp (molecule->set[i].type, "STATIC"))
      {
        if ((molecule->set[i].obj_type != NULL) &&
          !strcmp (molecule->set[i].obj_type, "BONDS"))
        {
          if (molecule->set[i].member_total > 0)
          {
            for (j = 0; j < molecule->set[i].member_total; j++)
            {
              if ((molecule->set[i].member[j] >= 0) &&
                (molecule->set[i].member[j] < molecule->total.bonds))
                molecule->bond[molecule->set[i].member[j]].ring_flag = TRUE;

              else
                exit (fprintf (global.outfile,
                  "ERROR detect_rings: RIGID set contains invalid bonds\n"));
            }
          }

          else
            exit (fprintf (global.outfile,
              "ERROR detect_rings: RIGID set empty\n"));
        }

        else
          exit (fprintf (global.outfile,
            "ERROR detect_rings: RIGID set must be BONDS sub_type\n"));
      }

      else
        exit (fprintf (global.outfile,
          "ERROR detect_rings: RIGID set must be STATIC type\n"));
    }
  }

/*
  for (i = 0; i < molecule->total.bonds; i++)
    if (molecule->bond[i].ring_flag)
      fprintf (global.outfile, "Ring bond %d %s %s\n", i + 1,
        molecule->atom[molecule->bond[i].origin].name,
        molecule->atom[molecule->bond[i].target].name);
*/
}


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

Recursive subroutine that traverses all atom linkage paths in a molecule.
The level of recursion is recorded for each atom as it is traversed.
The recursion level is reported back, unless an atom is re-encountered
during a higher level of recursion; then, the level of the previously seen
atom is reported.

A ring bond is identified when an atom is linked to an atom previously seen.

4/97 te

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

int next_atom_level
(
  int level,
  int current_atom,
  int previous_atom,
  MOLECULE *molecule
)
{
  int i;
  int bond_id;                    /* Bond id linking current neighbor */
  int neighbor_flag;              /* Flag of neighbor atom */

/*
* If the current atom hasn't been flagged, then check the flags
* of its neighbors
* 4/97 te
*/
  if (molecule->atom[current_atom].flag == 0)
  {
    molecule->atom[current_atom].flag = level;

    for (i = 0; i < molecule->atom[current_atom].neighbor_total; i++)
    {
      if (molecule->atom[current_atom].neighbor[i].id != previous_atom)
      {
        neighbor_flag =
          next_atom_level
          (
            level + 1,
            molecule->atom[current_atom].neighbor[i].id,
            current_atom,
            molecule
          );

        if (neighbor_flag <= level)
        {
          bond_id = molecule->atom[current_atom].neighbor[i].bond_id;
          molecule->bond[bond_id].ring_flag = TRUE;
        }

        if (neighbor_flag < molecule->atom[current_atom].flag)
          molecule->atom[current_atom].flag = neighbor_flag;
      }
    }
  }

  return molecule->atom[current_atom].flag;
}


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

int check_peripheral_torsion
(
  MOLECULE      *molecule,
  int           torsion_id
)
{
  int atom_id;

  atom_id = molecule->torsion[torsion_id].origin_neighbor;
  if (molecule->atom[atom_id].heavy_flag == FALSE)
    return TRUE;

  atom_id = molecule->torsion[torsion_id].target_neighbor;
  if (molecule->atom[atom_id].heavy_flag == FALSE)
    return TRUE;

  return FALSE;
}


/* ////////////////////////////////////////////////////////////////////////// */
/*
Routine to assign torsions to segments and segments to layers.
It also loops through all anchor fragments.

Return values:
  TRUE  successful identification of anchor segment
  EOF   unable to identify any more anchor segments

12/96 te

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

int get_anchor
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *mol_init,
  MOLECULE      *mol_anch,
  int           anchor
)
{
/*
* Initialize variables
* 12/96 te
*/
  if (anchor == 0)
  {
/*
*   Assign flexible labels
*   12/96 te
*/
    if (label_flex->flag)
    {
      mol_init->transform.tors_flag = FALSE;
      assign_flex_labels (label_flex, mol_init);

      if (mol_init->total.torsions > label_flex->max_torsions)
      {
        if (global.output_volume != 't')
          fprintf(global.outfile, "Skipped %s (%d rotatable bonds).\n",
            mol_init->info.name, mol_init->total.torsions);

        return EOF;
      }
    }

    get_segments (mol_init);
    copy_molecule (mol_anch, mol_init);
  }

/*
* Exit if return visit, but multiple anchors not requested
* 12/96 te
*/
  else if (!label_flex->multiple_anchors)
    return EOF;

  else
    copy_segments (mol_anch, mol_init);

/*
* Find next anchor
* 12/96 te
*/
  if (get_anchor_segment (label_flex, mol_anch, anchor) != TRUE)
    return EOF;

  initialize_segments (label_flex, mol_anch);
```

```c
  initialize_anchor_layer (label_flex, mol_anch);

  return TRUE;
}


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

Routine to divide a molecule into rigid segments.

11/96 te

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

void get_segments (MOLECULE *mol_init)
{
  int i, j;
  int atom_id;                  /* Atom id */
  int bond_id;                  /* Bond id */
  int torsion_id;               /* Torsion id */
  int origin;                   /* Origin atom id */
  int target;                   /* Target atom id */
  int segment;                  /* Current segment id */
  int neighbor;                 /* Neighboring segment id */
  int neighbor_id;              /* Current position in neighbor list */

  static SEARCH search = {0};

/*
* Allocate more than enough space for segments
* 11/96 te
*/
  mol_init->total.segments = mol_init->total.torsions + 1;
  reallocate_segments (mol_init);

  for (i = 0; i < mol_init->total.segments; i++)
  {
    mol_init->segment[i].atom_total = mol_init->total.atoms;
    mol_init->segment[i].neighbor_total = mol_init->total.torsions;

    reallocate_segment_atoms (&mol_init->segment[i]);
    reallocate_segment_neighbors (&mol_init->segment[i]);

    reset_segment (&mol_init->segment[i]);
  }

/*
* Identify rigid segments separated by rotatable bonds
* 11/96 te
*/
  for
  (
    i = mol_init->total.segments = 0;
    breadth_search
    (
      &search,
      mol_init->atom,
      mol_init->total.atoms,
      get_atom_neighbor,
      NULL,
      &i, 1, NEITHER, i
    ) != EOF;
    i++
  )
  {
    origin = get_search_origin (&search, NEITHER, NEITHER, NEITHER);
    target = get_search_target (&search, NEITHER, NEITHER, NEITHER);

/*
*   If this is the first atom, then initialize segment list only
*   11/96 te
*/
    if (i == 0)
    {
      mol_init->atom[target].segment_id = 0;
      mol_init->segment[0].atom[0] = target;
      mol_init->segment[0].atom_total = 1;
      mol_init->total.segments = 1;
      continue;
    }

/*
*   Check if this linkage is a rotatable bond
*   11/96 te
*/
    for
    (
      j = 0, torsion_id = NEITHER;
      (j < mol_init->total.torsions) && (torsion_id == NEITHER);
      j++
    )
      if (mol_init->torsion[j].flex_id > 0)
      {
        bond_id = mol_init->torsion[j].bond_id;

        if (((mol_init->bond[bond_id].origin == origin) &&
          (mol_init->bond[bond_id].target == target)) ||
          ((mol_init->bond[bond_id].origin == target) &&
          (mol_init->bond[bond_id].target == origin)))
          torsion_id = j;
      }

/*
*   If the link is flexible,then increment the segment total,
*   update the segment neighbor lists and atom list, and atom segment id
*   11/96 te
*/
    if (torsion_id != NEITHER)
    {
      segment = mol_init->total.segments++;
      neighbor = mol_init->atom[origin].segment_id;

      neighbor_id = mol_init->segment[segment].neighbor_total;
      mol_init->segment[segment].neighbor[neighbor_id].id = neighbor;
      mol_init->segment[segment].neighbor_total++;

      neighbor_id = mol_init->segment[neighbor].neighbor_total;
      mol_init->segment[neighbor].neighbor[neighbor_id].id = segment;
      mol_init->segment[neighbor].neighbor_total++;

      atom_id = mol_init->segment[segment].atom_total;
      mol_init->segment[segment].atom[atom_id] = target;
      mol_init->segment[segment].atom_total++;

      mol_init->atom[target].segment_id = segment;

      mol_init->segment[segment].periph_flag =
        mol_init->torsion[torsion_id].periph_flag;
    }

/*
*   Otherwise, update the segment atom list and atom segment id
*   11/96 te
*/
    else
    {
      segment = mol_init->atom[origin].segment_id;

      atom_id = mol_init->segment[segment].atom_total;
      mol_init->segment[segment].atom[atom_id] = target;
      mol_init->segment[segment].atom_total++;

      mol_init->atom[target].segment_id = segment;
    }
  }

/*
* Determine the size of each segment
* 1/97 te
*/
  for (i = 0; i < mol_init->total.segments; i++)
  {
    mol_init->segment[i].heavy_total = 0;

    for (j = 0; j < mol_init->segment[i].atom_total; j++)
    {
      atom_id = mol_init->segment[i].atom[j];

      if (mol_init->atom[atom_id].heavy_flag == TRUE)
        mol_init->segment[i].heavy_total++;
    }
  }

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "\nInitial segments\n");

    for (i = 0; i < mol_init->total.segments; i++)
    {
      fprintf (global.outfile, "  segment %d\n", i + 1);
      fprintf (global.outfile, "    neighbors:");

      for (j = 0; j < mol_init->segment[i].neighbor_total; j++)
        fprintf (global.outfile, " %d",
          mol_init->segment[i].neighbor[j].id + 1);

      fprintf (global.outfile, "\n    atoms    :");

      for (j = 0; j < mol_init->segment[i].atom_total; j++)
        fprintf (global.outfile, " %s",
          mol_init->atom[mol_init->segment[i].atom[j]].name);

      fprintf (global.outfile, "\n");
    }
  }
}


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

Routine to identify segments to use as anchors.

Return values:
        TRUE:   next anchor found
        EOF:    unable to find any more anchors

12/96 te

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

int get_anchor_segment
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule,
  int           anchor_count
)
{
  int i;
  int size;
  int anchor_found = FALSE;     /* Flag for whether anchor found */

  if ((anchor_count > 0) &&
    (label_flex->multiple_anchors == FALSE))
    return EOF;

/*
* Check if an anchor was specified in input
* 1/97 te
*/
  for (i = 0; i < molecule->total.sets; i++)
    if (!strcmp (molecule->set[i].name, "ANCHOR"))
    {
      if
      (
        (anchor_count == 0) &&
        (molecule->set[i].member[0] >= 0) &&
        (molecule->set[i].member[0] < molecule->total.atoms)
      )
      {
        molecule->transform.anchor_atom = molecule->set[i].member[0];
        anchor_found = TRUE;
      }

      else
        return EOF;
    }

/*
* If multiple anchors allowed, then find the next segment
* that satisfies the anchor size cutoff
* 1/97 te
*/
  if ((anchor_found == FALSE) &&
    (label_flex->multiple_anchors == TRUE))
  {
    for
    (
      anchor_found = FALSE,
        i = (anchor_count > 0) ?
          molecule->atom[molecule->transform.anchor_atom].segment_id + 1 : 0;
      (anchor_found == FALSE) &&
        (i < molecule->total.segments);
      i++
    )
      if ((molecule->segment[i].heavy_total +
        molecule->segment[i].neighbor_total) >=
        label_flex->anchor_size)
      {
```

UCSF LIBRARY

```
        molecule->transform.anchor_atom = molecule->segment[i].atom[0];
        anchor_found = TRUE;
      }
  }

/*
* Identify the largest segment (counting all heavy atoms and one atom
* from each neighbor)
* 1/97 te
*/
  if
  (
    (anchor_found == FALSE) &&
    (((label_flex->multiple_anchors == TRUE) && (anchor_count == 0)) ||
    (label_flex->multiple_anchors == FALSE))
  )
  {
    for
    (
      i = 0, size = INT_MIN;
      i < molecule->total.segments;
      i++
    )
      if ((molecule->segment[i].heavy_total +
        molecule->segment[i].neighbor_total) > size)
      {
        size = molecule->segment[i].heavy_total +
          molecule->segment[i].neighbor_total;
        molecule->transform.anchor_atom = molecule->segment[i].atom[0];
      }

    anchor_found = TRUE;
  }

  if (anchor_found == TRUE)
    return TRUE;

  else
    return EOF;
}


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

Routine to initialize segments by
1. updating atom segment ids
2  assigning torsions to each segment and
3. transfering bond target atoms from the target segment back to the
   origin segment (since they do not move during bond rotation).

11/96 te

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

void initialize_segments
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule
)
{
  int i, j, k;
  int atom_id;                    /* Atom id */
  int origin;                     /* Origin atom id */
  int target;                     /* Target atom id */
  int segment;                    /* Current segment id */

  static SEARCH search = {0};
  static MOLECULE temporary = {0};

/*
* Reset atom segment records and neighbor flags
* 12/96 te
*/
  for (i = 0; i < molecule->total.segments; i++)
    for (j = 0; j < molecule->segment[i].atom_total; j++)
    {
      atom_id = molecule->segment[i].atom[j];
      molecule->atom[atom_id].segment_id = i;
    }

/*
* Allocate extra space in temporary structure and copy from molecule
* 12/96 te
*/
  temporary.total.segments = molecule->total.segments;
  reallocate_segments (&temporary);

  for (i = 0; i < molecule->total.segments; i++)
  {
    temporary.segment[i].atom_total = molecule->total.atoms;
    reallocate_segment_atoms (&temporary.segment[i]);
  }

  copy_atoms (&temporary, molecule);
  copy_bonds (&temporary, molecule);
  copy_segments (&temporary, molecule);

/*
* Sort atoms radially from anchor atom
* 1/97 te
*/
  for
  (
    i = 0;
    breadth_search
    (
      &search,
      molecule->atom,
      molecule->total.atoms,
      get_atom_neighbor,
      flag_atom_neighbor,
      &molecule->transform.anchor_atom, 1,
      NEITHER, i
    ) != EOF;
    i++
  );

/*
* Assign each flexible bond to segments and transfer target atom
* over to origin segment, since it does not move upon bond rotation.
* 11/96 te
*/
  for (i = 0; i < molecule->total.torsions; i++)
  {
    if (molecule->torsion[i].flex_id)
    {
/*

      fprintf (global.outfile, "tors %d orig %s (%d) targ %s (%d)\n",
        i, molecule->atom[molecule->torsion[i].origin].name,
        molecule->atom[molecule->torsion[i].origin].segment_id,
        molecule->atom[molecule->torsion[i].target].name,
        molecule->atom[molecule->torsion[i].target].segment_id);
*/

      origin = molecule->torsion[i].origin;
      target = molecule->torsion[i].target;

      if (get_search_radius (&search, origin, NEITHER) >
        get_search_radius (&search, target, NEITHER))
        reverse_torsion (&molecule->torsion[i]);

      origin = molecule->torsion[i].origin;
      target = molecule->torsion[i].target;

/*
*     Update segment torsion records
*     3/97 te
*/
      segment = temporary.atom[target].segment_id;
      molecule->torsion[i].segment_id = segment;

      if (temporary.segment[segment].torsion_id == NEITHER)
        temporary.segment[segment].torsion_id = i;

      else
        exit (fprintf (global.outfile,
          "ERROR initialize_segments: torsion assigned twice to segment\n"));

      temporary.segment[segment].conform_total =
        label_flex->member[molecule->torsion[i].flex_id].torsion_total;

/*
*     Update segment atom records
*     3/97 te
*/
      if (temporary.atom[origin].segment_id !=
        temporary.atom[target].segment_id)
      {
/*
*       Delete target atom from target segment
*       11/96 te
*/
        for (j = 0; j < temporary.segment[segment].atom_total; j++)
          if (temporary.segment[segment].atom[j] == target)
          {
            for (k = j + 1; k < temporary.segment[segment].atom_total; k++)
              temporary.segment[segment].atom[k - 1] =
                temporary.segment[segment].atom[k];

            temporary.segment[segment].atom_total--;

            if (temporary.atom[target].heavy_flag)
              temporary.segment[segment].heavy_total--;

            break;
          }

/*
*       Add target atom to origin segment
*       11/96 te
*/
        segment = temporary.atom[origin].segment_id;
        atom_id = temporary.segment[segment].atom_total++;
        temporary.segment[segment].atom[atom_id] = target;
        molecule->atom[target].segment_id = segment;

        if (temporary.atom[target].heavy_flag)
          temporary.segment[segment].heavy_total++;
      }

      else
        exit (fprintf (global.outfile,
          "ERROR initialize_segments: torsion atoms in same segment\n"));
    }
  }

  copy_segments (molecule, &temporary);

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "\nFinal segments\n");

    for (i = 0; i < molecule->total.segments; i++)
    {
      fprintf (global.outfile, "  segment    : %d\n", i + 1);
      fprintf (global.outfile, "    conforms : %d\n",
        molecule->segment[i].conform_total);
      fprintf (global.outfile, "    neighbors:");

      for (j = 0; j < molecule->segment[i].neighbor_total; j++)
        fprintf (global.outfile, " %d",
          molecule->segment[i].neighbor[j].id + 1);

      fprintf (global.outfile, "\n    atoms    :");

      for (j = 0; j < molecule->segment[i].atom_total; j++)
        fprintf (global.outfile, " %s",
          molecule->atom[molecule->segment[i].atom[j]].name);

      fprintf (global.outfile, "\n");
    }
  }
}


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

void initialize_anchor_layer
(
  LABEL_FLEX    *label_flex,
  MOLECULE      *molecule
)
{
  int segment;
  int segment_id;
  int segment_anchor;
  int layer;
  static SEARCH search = {0};
  SORT_INT *list = NULL;

/*
* If no anchor search, initialize first layer to contain all segments
* 3/97 te
*/
  if (label_flex->anchor_flag == FALSE)
  {
    molecule->total.layers = molecule->total.segments;
```


LSE LIBRARY

```
    reallocate_layers (molecule);
    molecule->total.layers = 1;

    molecule->layer[0].segment_total = molecule->total.segments;
    reallocate_layer_segments (&molecule->layer[0]);

    segment_anchor = molecule->atom[molecule->transform.anchor_atom].segment_id;
    molecule->layer[0].segment[0] = segment_anchor;
    molecule->layer[0].conform_total = 1;

    for
    (
      segment = 0, segment_id = 1;
      segment < molecule->total.segments;
      segment++
    )
    {
      if (segment != segment_anchor)
        molecule->layer[0].segment[segment_id++] = segment;

      molecule->layer[0].conform_total *=
        molecule->segment[segment].conform_total;
      molecule->segment[segment].layer_id = 0;
      molecule->segment[segment].active_flag = TRUE;
      molecule->segment[segment].min_flag = TRUE;
    }

    return;
  }

/*
* Otherwise, allocate enough layers to hold one segment each AND
* each of these layers to hold all segments (if necessary).
* 3/97 te
*/
  molecule->total.layers = molecule->total.segments + 1;
  reallocate_layers (molecule);

  for (layer = 0; layer < molecule->total.layers; layer++)
  {
    molecule->layer[layer].segment_total = molecule->total.segments;
    reallocate_layer_segments (&molecule->layer[layer]);
  }

  molecule->total.layers = 1;

/*
* Assign anchor segment to first layer and initialize layer_id of all segments
* 3/97 te
*/
  segment_anchor = molecule->atom[molecule->transform.anchor_atom].segment_id;
  molecule->layer[0].segment_total = 1;
  molecule->layer[0].segment[0] = segment_anchor;
  molecule->layer[0].conform_total = 1;

  molecule->segment[segment_anchor].layer_id = 0;
  molecule->segment[segment_anchor].active_flag = TRUE;
  molecule->segment[segment_anchor].min_flag = TRUE;

  for (segment = 0; segment < molecule->total.segments; segment++)
    if (segment != segment_anchor)
      molecule->segment[segment].layer_id = NEITHER;

/*
* Flag the segment neighbors radially from the anchor segment
* 3/97 te

  NOTE: This section of the code has been disabled.
        It allows additional segments to be added to the anchor layer
        to satisfy the anchor_size parameter.  Since this adds flexibility
        to the anchor, I have decided to temporarily disable it until
        it can be proven to be necessary.
        3/97 te

  for
  (
    segment = 0;
    breadth_search
    (
      &search,
      molecule->segment,
      molecule->total.segments,
      get_segment_neighbor,
      flag_segment_neighbor,
      &segment_anchor, 1,
      NEITHER, segment
    ) != EOF;
    segment++
  );


* Build the anchor layer from anchor segment until it is sufficiently large.
* Merge the largest adjacent segment at each iteration.
* 3/97 te

  for
  (
    anchor_size =
      molecule->segment[segment_anchor].heavy_total;
    anchor_size < label_flex->anchor_size;
  )
  {
    for
    (
      segment_id = 0, segment_anchor = NEITHER, heavy_max = INT_MIN;
      segment_id < molecule->layer[0].segment_total;
      segment_id++
    )
    {
      segment = molecule->layer[0].segment[segment_id];

      for
      (
        neighbor_id = 0;
        neighbor_id < molecule->segment[segment].neighbor_total;
        neighbor_id++
      )
      {
        if (molecule->segment[segment].neighbor[neighbor_id].out_flag == TRUE)
        {
          neighbor = molecule->segment[segment].neighbor[neighbor_id].id;

          if
          (
            (molecule->segment[neighbor].layer_id == NEITHER) &&
            (molecule->segment[neighbor].heavy_total > heavy_max)
          )
          {
            heavy_max = molecule->segment[neighbor].heavy_total;
            segment_anchor = neighbor;
          }
        }
      }
    }

    if (segment_anchor != NEITHER)
    {
      molecule->layer[0].segment[molecule->layer[0].segment_total++] =
        segment_anchor;
      molecule->layer[0].conform_total *=
        molecule->segment[segment_anchor].conform_total;
      molecule->segment[segment_anchor].layer_id = 0;
      molecule->segment[segment_anchor].active_flag = TRUE;
      molecule->segment[segment_anchor].min_flag = TRUE;
      anchor_size += molecule->segment[segment_anchor].heavy_total;
    }

    else
      break;
  }
*/

/*
* Regroup the segments radially from the anchor segment
* 3/97 te
*/
  for
  (
    segment = 0;
    breadth_search
    (
      &search,
      molecule->segment,
      molecule->total.segments,
      get_segment_neighbor,
      flag_segment_neighbor,
      molecule->layer[0].segment, molecule->layer[0].segment_total,
      NEITHER, segment
    ) != EOF;
    segment++
  );

/*
* Assign segments to outer layers
* 3/97 te
*/
  ecalloc
  (
    (void **) &list,
    molecule->total.segments,
    sizeof (SORT_INT),
    "initial layer sort list",
    global.outfile
  );

  molecule->total.layers = search.radius + 1;

  for
  (
    layer = 1;
    layer <= search.radius;
    layer++
  )
  {
    molecule->layer[layer].segment_total = search.total[layer];
    molecule->layer[layer].conform_total = 1;

    for
    (
      segment_id = 0;
      segment_id < search.total[layer];
      segment_id++
    )
    {
      segment = search.target[layer][segment_id];

      list[segment_id].id = segment;
      list[segment_id].value = molecule->segment[segment].heavy_total;
    }

    qsort
    (
      list,
      search.total[layer],
      sizeof (SORT_INT),
      (int (*)()) compare_int_descend
    );

    for
    (
      segment_id = 0;
      segment_id < search.total[layer];
      segment_id++
    )
    {
      segment = list[segment_id].id;
      molecule->segment[segment].layer_id = layer;
      molecule->layer[layer].segment[segment_id] = segment;
      molecule->layer[layer].conform_total *=
        molecule->segment[segment].conform_total;
    }
  }

  efree ((void **) &list);

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "Molecule Layers\n");
    for (layer = 0; layer < molecule->total.layers; layer++)
    {
      fprintf (global.outfile, "layer %2d: conforms %4d, segments",
        layer + 1, molecule->layer[layer].conform_total);

      for
      (
        segment = 0;
        segment < molecule->layer[layer].segment_total;
        segment++
      )
        fprintf (global.outfile, " %2d ",
          molecule->layer[layer].segment[segment] + 1);

      fprintf (global.outfile, "\n");
    }
  }
}

*****************************************************************************
```

```c
********************************** LABEL_NODE.H **********************************
**********************************************************************************
/*                                                                              */
/*                              Copyright UCSF, 1997                            */
/*                                                                              */

/*
Written by Todd Ewing
10/95
*/

/* Structures to store atom labelling definitions */

typedef struct node_struct
{
  STRING5 type;
  int include, next_total;
  int vector_atom, multiplicity;
  float weight;
  struct node_struct *next[6];
} NODE;


/* Routines to manipulate atom label structures */

NODE *create_node (void);
int assign_node    (NODE *node, int include);
void free_node     (NODE *node);
void print_node    (NODE *node, int level);
int check_type     (char *candidate, char *reference);

int check_atom
(
  MOLECULE *molecule,
  int      current_atom,
  NODE     *node
);

int check_bonded_atoms
(
  MOLECULE *molecule,
  int      current_atom,
  int      previous_atom,
  NODE     *node
);

**********************************************************************************
********************************** LABEL_NODE.C **********************************
**********************************************************************************
/*                                                                              */
/*                              Copyright UCSF, 1997                            */
/*                                                                              */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "label_node.h"


/* ======================================================================== */
int assign_node
(
  NODE *node,
  int include
)
{
  char *branch;
  STRING5 temp;

  (*node).next_total = 0;
  strcpy (temp, strtok (NULL, " "));

  if (isdigit (temp[0]))
  {
    (*node).multiplicity = atoi (temp);
    if ((*node).multiplicity < 0)
    {
      fprintf (global.outfile,
        "Cannot specify negative multiplicity in definition.\n");
      return FALSE;
    }
    strcpy ((*node).type, strtok (NULL, " "));
  }
  else
  {
    (*node).multiplicity = 0;
    strcpy ((*node).type, temp);
  }

  (*node).include = include;

  while (branch = strtok (NULL, " "))
  {
    if ((!strncmp (branch, "(", 1)) || (!strncmp (branch, "[", 1)))
    {
      if ((*node).next_total >= 6)
      {
        fprintf (global.outfile,
          "Cannot exceed 6 substituents for each atom in definition.\n");
        return FALSE;
      }

      ecalloc
      (
        (void **) &(*node).next[(*node).next_total],
        1,
        sizeof (NODE),
        "next label node",
        global.outfile
      );

      if ((!strncmp (branch, "(", 1)) &&
        (!assign_node ((*node).next[(*node).next_total], TRUE)))
        return FALSE;
      if ((!strncmp (branch, "[", 1)) &&
        (!assign_node ((*node).next[(*node).next_total], FALSE)))
        return FALSE;
      (*node).next_total++;
    }

    else if ((!strncmp (branch, ")", 1)) || (!strncmp (branch, "]", 1)))
      return TRUE;

    else
      return FALSE;
  }
  return TRUE;
}

/* ======================================================================== */
void free_node (NODE *node)
{
  int i;

  for (i = 0; i < node->next_total; i++)
  {
    free_node (node->next[i]);
    efree ((void **) &node->next[i]);
  }
}


/* ======================================================================== */
void print_node (NODE *node, int level)
{
  int i;

  if (level)
  {
    if ((*node).include)
      fprintf (global.outfile, " (");
    else
      fprintf (global.outfile, " [");
  }

  if ((*node).multiplicity)
    fprintf (global.outfile, "%d ", (*node).multiplicity);

  fprintf (global.outfile, "%s", (*node).type);

  for (i = 0; i < (*node).next_total; i++)
    print_node ((*node).next[i], level + 1);

  if (level)
  {
    if ((*node).include)
      fprintf (global.outfile, ")");
    else
      fprintf (global.outfile, "]");
  }
}


/* ======================================================================== */
int check_type (char *candidate, char *reference)
{
  if ((strstr (candidate, reference)) || (reference[0] == '*'))
    return TRUE;
  else
    return FALSE;
}


/* ======================================================================== */
int check_atom
(
  MOLECULE *mol,
  int currentatom,
  NODE *node
)
{
  int i;
  int match;

  if (match = check_type (mol->atom[currentatom].type, (*node).type))
    for (i = 0; i < (*node).next_total; i++)
      if (!check_bonded_atoms
        (mol, currentatom, currentatom, (*node).next[i]))
        match = FALSE;

  return match;
}

/* ======================================================================== */
int check_bonded_atoms
(
  MOLECULE *mol,
  int currentatom,
  int previousatom,
  NODE *node
)
{
  int i, j, nextatom;
  int match, match_count = 0;

  for (i = 0; i < mol->atom[currentatom].neighbor_total; i++)
  {
    nextatom = mol->atom[currentatom].neighbor[i].id;
    match = FALSE;

    if ((nextatom != previousatom) &&
      (match = check_type (mol->atom[nextatom].type, (*node).type)))
    {
      for (j = 0; j < (*node).next_total; j++)
        if (!check_bonded_atoms
          (mol, nextatom, currentatom, (*node).next[j]))
          match = FALSE;
    }

    if (match)
      match_count++;
  }

  if ((*node).multiplicity)
  {
    if ((*node).multiplicity == match_count)
      match = TRUE;
    else
      match = FALSE;
  }
  else
  {
    if (match_count)
      match = TRUE;

    else
      match = FALSE;
  }

  if (match == (*node).include)
    return TRUE;
  else
    return FALSE;
```

```
}

################################################################################
################################## LABEL_VDW.H #################################
################################################################################
/*                                                                            */
/*                           Copyright UCSF, 1997                             */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

/* Structures to store atom labelling definitions */

typedef struct vdw_member_struct
{
  STRING20      name;                   /* Member name */
  NODE          *definition;            /* Atom definition */
  int           definition_total;       /* Number of definitions */

  int           atom_model;             /* All-atom or united-atom */
  int           bump_id;                /* Identifier used in bump grid */
  int           heavy_flag;             /* Flag for heavy (non-hydrogen) atom */
  int           valence;                /* Number of allowed bonds */
  float         radius;                 /* VDW radius */
  float         well_depth;             /* VDW well-depth */

} VDW_MEMBER;

typedef struct label_vdw_struct
{
  int           flag;                   /* Flag for vdw labeling */
  int           init_flag;              /* Flag for vdw initialization */
  VDW_MEMBER    *member;                /* Label members */
  int           total;                  /* Number of label members */
  NODE          *definition;            /* Member definitions */
  FILE_NAME     file_name;              /* File containing label parameters */

} LABEL_VDW;


/* Routines to manipulate atom label structures */

void get_vdw_labels (LABEL_VDW *vdw);
void free_vdw_labels (LABEL_VDW *vdw);

int  assign_vdw_labels
(
  LABEL_VDW     *vdw,
  int           atom_model,
  MOLECULE      *molecule
);

int count_heavies
(
  MOLECULE      *molecule,
  int           segment_id
);

################################################################################
################################## LABEL_VDW.C #################################
################################################################################
/*                                                                            */
/*                           Copyright UCSF, 1997                             */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "label_node.h"
#include "label_vdw.h"

/* ======================================================================== */
void get_vdw_labels (LABEL_VDW *label_vdw)
{
  int i, definition_total, definition_count;
  STRING100 line, model;
  FILE *vdw_file;

  label_vdw->init_flag = TRUE;

  vdw_file = efopen (label_vdw->file_name, "r", global.outfile);

/*
* Count up the number of label_vdw->member declarations and definitions,
* then allocate memory
* 6/95 te
*/
  label_vdw->total = 1;
  definition_count = 0;

  while (fgets (line, 100, vdw_file) != NULL)
  {
    if (!strncmp (line, "name", 4))
      label_vdw->total++;
    if (!strncmp (line, "definition", 10))
      definition_count++;
  }

  rewind (vdw_file);
  definition_total = definition_count;

  ecalloc
  (
    (void **) &label_vdw->member,
    label_vdw->total,
    sizeof (VDW_MEMBER),
    "label_vdw->member structures",
    global.outfile
  );

  ecalloc
  (
    (void **) &label_vdw->definition,
    definition_count,
    sizeof (NODE),
    "atom label_vdw->member definitions",
    global.outfile
  );

/*
* Read in the atom label_vdw->member definitions
* 6/95 te
*/
  strcpy (label_vdw->member[0].name, "null");

  for (i = 0; i < label_vdw->total; i++)
    label_vdw->member[i].valence = INT_MAX;

  label_vdw->total = 1;
  definition_count = definition_total = 0;
  while (fgets (line, 100, vdw_file) != NULL)
  {
/*
*   Process label_vdw->member declaration
*   6/95 te
*/
    if (!strncmp (line, "name", 4))
    {
      label_vdw->member[label_vdw->total - 1].definition_total =
        definition_total;
      definition_total = 0;
      label_vdw->member[label_vdw->total].definition =
        &label_vdw->definition[definition_count];

      if (sscanf (line, "%*s %s", label_vdw->member[label_vdw->total].name) < 1)
      {
        fprintf (global.outfile, "Incomplete vdw member declaration.\n");
        exit (EXIT_FAILURE);
      }

      label_vdw->total++;
    }

    else if (!strncmp (line, "atom_model", 10))
    {
      if (sscanf (line, "%*s %s", model) != 1)
      {
        fprintf (global.outfile, "Incomplete atom_model specification.\n");
        exit (EXIT_FAILURE);
      }

      label_vdw->member[label_vdw->total - 1].atom_model = tolower (model[0]);

      if ((label_vdw->member[label_vdw->total - 1].atom_model != 'a') &&
        (label_vdw->member[label_vdw->total - 1].atom_model != 'u') &&
        (label_vdw->member[label_vdw->total - 1].atom_model != 'e'))
      {
        fprintf (global.outfile,
          "Atom_model specification restricted to ALL, UNITED, or EITHER.\n");
        exit (EXIT_FAILURE);
      }
    }

    else if (!strncmp (line, "heavy_flag", 8))
    {
      if (sscanf (line, "%*s %d",
        &label_vdw->member[label_vdw->total - 1].heavy_flag) != 1)
      {
        fprintf (global.outfile, "Incomplete heavy_flag specification.\n");
        exit (EXIT_FAILURE);
      }
    }

    else if (!strncmp (line, "radius", 6))
    {
      if (sscanf (line, "%*s %f",
        &label_vdw->member[label_vdw->total - 1].radius) != 1)
      {
        fprintf (global.outfile, "Incomplete radius specification.\n");
        exit (EXIT_FAILURE);
      }
    }

    else if (!strncmp (line, "well_depth", 10))
    {
      if (sscanf (line, "%*s %f",
        &label_vdw->member[label_vdw->total - 1].well_depth) != 1)
      {
        fprintf (global.outfile, "Incomplete well_depth specification.\n");
        exit (EXIT_FAILURE);
      }
    }

    else if (!strncmp (line, "valence", 7))
    {
      if (sscanf (line, "%*s %d",
        &label_vdw->member[label_vdw->total - 1].valence) != 1)
      {
        fprintf (global.outfile, "Incomplete valence specification.\n");
        exit (EXIT_FAILURE);
      }
    }

/*
*   Process label_vdw->member definition
*   6/95 te
*/
    else if (!strncmp (line, "definition", 10))
    {
      strtok (white_line (line), " ");

      if (!assign_node
        (&label_vdw->definition[definition_count], TRUE))
      {
        fprintf (global.outfile,
          "Error assigning vdw member definitions.\n");
        exit (EXIT_FAILURE);
      }

      definition_count++;
      definition_total++;
    }
  }

/*
* Update last label_vdw->member info also
* 6/95 te
*/
  label_vdw->member[label_vdw->total - 1].definition_total = definition_total;

  efclose (&vdw_file);

/*
* Calculate parameters
* 6/95 te
*/
  for (i = 0; i < label_vdw->total; i++)
  {
    if (label_vdw->member[i].heavy_flag)
```

LIBRARY

```
        label_vdw->member[i].bump_id =
          NINT (10.0 * label_vdw->member[i].radius);

      else
        label_vdw->member[i].bump_id = 0;
  }

/*
* Print out the label_vdw->member and their definitions
* 6/95 te
*/

/*
  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "\n____VDW_Label_Definitions____\n\n");
    for (i = 1; i < label_vdw->total; i++)
    {
      NFPUTC ('_', 40);
      fprintf (global.outfile, "\n");
      fprintf (global.outfile, "%-20s%s\n", "name",
        label_vdw->member[i].name);
      fprintf (global.outfile, "%-20s%8c\n", "atom_model",
        label_vdw->member[i].atom_model);
      fprintf (global.outfile, "%-20s%8.3f\n", "radius",
        label_vdw->member[i].radius);
      fprintf (global.outfile, "%-20s%8.3f\n", "well_depth",
        label_vdw->member[i].well_depth);
      fprintf (global.outfile, "%-20s%8d\n", "heavy_flag",
        label_vdw->member[i].heavy_flag);
      fprintf (global.outfile, "%-20s%8d\n", "bump_id",
        label_vdw->member[i].bump_id);

      fprintf (global.outfile, "\n");

      for (j = 0; j < label_vdw->member[i].definition_total; j++)
      {
        fprintf (global.outfile, "%-20s", "definition");
        print_node (&label_vdw->member[i].definition[j], 0);
        fprintf (global.outfile, "\n");
      }

      fprintf (global.outfile, "\n");
    }

    fprintf (global.outfile, "\n\n");
  }
*/
}


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

void free_vdw_labels (LABEL_VDW *label_vdw)
{
  int i, j;

  for (i = 0; i < label_vdw->total; i++)
    for (j = 0; j < label_vdw->member[i].definition_total; j++)
      free_node (&label_vdw->member[i].definition[j]);

  efree ((void **) &label_vdw->member);
  efree ((void **) &label_vdw->definition);
}

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

int assign_vdw_labels
(
  LABEL_VDW *label_vdw,
  int atom_model,
  MOLECULE *molecule
)
{
  int i, j, k;
  int vdw_assigned = FALSE;

  if (!label_vdw->init_flag)
    get_vdw_labels (label_vdw);

  for (i = 0; i < molecule->total.atoms; i++)
  {
    for (j = 1; j < label_vdw->total; j++)
    {
      if ((atom_model == 'a') &&
        (label_vdw->member[j].atom_model == 'u'))
          continue;

      if ((atom_model == 'u') &&
        (label_vdw->member[j].atom_model == 'a'))
          continue;

      for (k = 0; k < label_vdw->member[j].definition_total; k++)
        if (check_atom
          (molecule, i, &label_vdw->member[j].definition[k]))
        {
          molecule->atom[i].vdw_id = j;
          vdw_assigned = TRUE;
        }
    }

    if (!vdw_assigned)
    {
      fprintf (global.outfile,
        "WARNING assign_vdw_labels: "
        "No vdw parameters for %ld %s %s %ld %s %s.\n",
        i + 1, molecule->atom[i].name, molecule->atom[i].type,
        molecule->atom[i].subst_id,
        molecule->subst[molecule->atom[i].subst_id].name,
        molecule->info.name);
      return NULL;
    }

    if (molecule->atom[i].neighbor_total >
      label_vdw->member[molecule->atom[i].vdw_id].valence)
    {
      for (j = k = 0; j < molecule->atom[i].neighbor_total; j++)
        if (strcmp (molecule->atom[molecule->atom[i].neighbor[j].id].type, "LP")
          &&
          strcmp (molecule->atom[molecule->atom[i].neighbor[j].id].type, "Du"))
          k++;

      if (k > label_vdw->member[molecule->atom[i].vdw_id].valence)
      {
        fprintf (global.outfile,
          "WARNING assign_vdw_labels: Atom valence violated for "
          "%s %s %s %d %s %d.\n",
          molecule->info.name,
          molecule->info.comment,
          molecule->subst[molecule->atom[i].subst_id].name,
          molecule->atom[i].subst_id + 1,
          molecule->atom[i].name, i + 1);

        return FALSE;
      }
    }

/*
*   Check to see if partial charge should be transferred from an atom
*   that is excluded from the current model (ie. aliphatic hydrogen in
*   a united atom model) to one that is
*   6/95 te
*/
    if ((atom_model == 'u') &&
      (ABS (label_vdw->member[molecule->atom[i].vdw_id].well_depth) < 0.00001))
    {
      if (molecule->atom[i].neighbor_total == 1)
      {
        molecule->atom[molecule->atom[i].neighbor[0].id].charge +=
          molecule->atom[i].charge;
        molecule->atom[i].charge = 0.0;
      }

      else
      {
        fprintf (global.outfile,
          "WARNING assign_vdw_labels: "
          "Unable to transfer partial charge away from "
          "%s %s %s %d %s %d.\n",
          molecule->info.name,
          molecule->info.comment,
          molecule->subst[molecule->atom[i].subst_id].name,
          molecule->atom[i].subst_id + 1,
          molecule->atom[i].name, i + 1);

        molecule->atom[i].charge = 0.0;
      }
    }
  }

  for (i = molecule->transform.heavy_total = 0; i < molecule->total.atoms; i++)
  {
    molecule->atom[i].heavy_flag =
      label_vdw->member[molecule->atom[i].vdw_id].heavy_flag;

    if (molecule->atom[i].heavy_flag == TRUE)
      molecule->transform.heavy_total++;
  }

  if (global.output_volume == 'v')
  {
    fprintf (global.outfile, "____VDW_Assignments____\n");
    fprintf (global.outfile, "%s:\n\n", molecule->info.name);
    for (i = 0; i < molecule->total.atoms; i++)
      fprintf (global.outfile, "%6d %4s %5s %-20s %d %d\n", i + 1,
        molecule->atom[i].name,
        molecule->atom[i].type,
        label_vdw->member[molecule->atom[i].vdw_id].name,
        molecule->atom[i].heavy_flag,
        label_vdw->member[molecule->atom[i].vdw_id].bump_id);
    fprintf (global.outfile, "\n\n");
  }

  return TRUE;
}


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

int count_heavies
(
  MOLECULE      *molecule,
  int           segment_id
)
{
  int i;
  int heavy_total;

  if (segment_id == NEITHER)
  {
    for (i = heavy_total = 0; i < molecule->total.atoms; i++)
      if (molecule->atom[i].heavy_flag)
        heavy_total++;

    return heavy_total;
  }

  else if ((segment_id >= 0) && (segment_id < molecule->total.segments))
  {
    for
    (
      i = heavy_total = 0;
      i < molecule->segment[segment_id].atom_total;
      i++
    )
      if (molecule->atom[molecule->segment[segment_id].atom[i]].heavy_flag)
        heavy_total++;

    return heavy_total;
  }

  else
    return NEITHER;
}

*****************************************************************************
*********************************** MATCH.H *********************************
*****************************************************************************
/*                                                                         */
/*                          Copyright UCSF, 1997                           */
/*                                                                         */

/*
Written by Todd Ewing
3/96
*/

typedef struct distance_list_struct
{
  float distance;
  int i, j;

} DISTANCE_LIST;

typedef struct edge_struct
{
  float residual;
  int node,
```

```
} EDGE;

typedef struct clique_struct
{
  EDGE *edge;
  int edge_total;
  int edge_max;
  float residual;
  int reflect_flag;

} CLIQUE;

typedef struct singly_linked_clique_struct
{
  CLIQUE clique;
  struct singly_linked_clique_struct *next;

} SLCLIQUE;

typedef struct match_struct
{
  int flag;                      /* Flag for matching */
  int uniform_flag;              /* Flag for uniform matching */
  int random_flag;               /* Flag for random matching */
  int centers_flag;              /* Flag for ligand centers from file */
  int chiral_flag;               /* Flag to check clique chirality */
  int reflect_flag;              /* Flag to reflect ligand */
  int degeneracy_flag;           /* Flag for checking if clique is subclique */
  int critical_flag;             /* Flag for critical spheres to prune search */
  int multiple_flag;             /* Flag for multiple selections from cluster */
  int chemical_flag;             /* Flag for chemical labels to prune search */

  int clique_size_min;           /* Minimum nodes in match for orient */
  int clique_size_max;           /* Maximum nodes in match for orient */
  float distance_minimum;        /* Minimum internal distance to match */
  float distance_tolerance;      /* Difference in internal distances to match */

  int total;                     /* Number of matches */
  int max;                       /* Maximum number of matches */

  FILE_NAME ligand_file_name;    /* File containing ligand centers */
  FILE_NAME receptor_file_name;  /* File containing receptor site points */

  float **receptor_distances;    /* Receptor distance matrix */
  int receptor_distance_size;    /* Receptor distance matrix size */

  float **ligand_distances;      /* Ligand distance matrix */
  int ligand_distance_size;      /* Ligand distance matrix size */

  XYZ **receptor_vectors;        /* Receptor vector matrix (for chirality) */
  int receptor_vector_size;      /* Receptor vector matrix size */

  XYZ **ligand_vectors;          /* Ligand vector matrix */
  int ligand_vector_size;        /* Ligand vector matrix size */

  SLINT2 **bin;                  /* Bin array of receptor distances */
  float bin_length;              /* Longest distance in bin array */
  float bin_width;               /* Width of each bin */
  int bin_total;                 /* Number of bins */

  int node_max;                  /* Maximum number of nodes */
  int node_total;                /* Current number of nodes */

  int init_flag;                 /* Flag for whether match info initialized */
  CLIQUE clique;                 /* Current clique */
  int *adjacent;                 /* Node under consideration in adjacency list */
  int *degenerate;               /* Record of whether clique is degenerate */
  SLCLIQUE *clique_link;         /* Linked list of all cliques generated */
  CLIQUE *clique_sort;           /* Truncated, sorted list of cliques */
  int clique_sort_total;         /* Length of list of sorted cliques */
  int cycle;                     /* Number of times matching performed */

  EDGE **adjacency_list;         /* Adjacency lists */
  int *adjacency_total;          /* Length of each adjacency list */
  int *adjacency_max;            /* Maximum length of each adjacency list */
  EDGE **clique_adjacency_list;  /* Adjacency lists for cliques */
  int *clique_adjacency_total;   /* Length of clique adjacency lists */

  char *chemical_filter;         /* Array encoding chemical nodes */
  char **critical_filter;        /* Array encoding critical clusters */
  int cluster_total;             /* Number of critical clusters */
  int **cluster;                 /* Members of each cluster */
  int *cluster_size;             /* Number of members in each cluster */
  int *histogram;                /* Histogram of clique populations */

  int *ligand key;               /* List of which atoms used as centers */
  int ligand_key_total;          /* Length of list of atoms as centers */
  MOLECULE ligand_center;        /* Ligand centers for matching */
  MOLECULE receptor_site;        /* Receptor site points for matching */

  MOLECULE ligand_clique;        /* Ligand centers in clique */
  MOLECULE receptor_clique;      /* Receptor site points in clique */

} MATCH;

/*
Routines defined in match.c, that are called by outside functions
*/

int get_match
(
  DOCK          *dock,
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *mol_conf,
  int           conformation_id,
  int           orientation_id
);

int get_random_match
(
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *mol_conf,
  int           molecule_id,
  int           conformation_id,
  int           orientation_id
);

void init_match_site
(
  MATCH         *match,
  LABEL         *label
);

void free_matches
(
  LABEL         *label,
  MATCH         *match
);

void free_random_matches (MATCH *match);

void free_match_site
(
  MATCH         *match,
  LABEL         *label
);

int init_match_ligand
(
  MATCH         *match,
  LABEL         *label
);

int initialize_adjacency
(
  MATCH         *match,
  LABEL         *label
);

void calculate_vectors (MOLECULE *molecule, XYZ ***matrix, int *size);
void free_vectors (XYZ ***matrix, int *size);

void get_site
(
  MATCH         *match,
  LABEL         *label
);

void get_centers
(
  MATCH         *match,
  LABEL         *label
);

void id_site_points (MATCH *match);
void order_site_points (MATCH *match);
void free_site_points (MATCH *match);
void make_distance_bins (MATCH *match);
void free_distance_bins (MATCH *match);
void allocate_clique_atoms (MATCH *match);

void allocate_match (MATCH *match);
void reallocate_match (MATCH *match);
void free_match (MATCH *match);
void reset_match (MATCH *match);

void allocate_clique (CLIQUE *clique);
void reallocate_clique (CLIQUE *clique);
void free_clique (CLIQUE *clique);
void reset_clique (CLIQUE *clique);
void copy_clique (CLIQUE *copy, CLIQUE *original);

int get_ligand_centers
(
  MATCH         *match,
  MOLECULE      *molecule
);

void free_ligand_centers (MATCH *match);

int get_ligand_keys
(
  MATCH         *match,
  MOLECULE      *molecule
);

int screen_match
(
  DOCK          *dock,
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *molecule
);

void make_chemical_filter
(
  LABEL_CHEMICAL *label_chemical,
  MATCH         *match
);

void compute_adjacency (MATCH *match);
void compute_chemical_adjacency (MATCH *match, LABEL *label);
void append_adjacency (MATCH *match, int list, int node, float residual);
void reset_adjacency (MATCH *match);
void free_adjacency (MATCH *match);
void reset_match (MATCH *match);
void free_match (MATCH *match);
void make_critical_filter (MATCH *match);

int compute_matches (MATCH *);
void free_clique_list (MATCH *);
void free_clique_link (MATCH *);
int match_distance (MATCH *);

void output_match_info (MATCH *match);
void extract_clique (MATCH *match, MOLECULE *molecule);

int check_screen
(
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *mol_init,
  int           molecule_id
);

void free_screen
(
  MATCH         *match,
  LABEL         *label
);

void init_similar_site
(
  MATCH         *match,
  LABEL         *label
);

*****************************************************************************
********************************** MATCH.C **********************************
*****************************************************************************
/*                                                                         */
/*                        Copyright UCSF, 1997                             */
/*                                                                         */

/*
```

```
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "screen.h"
#include "match.h"
#include "io.h"
#include "mol_prep.h"
#include "rotrans.h"


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

Routine to manage matching procedure
3/96 te

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

int get_match
(
  DOCK          *dock,
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *mol_conf,
  int           conformation_id,
  int           orientation_id
)
{
  if (orientation_id == 0)
  {
    match->total = 0;
    match->max = 0;
    match->cycle = 0;
    match->chiral_flag = TRUE;

    if (!match->init_flag)
      init_match_site (match, label);

    if (!match->centers_flag)
      get_ligand_centers (match, mol_conf);

    if (conformation_id == 0)
      init_match_ligand (match, label);
  }

  if (match->uniform_flag)
  {
    while (match->total >= match->max)
    {
      match->cycle++;
      initialize_adjacency (match, label);

      if (compute_matches (match) == EOF)
        return EOF;

      match->total = 0;
    }

    copy_clique (&match->clique, &match->clique_sort[match->total]);
    match->total++;
    return TRUE;
  }

  else
  {
    if (orientation_id == 0)
    {
      match->cycle++;
      initialize_adjacency (match, label);
    }

    if (match_distance (match) != EOF)
    {
      match->total++;
      return TRUE;
    }

    else
    {
      if (!dock->multiple_ligands || !dock->multiple_conforms)
        output_match_info (match);

      return EOF;
    }
  }
}


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

Routine to free matching arrays
5/97 te

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

void free_matches
(
  LABEL         *label,
  MATCH         *match
)
{
  free_clique_list (match);
  free_clique_link (match);
  free_match (match);
  free_match_site (match, label);
  free_ligand_centers (match);
}


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

Routine to allocate space for matching arrays
3/96 te

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

void init_match_site
(
  MATCH         *match,
  LABEL         *label
)
{
/*
* Construct match table
* 11/96 te
*/
  if (match->chemical_flag && !label->chemical.match_table)
    get_table
    (
      &label->chemical,
      label->chemical.match_file_name,
      &label->chemical.match_table
    );

/*
* Read in site points for matching
* 8/96 te
*/
  get_site (match, label);
  get_centers (match, label);

/*
* Construct distance matrix for receptor site points
* 6/95 te
*/
  match->bin_length =
    calculate_distances
    (
      &match->receptor_site,
      &match->receptor_distances,
      &match->receptor_distance_size
    );

/*
* Make keys for pharmacophore search
* 1/97 te
*/
  if (label->chemical.screen.pharmaco_flag)
  {
    if (label->chemical.screen.fold_flag)
    {
      update_folded_keys (&label->chemical, &match->receptor_site, 0);
      return;
    }

    else
      update_unfolded_keys (&label->chemical, &match->receptor_site, 0);
  }

  ecalloc
  (
    (void **) &match->clique_adjacency_total,
    match->clique_size_max + 1,
    sizeof (int),
    "clique adjacency list total",
    global.outfile
  );

  ecalloc
  (
    (void **) &match->clique_adjacency_list,
    match->clique_size_max + 1,
    sizeof (EDGE *),
    "clique adjacency list",
    global.outfile
  );

  ecalloc
  (
    (void **) &match->adjacent,
    match->clique_size_max + 1,
    sizeof (int),
    "adjacent iterator list",
    global.outfile
  );

  match->clique.edge_max = match->clique_size_max;
  allocate_clique (&match->clique);

  ecalloc
  (
    (void **) &match->degenerate,
    match->clique_size_max,
    sizeof (int),
    "degenerate clique list",
    global.outfile
  );

  emalloc
  (
    (void **) &match->histogram,
    match->clique_size_max * sizeof (int),
    "clique histogram",
    global.outfile
  );

/*
* Update the critical cluster identifiers of the site points
* 7/95 te
*/
  id_site_points (match);

/*
* Reorder site points so that critical clusters come first
* 7/95 te
*/
  if (match->critical_flag)
    order_site_points (match);

/*
* Allocate memory for the critical cluster filters
* 7/95 te
*/
  ecalloc
  (
    (void **) &match->critical_filter,
    match->cluster_total + 1,
    sizeof (char *),
    "matching filter",
    global.outfile
  );

/*
* Construct matrix needed to calculate chirality
* 10/95 te
*/
  calculate_vectors
  (
    &match->receptor_site,
    &match->receptor_vectors,
    &match->receptor_vector_size
  );
```

```
/*
* Construct receptor distance bins
* 1/97 te
*/
  make_distance_bins (match);

  allocate_clique_atoms (match);

  match->init_flag = TRUE;
}


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

Routine to free space for matching arrays
4/97 te

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

void free_match_site
(
  MATCH         *match,
  LABEL         *label
)
{
/*
* Free match table
* 5/97 te
*/
  if (match->chemical_flag && label->chemical.match_table)
    free_table
    (
      &label->chemical,
      &label->chemical.match_table
    );

/*
* Free site points for matching
* 5/97 te
*/
  free_molecule (&match->receptor_site);

  free_distances
  (
    &match->receptor_distances,
    &match->receptor_distance_size
  );

  efree ((void **) &match->clique_adjacency_total);
  efree ((void **) &match->clique_adjacency_list);
  efree ((void **) &match->adjacent);
  free_clique (&match->clique);
  efree ((void **) &match->degenerate);
  efree ((void **) &match->histogram);

  if (match->critical_flag)
    free_site_points (match);

  efree ((void **) &match->critical_filter);

  free_vectors
  (
    &match->receptor_vectors,
    &match->receptor_vector_size
  );

  free_distance_bins (match);
  free_molecule (&match->ligand_clique);
  free_molecule (&match->receptor_clique);
}


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

Routine to allocate space for matching arrays
3/96 te

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

void allocate_match (MATCH *match)
{
  int i;

  ecalloc
  (
    (void **) &match->adjacency_total,
    match->node_max,
    sizeof (int),
    "adjacency list total",
    global.outfile
  );

  ecalloc
  (
    (void **) &match->adjacency_max,
    match->node_max,
    sizeof (int),
    "adjacency list max",
    global.outfile
  );

  ecalloc
  (
    (void **) &match->adjacency_list,
    match->node_max,
    sizeof (EDGE *),
    "adjacency matrix",
    global.outfile
  );

  for (i = 0; i < match->clique_size_max + 1; i++)
    ecalloc
    (
      (void **) &match->clique_adjacency_list[i],
      match->node_max,
      sizeof (EDGE),
      "clique adjacency matrix",
      global.outfile
    );

  for (i = 0; i < match->node_max; i++)
  {
    match->clique_adjacency_list[0][i].node = i;
    match->clique_adjacency_list[0][i].residual = 0;
  }

/*
* Allocate memory for and initialize the chemical matching filter
* 7/95 te
*/
  emalloc
  (
    (void **) &match->chemical_filter,
    match->node_max * sizeof (char),
    "chemical label filter",
    global.outfile
  );

  memset (match->chemical_filter, 1, match->node_max);

/*
* Allocate memory for the critical cluster filters
* 7/95 te
*/
  for (i = 0; i < match->cluster_total + 1; i++)
    ecalloc
    (
      (void **) &match->critical_filter[i],
      match->node_max,
      sizeof (char),
      "matching filter",
      global.outfile
    );
}


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

Routine to reset match variables.
11/96 te

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

void reset_match (MATCH *match)
{
  int i;

  memset (match->adjacency_total, 0, match->node_max * sizeof (int));

  memset
    (match->clique_adjacency_total, 0,
    (match->clique_size_max + 1) * sizeof (int));

  for (i = 1; i < match->clique_size_max + 1; i++)
    memset
      (match->clique_adjacency_list[i], 0,
      match->node_max * sizeof (EDGE));

  reset_clique (&match->clique);
  memset (match->adjacent, 0, (match->clique_size_max + 1) * sizeof (int));
  memset (match->degenerate, 0, match->clique_size_max * sizeof (int));
  memset (match->histogram, 0, match->clique_size_max * sizeof (int));
/*
  memset (match->chemical_filter, 1, match->node_max);

  for (i = 0; i < match->cluster_total + 1; i++)
    memset (match->critical_filter[i], 1, match->node_max);
*/
}

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

Routine to free match variables.
11/96 te

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

void free_match (MATCH *match)
{
  int i;

  efree ((void **) &match->adjacency_total);
  efree ((void **) &match->adjacency_max);

  for (i = 0; i < match->node_max; i++)
    efree ((void **) &match->adjacency_list[i]);

  efree ((void **) &match->adjacency_list);

  if (match->clique_adjacency_list)
    for (i = 0; i < match->clique_size_max + 1; i++)
      efree ((void **) &match->clique_adjacency_list[i]);

  efree ((void **) &match->chemical_filter);

  if (match->critical_filter)
    for (i = 0; i < match->cluster_total + 1; i++)
      efree ((void **) &match->critical_filter[i]);

  match->node_max = 0;
}


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

Routine to reallocate match variables.
3/97 te

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

void reallocate_match (MATCH *match)
{
  if (match->node_max < match->node_total)
  {
    free_match (match);
    match->node_max = match->node_total;
    allocate_match (match);
  }
}


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

void allocate_clique_atoms (MATCH *match)
{
  match->ligand_clique.max.atoms = match->clique_size_max;
  allocate_molecule (&match->ligand_clique);

  match->receptor_clique.max.atoms = match->clique_size_max;
  allocate_molecule (&match->receptor_clique);
}


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

void allocate_clique (CLIQUE *clique)
{
```

```
  if (clique->edge_max > 0)
    ecalloc
    (
      (void **) &clique->edge,
      clique->edge_max,
      sizeof (EDGE),
      "clique edges",
      global.outfile
    );
}

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

void reallocate_clique (CLIQUE *clique)
{
  if (clique->edge_max < clique->edge_total)
  {
    free_clique (clique);
    clique->edge_max = clique->edge_total;
    allocate_clique (clique);
  }
}

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

void free_clique (CLIQUE *clique)
{
  if (clique->edge_max > 0)
    efree ((void **) &clique->edge);

  clique->edge_max = 0;
}

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

void reset_clique (CLIQUE *clique)
{
  if (clique->edge_max > 0)
    memset (clique->edge, 0, clique->edge_max * sizeof (EDGE));

  clique->edge_total = 0;
  clique->residual = 0;
  clique->reflect_flag = FALSE;
}

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

void copy_clique (CLIQUE *copy, CLIQUE *original)
{
  int i;

  copy->edge_total = original->edge_total;
  reallocate_clique (copy);

  for (i = 0; i < original->edge_total; i++)
    copy->edge[i] = original->edge[i];

  copy->residual = original->residual;
  copy->reflect_flag = original->reflect_flag;
}

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

Routine to set up arrays used during matching.
7/95 te

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

int init_match_ligand
(
  MATCH         *match,
  LABEL         *label
)
{
  match->node_total =
    match->ligand_center.total.atoms * match->receptor_site.total.atoms;

  reallocate_match (match);

/*
* If chemical matching is performed, construct new chemical filter
* 7/95 te
*/
  if (match->chemical_flag)
    make_chemical_filter (&label->chemical, match);

/*
* If critical cluster matching is performed, construct new filter
* to guide matching
* 6/95 te
*/
  if (match->critical_flag)
    make_critical_filter (match);

  return TRUE;
}


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

Routine to set up arrays used during matching.
7/95 te

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

int initialize_adjacency
(
  MATCH         *match,
  LABEL         *label
)
{
  reset_match (match);

  if (label->chemical.screen.process_flag)
    compute_chemical_adjacency (match, label);

  else
    compute_adjacency (match);

  match->clique_adjacency_total[0] = match->node_total;

  return TRUE;
}

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

Routine to get site points
11/96 te

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

void get_site
(
  MATCH         *match,
  LABEL         *label
)
{
  FILE *file;

/*
* Read in receptor site points
* 3/96 te
*/
  file = efopen (match->receptor_file_name, "r", global.outfile);

  if (read_molecule
    (&match->receptor_site, NULL, match->receptor_file_name, file, TRUE)
    != TRUE)
    exit (fprintf (global.outfile,
      "ERROR get_site: Unable to read receptor site points from %s\n",
      match->receptor_file_name));

  if (match->clique_size_min > match->receptor_site.total.atoms)
    exit (fprintf (global.outfile,
      "ERROR get_site: clique_size_min > number of site points.\n"));

  if (prepare_molecule
    (
      &match->receptor_site,
      match->receptor_file_name,
      file,
      label,
      'a',
      FALSE,
      match->chemical_flag,
      FALSE
    ) != TRUE)
    exit (fprintf (global.outfile,
      "ERROR get_site: Unable to prepare receptor site points.\n"));

  fclose (file);
}


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

Routine to get (or just allocate) ligand centers
3/96 te

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

void get_centers
(
  MATCH         *match,
  LABEL         *label
)
{
  FILE *file;

/*
* If requested, read in ligand centers
* 3/96 te
*/
  if (match->centers_flag)
  {
    file = efopen (match->ligand_file_name, "r", global.outfile);

    if (read_molecule
      (
        &match->ligand_center,
        NULL,
        match->ligand_file_name,
        file,
        TRUE
      ) != TRUE)
      exit (fprintf (global.outfile,
        "ERROR get_centers: Unable to read ligand site points from %s\n",
        match->ligand_file_name));

    if (prepare_molecule
      (
        &match->ligand_center,
        match->ligand_file_name,
        file,
        label,
        'a',
        FALSE,
        match->chemical_flag,
        FALSE
      ) != TRUE)
      exit (fprintf (global.outfile,
        "ERROR get_centers: Unable to prepare ligand site points.\n"));

    center_of_mass
    (
      match->ligand_center.coord,
      match->ligand_center.total.atoms,
      match->ligand_center.transform.com
    );

    fclose (file);

    calculate_distances
    (
      &match->ligand_center,
      &match->ligand_distances,
      &match->ligand_distance_size
    );

    calculate_vectors
    (
      &match->ligand_center,
      &match->ligand_vectors,
      &match->ligand_vector_size
    );
  }
}


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

Routine to fix the critical cluster identifiers of the site points
so that the zeroth cluster (unclustered site points) becomes the
last cluster.
7/95 te

////////////////////////////////////////////////////////////// */
```

```
void id_site_points (MATCH *match)
{
  int i;

  if (match->critical_flag)
  {
    match->cluster_total = match->receptor_site.total.substs;

/*
*   Make sure the largest cluster is not greater than the largest
*   allowed clique size
*   7/95 te
*/
    if (match->cluster_total > match->clique_size_max)
      exit (fprintf (global.outfile,
        "ERROR id_site_points: "
        "Critical cluster identifiers cannot exceed nodes_maximum,\n"
        "       or if a zeroth cluster is present, nodes_maximum - 1.\n"));
  }

/*
* If critical cluster filtering is not to be performed, then
* set all cluster identifiers to zero.
* 7/95 te
*/
  else
  {
    for (i = 0; i < match->receptor_site.total.atoms; i++)
      match->receptor_site.atom[i].subst_id = 0;

    match->cluster_total = 0;
  }
}


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

Routine to reshuffle order of the site points to get the members of
critical clusters first
7/95 te

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

void order_site_points (MATCH *match)
{
  int i, j;
  MOLECULE temporary = {0};
  int compare_site_points ();

/*
* Allocate space for the cluster lists
* 11/96 te
*/
  ecalloc
  (
    (void **) &match->cluster,
    match->cluster_total,
    sizeof (int *),
    "cluster list",
    global.outfile
  );

  ecalloc
  (
    (void **) &match->cluster_size,
    match->cluster_total,
    sizeof (int),
    "cluster list",
    global.outfile
  );

/*
* Determine the size of each cluster
* 11/96 te
*/
  for (i = 0; i < match->receptor_site.total.atoms; i++)
    match->cluster_size[match->receptor_site.atom[i].subst_id]++;

  for (i = 0; i < match->cluster_total; i++)
    ecalloc
    (
      (void **) &match->cluster[i],
      match->cluster_size[i],
      sizeof (int),
      "cluster list",
      global.outfile
    );

/*
* Allocate memory for space to reorder site points
* 7/95 te
*/
  temporary.max.atoms = match->receptor_site.total.atoms;
  vstrcpy (&temporary.info.name, "temporary");
  allocate_molecule (&temporary);

/*
* Copy over site points in critical clusters
* 7/95 te
*/
  for (i = temporary.total.atoms = 0; i < match->cluster_total; i++)
    for (j = match->cluster_size[i] = 0;
      j < match->receptor_site.total.atoms; j++)
      if (match->receptor_site.atom[j].subst_id == i)
      {
        copy_atom
          (&temporary.atom[temporary.total.atoms],
          &match->receptor_site.atom[j]);
        copy_coord
          (temporary.coord[temporary.total.atoms],
          match->receptor_site.coord[j]);

        match->cluster[i][match->cluster_size[i]++] = temporary.total.atoms++;
      }

/*
* Check to make the correct number of atoms were copied
* 7/95 te
*/
  if (match->receptor_site.total.atoms != temporary.total.atoms)
    exit (fprintf (global.outfile,
      "ERROR order_site_points: "
      "Critical cluster identifiers must be in the range "
      "[0, nodes_maximum].\n"));

/*
* Copy the reordered site points back into molecular structure
* 7/95 te
*/
  copy_atoms (&match->receptor_site, &temporary);
  free_molecule (&temporary);

/*
* Check to make sure the first cluster is identified as number 1
* 7/95 te
*/
  if (match->receptor_site.atom[0].subst_id != 0)
    exit (fprintf (global.outfile,
      "ERROR order_site_points: "
      "Critical cluster identifiers must begin at 1.\n"));

/*
* Check to make sure the clusters are numbered sequentially with no gaps
* 7/95 te
*/
  for (i = 1; i < match->receptor_site.total.atoms; i++)
    if ((match->receptor_site.atom[i].subst_id !=
      match->receptor_site.atom[i - 1].subst_id) &&
      (match->receptor_site.atom[i].subst_id !=
      match->receptor_site.atom[i - 1].subst_id + 1))
      exit (fprintf (global.outfile,
        "ERROR order_site_points: "
        "Critical cluster identifiers cannot have gaps.\n"));

/*
* Print out critical clusters
* 7/95 te
*/
  fprintf (global.outfile, "____Critical_Clusters____\n");

  fprintf (global.outfile, "%5s %5s %5s\n", "clust", "name", "type");
  for (i = 0; (i < match->receptor_site.total.atoms) &&
    (match->receptor_site.atom[i].subst_id <= match->cluster_total); i++)
    fprintf (global.outfile, "%5d %5s %5s\n",
      match->receptor_site.atom[i].subst_id + 1,
      match->receptor_site.atom[i].name, match->receptor_site.atom[i].type);

  fprintf (global.outfile, "\n\n");
}


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

Routine to free critical clusters
5/97 te

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

void free_site_points (MATCH *match)
{
  int i;

/*
* Free space for the cluster lists
* 11/96 te
*/
  efree ((void **) &match->cluster);
  efree ((void **) &match->cluster_size);

  for (i = 0; i < match->cluster_total; i++)
    efree ((void **) &match->cluster[i]);
}



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

Routine to construct a vector matrix: a N by N matrix in which each
element is a difference vector between the positions of each of N atoms
or site points.  This matrix is constructed to speed up the clique
chirality checking algorithms.
7/95 te

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

void calculate_vectors (MOLECULE *molecule, XYZ ***matrix, int *size)
{
  int i, j, k;

/*
* Either allocate space for difference vectors, or reset space
* 3/97 te
*/
  if (*size < molecule->total.atoms)
  {
    free_vectors (matrix, size);

    *size = molecule->total.atoms;

    ecalloc
    (
      (void **) matrix,
      *size,
      sizeof (XYZ *),
      "vectors matrix",
      global.outfile
    );

    for (i = 0; i < *size; i++)
      ecalloc
      (
        (void **) &(*matrix)[i],
        *size,
        sizeof (XYZ),
        "vectors matrix",
        global.outfile
      );
  }

  else
    for (i = 0; i < *size; i++)
      memset ((*matrix)[i], 0, *size * sizeof (XYZ));

/*
* Loop through all pairs of atoms/site points
* 3/97 te
*/
  for (i = 0; i < molecule->total.atoms; i++)
    for (j = 0; j < molecule->total.atoms; j++)
      for (k = 0; k < 3; k++)
        (*matrix)[i][j][k] =
          molecule->coord[j][k] -
          molecule->coord[i][k];
}


/* //////////////////////////////////////////////////////////////// */
```

```
void free_vectors (XYZ ***matrix, int *size)
{
  int i;

  for (i = 0; i < *size; i++)
    efree ((void **) &(*matrix)[i]);

  efree ((void **) matrix);
  *size = 0;
}


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

Routine to construct the receptor distance bins used in match_driver.
1/97 te

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

void make_distance_bins (MATCH *match)
{
  int i, j, k;
  SLINT2 **current = NULL;
  SLINT2 **previous = NULL;

/*
* Allocate space for receptor distance bins
* 1/97 te
*/
  match->bin_width = 0.5;
  match->bin_total = NINT (match->bin_length / match->bin_width) + 1;

  ecalloc
  (
    (void **) &match->bin,
    match->bin_total,
    sizeof (SLINT2 *),
    "receptor distance bins",
    global.outfile
  );

  ecalloc
  (
    (void **) &previous,
    match->bin_total,
    sizeof (SLINT2 *),
    "receptor distance bins",
    global.outfile
  );

  ecalloc
  (
    (void **) &current,
    match->bin_total,
    sizeof (SLINT2 *),
    "receptor distance bins",
    global.outfile
  );

/*
* Divide receptor distances into bins
* 1/97 te
*/
  for (i = 0; i < match->receptor_site.total.atoms; i++)
    for (j = i + 1; j < match->receptor_site.total.atoms; j++)
    {
      k = NINT (match->receptor_distances[i][j] / match->bin_width);

      ecalloc
      (
        (void **) &current[k],
        1,
        sizeof (SLINT2),
        "next bin occupant",
        global.outfile
      );

      current[k]->i = i;
      current[k]->j = j;

      if (match->bin[k] != NULL)
        previous[k]->next = current[k];

      else
        match->bin[k] = current[k];

      previous[k] = current[k];
      current[k] = NULL;
    }

  efree ((void **) &current);
  efree ((void **) &previous);
}


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

Routine to free the receptor distance bins used in match_driver.
5/97 te

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

void free_distance_bins (MATCH *match)
{
  int i;
  SLINT2 *prev = NULL;
  SLINT2 *next = NULL;

  for (i = 0; i < match->bin_total; i++)
    for (prev = match->bin[i]; prev != NULL; prev = next)
    {
      next = prev->next;
      efree ((void **) &prev);
    }

  match->bin_total = 0;
  efree ((void **) &match->bin);
}


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

Routine to extract ligand centers from a ligand
3/96 te

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

int get_ligand_centers
(
  MATCH         *match,
  MOLECULE      *molecule
)
{
  int atom;
  int segment;
  int layer;

  match->ligand_center.total.atoms = molecule->total.atoms;
  reallocate_atoms (&match->ligand_center);

/*
* Allocate space for the ligand atom key
* 3/97 te
*/
  if (match->ligand_key_total < match->ligand_center.total.atoms)
  {
    efree ((void **) &match->ligand_key);
    match->ligand_key_total = match->ligand_center.total.atoms;

    ecalloc
    (
      (void **) &match->ligand_key,
      match->ligand_key_total,
      sizeof (int),
      "match ligand key",
      global.outfile
    );
  }

/*
* Copy the anchor heavy atoms over
* 3/96 te
*/
  for (atom = match->ligand_center.total.atoms = 0;
    atom < molecule->total.atoms; atom++)
  {
    if
    (
      ((segment = molecule->atom[atom].segment_id) != NEITHER) &&
      (segment < molecule->total.segments) &&
      ((layer = molecule->segment[segment].layer_id) != NEITHER) &&
      (layer < molecule->total.layers) &&
      (layer != 0)
    )
      continue;

    if (molecule->atom[atom].heavy_flag != TRUE)
      continue;

/*
    fprintf (global.outfile, "Lig cent %s, seg %d, layer %d\n",
      molecule->atom[atom].name, segment, layer);

    if (match->ligand_center.total.atoms >= match->ligand_center.max.atoms)
      exit (fprintf (global.outfile,
        "ERROR get_ligand_centers: Too many centers in molecule\n"));
*/

    copy_atom
      (&match->ligand_center.atom[match->ligand_center.total.atoms],
      &molecule->atom[atom]);
    copy_coord
      (match->ligand_center.coord[match->ligand_center.total.atoms],
      molecule->coord[atom]);

    match->ligand_key[match->ligand_center.total.atoms] = atom;
    match->ligand_center.total.atoms++;
  }

  copy_transform (&match->ligand_center, molecule);

/*
* Calculate distance matrix
* 3/97 te
*/
  calculate_distances
  (
    &match->ligand_center,
    &match->ligand_distances,
    &match->ligand_distance_size
  );

/*
* Compute vectors for chirality checking
* 1/97 te
*/
  calculate_vectors
  (
    &match->ligand_center,
    &match->ligand_vectors,
    &match->ligand_vector_size
  );

  return TRUE;
}

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

Routine to free ligand centers
5/97 te

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

void free_ligand_centers (MATCH *match)
{
  free_molecule (&match->ligand_center);
  efree ((void **) &match->ligand_key);

  free_distances
  (
    &match->ligand_distances,
    &match->ligand_distance_size
  );

  free_vectors
  (
    &match->ligand_vectors,
    &match->ligand_vector_size
  );
}

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

Routine to extract ligand keys from a ligand
3/96 te

//////////////////////////////////////////////////////////////// */
```

```
int get_ligand_keys
(
  MATCH         *match,
  MOLECULE      *molecule
)
{
  int i;

  if (molecule->transform.fold_flag == TRUE)
    exit (fprintf (global.outfile,
      "ERROR get_ligand_keys: ligand keys are folded\n"));

  if (molecule->total.atoms > match->ligand_center.max.atoms)
    exit (fprintf
      (global.outfile, "ERROR get_ligand_keys: too many ligand atoms\n"));

  match->ligand_center.total.atoms = molecule->total.atoms;

  for (i = 0; i < molecule->total.atoms; i++)
    match->ligand_center.atom[i].chem_id = molecule->atom[i].chem_id;

  copy_keys (&match->ligand_center, molecule);

  return TRUE;
}

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

Routine to make the chemical matching filter.
7/95 te

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

void make_chemical_filter
(
  LABEL_CHEMICAL *label_chemical,
  MATCH         *match
)
{
  int i, j, node;

  for (i = node = 0; i < match->receptor_site.total.atoms; i++)
    for (j = 0; j < match->ligand_center.total.atoms; j++, node++)
      match->chemical_filter[node] = (char) label_chemical->match_table
        [match->ligand_center.atom[j].chem_id]
        [match->receptor_site.atom[i].chem_id];
}


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

Routine to construct the critical cluster filter used during matching.
7/95

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

void make_critical_filter (MATCH *match)
{
  int i, j;

/*
* Reset critical filters
* 7/95 te
*/
  for (i = 0; i < match->cluster_total; i++)
    memset (match->critical_filter[i], 0, match->node_max);

  memset (match->critical_filter[match->cluster_total], 1, match->node_max);

/*
* Turn ON selected elements in filters
* 7/95 te
*/
  for (i = 0; i < match->receptor_site.total.atoms; i++)
    for (j = i * match->ligand_center.total.atoms;
      j < (i + 1) * match->ligand_center.total.atoms; j++)
      match->critical_filter[match->receptor_site.atom[i].subst_id][j] = 1;

/*
* If user wants MULTIPLE selections from each critical cluster,
* then make each screen INCLUDE selection from all preceding clusters
* 7/95 te
*/
  if (match->multiple_flag)
    for (i = 0; i < match->cluster_total - 1; i++)
      for (j = 0; j < match->node_max; j++)
        match->critical_filter[i + 1][j] |=
          match->critical_filter[i][j];
}

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

Routine to construct the node adjacency matrices used to guide clique
formation during the matching routine.
7/95 te

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

void compute_adjacency (MATCH *match)
{
  int li, lj;           /* Ligand atom iterators */
  int ri, rj;           /* Receptor point iterators */
  int inode, jnode;     /* Node values */

  int bin_id;           /* Bin position */
  int bin_tolerance;    /* Tolerance in bin position */
  int bin_center;       /* Center bin to query */
  int bin_initial;      /* Initial bin to query */
  int bin_final;        /* Final bin to query */
  SLINT2 *bin;          /* Bin pointer */

  float residual;       /* Difference in distances */

  int compare_nodes ();

  bin_tolerance = (int)
    (match->cycle * match->distance_tolerance / match->bin_width + .9999);

/*
* Loop through all possible ligand center pairs
* 7/95 te
*/
  for (li = 0; li < match->ligand_center.total.atoms; li++)
    for (lj = li + 1; lj < match->ligand_center.total.atoms; lj++)
    {
/*
*     Check first geometric criteria for adjacency
*     7/95 te
*/
      if (match->ligand_distances[li][lj] < match->distance_minimum)
        continue;

/*
*     Determine which receptor distance bins to check
*     1/97 te
*/
      bin_center =
        NINT (match->ligand_distances[li][lj] / match->bin_width);
      bin_initial = MAX (0, bin_center - bin_tolerance);
      bin_final = MIN (match->bin_total, bin_center + bin_tolerance + 1);

/*
*     Loop through all retrieved site point pairs
*     1/97 te
*/
      for (bin_id = bin_initial; bin_id < bin_final; bin_id++)
        for (bin = match->bin[bin_id]; bin != NULL; bin = bin->next)
        {
/*
*         Extract the site point id
*         1/97 te
*/
          ri = bin->i;
          rj = bin->j;

/*
*         Determine if the two receptor site points can join the two
*         ligand centers to form two adjacent nodes.  Node adjacency is
*         determined by the following geometric criteria:
*
*           1) satisfies "distance_tolerance"
*           2) satisfies "distance_minimum"
*
*         1/97 te
*/
          residual =
            match->ligand_distances[li][lj] -
            match->receptor_distances[ri][rj];

          residual = ABS (residual);

          if ((residual > match->cycle * match->distance_tolerance) ||
            (match->receptor_distances[ri][rj] < match->distance_minimum))
            continue;

/*
*         Compute the first two of four possible nodes and update
*         adjacency matrices.
*         7/95 te
*/
          inode = ri * match->ligand_center.total.atoms + li;
          jnode = rj * match->ligand_center.total.atoms + lj;

          append_adjacency (match, inode, jnode, residual);

          if (match->degeneracy_flag)
            append_adjacency (match, jnode, inode, residual);

/*
*         Compute the second two of four possible nodes and update
*         adjacency matrices.
*         7/95 te
*/
          inode = ri * match->ligand_center.total.atoms + lj;
          jnode = rj * match->ligand_center.total.atoms + li;

          append_adjacency (match, inode, jnode, residual);

          if (match->degeneracy_flag)
            append_adjacency (match, jnode, inode, residual);
        }
    }

/*
* Sort each adjacency list by ascending order
* 7/95 te
*/
  for (li = 0; li < match->node_total; li++)
  {
    if (!match->chemical_flag || match->chemical_filter[li])
      qsort
      (
        match->adjacency_list[li],
        match->adjacency_total[li],
        sizeof (EDGE),
        compare_nodes
      );
  }

/*
  for (li = 0; li < match->node_total; li++)
  {
    fprintf (global.outfile, "node %d:", li);
    for (lj = 0; lj < match->adjacency_total[li]; lj++)
      fprintf (global.outfile, " %d", match->adjacency_list[li][lj].node);
    fprintf (global.outfile, "\n");
  }
*/
}


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

Routine to construct the node adjacency matrices used to guide clique
formation during the matching routine (for chemical screening).
7/95 te

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

void compute_chemical_adjacency (MATCH *match, LABEL *label)
{
  int li, lj;           /* Ligand atom iterators */
  int ri, rj;           /* Receptor point iterators */
  int inode, jnode;     /* Node values */
  int lil, ljl;         /* Label id of ligand iterators */
  int ril, rjl;         /* Label id of receptor iterators */

  int compare_nodes ();

  update_uncertainty
  (
    &label->chemical,
    match->distance_tolerance,
    &match->ligand_center
  );

/*
```

```
* Loop through all possible ligand center pairs
* 11/96 te
*/
  for (ri = 0; ri < match->receptor_site.total.atoms; ri++)
  {
    ril = match->receptor_site.atom[ri].chem_id;

    for (rj = ri + 1; rj < match->receptor_site.total.atoms; rj++)
    {
      if (match->receptor_distances[ri][rj] < match->distance_minimum)
        continue;

      rjl = match->receptor_site.atom[rj].chem_id;

      for (li = 0; li < match->ligand_center.total.atoms; li++)
      {
        lil = match->ligand_center.atom[li].chem_id;

        if (!label->chemical.match_table[ril][lil])
          continue;

        for (lj = li + 1; lj < match->ligand_center.total.atoms; lj++)
        {
          ljl = match->ligand_center.atom[lj].chem_id;

          if (!label->chemical.match_table[rjl][ljl])
            continue;

          if (!(match->receptor_site.key[ri][rj].distance &
            match->ligand_center.key[li][lj].distance))
            continue;

          inode = ri * match->ligand_center.total.atoms + li;
          jnode = rj * match->ligand_center.total.atoms + lj;

          append_adjacency (match, inode, jnode, 0);
        }
      }
    }
  }

/*
* Sort each adjacency list by ascending order
* 11/96 te
*/
  for (li = 0; li < match->node_total; li++)
  {
    if (match->chemical_filter[li])
      qsort
      (
        match->adjacency_list[li],
        match->adjacency_total[li],
        sizeof (EDGE),
        compare_nodes
      );
  }
}


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

Routine used by qsort to determine the order of the adjacency lists.
7/95 te

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

int compare_nodes
(
  EDGE *pta,
  EDGE *ptb
)
{
  return (pta->node == ptb->node) ? 0 : ((pta->node > ptb->node) ? 1 : -1);
}


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

Routine to add a node to an adjacency list.
11/96 te

//////////////////////////////////////////////////////////////// */
void append_adjacency (MATCH *match, int list, int node, float residual)
{
  if (match->adjacency_total[list] + 1 >
    match->adjacency_max[list])
  {
    match->adjacency_max[list] += 10;

    erealloc
    (
      (void **) &match->adjacency_list[list],
      match->adjacency_max[list] * sizeof (EDGE),
      "adjacency list",
      global.outfile
    );
  }

  match->adjacency_list
    [list][match->adjacency_total[list]].node = node;
  match->adjacency_list
    [list][match->adjacency_total[list]].residual = residual;
  match->adjacency_total[list]++;
}

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

Routine to assemble all matches for a given distance tolerance

Return values:
        TRUE    matches recorded
        EOF     insufficient matches recorded (terminate orienting)

1/97 te

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

int compute_matches (MATCH *match)
{
  int i;
  int previous_max;             /* Record of the number of previous matches */
  SLCLIQUE *current = NULL;     /* Pointer to current clique */
  SLCLIQUE *previous = NULL;    /* Pointer to previous clique */

  int compare_cliques ();

/*
* Loop through all matches
* 1/97 te
*/
  for
  (
    previous_max = match->max, match->max = 0;
    match_distance (match) != EOF;
    match->max++
  )
  {
    ecalloc
    (
      (void **) &current,
      1,
      sizeof (SLCLIQUE),
      "clique link list",
      global.outfile
    );

    copy_clique (&current->clique, &match->clique);

    if (match->clique_link != NULL)
      previous->next = current;

    else
      match->clique_link = current;

    previous = current;
    current = NULL;
  }

/*
* Copy cliques over to linear array and sort them
* 1/97 te
*/
  if (match->clique_sort_total < match->max)
  {
    free_clique_list (match);
    match->clique_sort_total = match->max;

    ecalloc
    (
      (void **) &match->clique_sort,
      match->clique_sort_total,
      sizeof (CLIQUE),
      "clique sort list",
      global.outfile
    );
  }

  for
  (
    i = 0, current = match->clique_link;
    (i < match->max) && (current != NULL);
    i++, current = current->next
  )
    copy_clique (&match->clique_sort[i], &current->clique);

  if (i != match->max)
    exit (fprintf (global.outfile,
      "ERROR compute_matches: Insufficient cliques copied\n"));

  qsort
  (
    match->clique_sort,
    match->max,
    sizeof (CLIQUE),
    compare_cliques
  );

  free_clique_link (match);

/*
* Check if sufficient matches were made
* 1/97 te
*/
  if
  (
    (match->cycle >= 10) &&
    ((match->max == 0) ||
    (match->max <= previous_max))
  )
    return EOF;

  else
    return TRUE;
}


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

void free_clique_list (MATCH *match)
{
  int i;

  for (i = 0; i < match->clique_sort_total; i++)
    free_clique (&match->clique_sort[i]);

  efree ((void **) &match->clique_sort);
  match->clique_sort_total = 0;
}


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

void free_clique_link (MATCH *match)
{
  SLCLIQUE *current = NULL;     /* Pointer to current clique */
  SLCLIQUE *previous = NULL;    /* Pointer to previous clique */

/*
* Free up linked list
* 3/97 te
*/
  for
  (
    previous = match->clique_link;
    previous != NULL;
    previous = current
  )
  {
    current = previous->next;
    free_clique (&previous->clique);
    efree ((void **) &previous);
  }

  match->clique_link = NULL;
}

int compare_cliques
(
```

```
  CLIQUE *pta,
  CLIQUE *ptb
)
{
  return (pta->residual == ptb->residual) ? 0 :
    ((pta->residual > ptb->residual) ? 1 : -1);
}


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

Routine to generate ligand orientations by finding subsets of
ligand atoms to superimpose on receptor site points.

1.  At start, no nodes are in clique, and all nodes are available
    in clique adjacency list.

2.  Successive nodes are taken from the initial clique adjacency list
    to seed the clique.

3.  A new adjacency list is formed by the intersection of the previous
    adjacency list and the new node adjacency list.

4.  The clique is grown in a depth-first fashion until no more nodes
    are available in the adjacency list.  If the clique is large enough,
    it is returned as a match.

5.  Backtracking is implemented by successively removing nodes from
    the clique to try alternative nodes from the previous adjacency list.

6.  When backtracking has exhausted all nodes in the clique and all
    nodes in the initial clique adjacency list, then matching terminates.

Return values:
        TRUE:   suitable match found
        EOF:    no more matches can be formed

11/96 te

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

int match_distance (MATCH *match)
{
  int i, j;                     /* Counter variable */
  EDGE edge;                    /* Edge under consideration */
  EDGE iedge, jedge;            /* Edge comparisons */
  int valid_clique;             /* Flag for whether an valid_clique found */

  int match_chirality (MATCH *, int);

/*
* If a prior clique existed, then erase the last node and
* decrement the clique size
* 11/96 te
*/
  if (match->clique.edge_total > 0)
  {
    match->clique.edge_total--;
    match->clique.edge[match->clique.edge_total].node = 0;
    match->clique.edge[match->clique.edge_total].residual = 0;
  }

/*
* Perform clique detection until valid clique is found
* 11/96 te
*/
  for (valid_clique = FALSE; !valid_clique;)
  {
/*
*   Can we expand the current clique?  Check if:
*   1. The clique is smaller than the maximum size
*   2. There are unused nodes in the clique adjacency list
*   11/96 te
*/
    if
    (
      (match->clique edge_total < match->clique_size_max) &&
      (match->adjacent[match->clique edge_total] <
        match->clique_adjacency_total[match->clique.edge_total])
    )
    {
/*
*     Identify the next adjacent node and increment the adjacency list
*     11/96 te
*/
      edge = match->clique_adjacency_list
        [match->clique.edge_total]
        [match->adjacent[match->clique.edge_total]++];

/*
*     STAGE 1: The node is geometrically feasible, if
*     1. The node passes the critical cluster filter:
*        a. The filter is not requested, OR
*        b. The filter indicates valid node, AND
*     2. The clique has proper chirality:
*        a. Chirality checking is not necessary, OR
*        b. The node is not being added to a three-node clique, OR
*        c. The chirality of the new clique is correct, OR
*        d. The ligand is allowed to be reflected
*     11/96 te
*/
      if
      (
        (
          !match->critical_flag ||
          (((match->clique.edge_total == 0) &&
            match->critical_filter[0][edge.node]) ||
          ((match->clique.edge_total > 0) &&
            match->critical_filter
              [match->receptor_site.atom
                [match->clique.edge[match->clique.edge_total - 1].node /
                  match->ligand_center.total.atoms].subst_id + 1][edge.node]))
        ) &&
        (
          !match->chiral_flag ||
          (match->clique.edge_total != 3) ||
          !(match->clique.reflect_flag = match_chirality (match, edge.node)) ||
          match->reflect_flag
        )
      )
      {
/*
*       STAGE 2: The node is worth adding, if:
*       1. The current node isn't forming a redundant clique:
*          a. We don't care, OR
*          b. It the first in the clique, OR
*          c. The node is numerically above the previously added node, AND
*       2. The current node passes the chemical filter
*          a. The filter is not requested, OR
*          a. The filter indicates a valid node
*       11/96 te
*/
        if
        (
          (
            !match->degeneracy_flag ||
            !match->clique.edge_total ||
            (edge node > match->clique.edge[match->clique.edge_total - 1].node)
          ) &&
          (
            !match->chemical_flag ||
            match->chemical_filter[edge.node]
          )
        )
        {
/*
*         Record the node in the clique and increment the clique size.
*         Also, document the finding of a clique of this size,
*         and erase the record of the next larger one.
*         11/96 te
*/
          match->clique.edge[match->clique.edge_total++] = edge;

          if (match->degeneracy_flag)
          {
            match->degenerate[match->clique.edge_total - 1] = TRUE;
            match->degenerate[match->clique.edge_total] = FALSE;
          }


/*
*         Construct an adjacency list for the new clique
*         11/96 te
*/
          for
          (
            i = j = match->adjacent[match->clique.edge_total] =
              match->clique_adjacency_total[match->clique.edge_total] = 0;
            (i < match->clique_adjacency_total[match->clique.edge_total - 1]) &&
              (j < match->adjacency_total[edge.node]);
          )
          {
            iedge = match->clique_adjacency_list
              [match->clique.edge_total - 1][i];
            jedge = match->adjacency_list[edge.node][j];

            if (jedge.node < iedge.node)
              j++;

            else if (iedge.node < jedge.node)
              i++;

            else
            {
              match->clique_adjacency_list
                [match->clique.edge_total]
                [match->clique_adjacency_total[match->clique.edge_total]++] =
                (iedge.residual > jedge.residual) ? iedge : jedge;

              i++, j++;
            }
          }
        }

/*
*       If this node fails STAGE2 check, then for the sake of degeneracy
*       checking, record that this clique "could" have existed
*       11/96 te
*/
        else if (match->degeneracy_flag)
          match->degenerate[match->clique.edge_total] = TRUE;
      }
    }

/*
*   If we cannot expand the current clique, then process it.
*   10/96 te
*/
    else if (match->clique.edge_total > 0)
    {
/*
*     Compute residual of clique and check its validity
*     1/97 te
*/
      for (i = match->clique.residual = 0; i < match->clique.edge_total; i++)
        match->clique.residual =
          MAX (match->clique.residual, match->clique.edge[i].residual);

/*
*     This clique is sufficient for generating an orientation, if:
*     1. The clique is large enough, AND
*     2. The clique passes the degeneracy filter
*        a. The filter is not requested, OR
*        b. The filter indicates the cliqe is not degenerate
*     11/96 te
*/
      if
      (
        (match->clique.edge_total >= match->clique_size_min) &&
        (!match->degeneracy_flag ||
          !match->degenerate[match->clique.edge_total]) &&
        (match->clique.residual >
          ((match->cycle - 1) * match->distance_tolerance))
      )
        valid_clique = TRUE;

/*
*     Otherwise, erase the last node and decrement the clique size
*     11/96 te
*/
      else
      {
        match->clique.edge_total--;
        match->clique.edge[match->clique.edge_total].node = 0;
        match->clique.edge[match->clique.edge_total].residual = 0;
      }
    }

    else
      return EOF;
  }

/*
* Update match statistics
* 11/96 te
*/
  match->histogram[match->clique.edge_total - 1]++;

/*
```

```
* Reset ligand reflection state
* 1/97 te
*/
  if (match->clique.edge_total < 4)
    match->clique.reflect_flag = FALSE;

  return TRUE;
}


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

Routine to check if the ligand and receptor portions of a clique
have the same or opposite chirality.

Return values:
        TRUE:   The ligand must be reflected
        FALSE:  The ligand must not be reflected
7/95 te

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

int match_chirality (MATCH *match, int node)
{
  int i;
  static int atoms[4], sites[4];

  float calc_chirality (int *, XYZ **);

  for (i = 0; i < 3; i++)
  {
    atoms[i] = match->clique.edge[i].node % match->ligand_center.total.atoms;
    sites[i] = match->clique.edge[i].node / match->ligand_center.total.atoms;
  }

  atoms[3] = node % match->ligand_center.total.atoms;
  sites[3] = node / match->ligand_center.total.atoms;

  if (calc_chirality (atoms, match->ligand_vectors) *
    calc_chirality (sites, match->receptor_vectors) < 0.0)
    return TRUE;

  else
    return FALSE;
}

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

Routine to calculate the whether the fourth member is above or
below the plane defined by the first three members of a group.
7/95 te

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

float calc_chirality (int atoms[4], XYZ **vector)
{
  static XYZ normal;
  void vcrossv3 (XYZ, XYZ, XYZ);
  float vdotv3 (XYZ, XYZ);

  vcrossv3 (vector[atoms[0]][atoms[1]], vector[atoms[0]][atoms[2]], normal);
  return vdotv3 (vector[atoms[0]][atoms[3]], normal);
}



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

Routine to output match routine statistics
7/95 te

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

void output_match_info (MATCH *match)
{
  int i;

/*
* Output histogram of clique formation
* 6/95 te
*/
  if ((global.output_volume != 't') && (match->total > 0))
  {
    fprintf (global.outfile, "\n______Matching_Histogram______\n");

    for (i = 0; i < match->clique_size_max; i++)
      if (match->histogram[i] > 0)
        fprintf (global.outfile, "%7d node%s: %9d\n",
          i + 1, (i == 0 ? " " : "s"),
          match->histogram[i]);

    fprintf (global.outfile, "\n");
  }
}

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

void extract_clique (MATCH *match, MOLECULE *molecule)
{
  int i, lig, rec;

  molecule->transform.refl_flag = match->clique.reflect_flag;

  for (i = 0; i < match->clique.edge_total; i++)
  {
    rec = match->clique.edge[i].node / match->ligand_center.total.atoms;

    copy_atom
      (&match->receptor_clique.atom[i],
      &match->receptor_site.atom[rec]);
    copy_coord
      (match->receptor_clique.coord[i],
      match->receptor_site.coord[rec]);
  }

  for (i = 0; i < match->clique.edge_total; i++)
  {
    lig = match->clique.edge[i].node % match->ligand_center.total.atoms;

    copy_atom
      (&match->ligand_clique.atom[i],
      &match->ligand_center.atom[lig]);
    copy_coord
      (match->ligand_clique.coord[i],
      match->ligand_center.coord[lig]);
  }

  match->ligand_clique.total.atoms =
    match->receptor_clique.total.atoms = match->clique.edge_total;
}


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

Routine to construct a random match
3/97 te

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

int get_random_match
(
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *mol_conf,
  int           molecule_id,
  int           conformation_id,
  int           orientation_id
)
{
  int i, j;
  int unique;

  if (orientation_id == 0)
  {
    if ((molecule_id == 0) && (conformation_id == 0))
    {
      get_site (match, label);
      get_centers (match, label);

      match->clique.edge_max = match->clique_size_max;
      allocate_clique (&match->clique);
      match->clique.edge_total = match->clique_size_max;

      allocate_clique_atoms (match);
    }

    if (!match->centers_flag)
      get_ligand_centers (match, mol_conf);

    if (conformation_id == 0)
      match->node_total =
        match->ligand_center.total.atoms * match->receptor_site.total.atoms;
  }

/*
* Assign nodes to clique
* 3/97 te
*/
  for (i = 0; i < match->clique.edge_total;)
  {
    match->clique.edge[i].node = rand () % match->node_total;

    for (j = 0, unique = TRUE; j < i; j++)
      if ((match->clique.edge[i].node % match->ligand_center.total.atoms ==
        match->clique.edge[j].node % match->ligand_center.total.atoms) ||
        (match->clique.edge[i].node / match->ligand_center.total.atoms ==
        match->clique.edge[j].node / match->ligand_center.total.atoms))
        unique = FALSE;

    if (unique)
      i++;
  }

  return TRUE;
}


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

Routine to free random matching arrays
5/97 te

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

void free_random_matches (MATCH *match)
{
  free_clique (&match->clique);
  free_ligand_centers (match);
  free_molecule (&match->receptor_site);
  free_molecule (&match->ligand_clique);
  free_molecule (&match->receptor_clique);
}


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

Routine to manage chemical screening
3/96 te

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

int check_screen
(
  MATCH         *match,
  LABEL         *label,
  MOLECULE      *molecule,
  int           molecule_id
)
{
  static MOLECULE temporary = {0};

  match->chiral_flag = FALSE;

  if (label->chemical.screen.pharmaco_flag)
  {
    if (molecule_id == 0)
      init_match_site (match, label);

    if (label->chemical.screen.fold_flag)
      return
        check_pharmacophore
        (
          &label->chemical,
          match->distance_tolerance,
          &match->receptor_site,
          molecule
        );

    else
    {
      get_ligand_keys (match, molecule);
      init_match_ligand (match, label);
      initialize_adjacency (match, label);

      if (match_distance (match) != EOF)
        return TRUE;

      else
```

```
          return FALSE;
      }
    }

    else if (label->chemical.screen.similar_flag)
    {
      if (molecule_id == 0)
        init_similar_site (match, label);

      molecule->score.total =
        molecule->score.inter.total =
        check_dissimilarity (&label->chemical, &match->receptor_site, molecule);

      if (molecule->score.total <=
        label->chemical.screen.dissimilar_maximum)
        return TRUE;

      else
        return FALSE;
    }

    else if (label->chemical.screen.fold_flag)
    {
      if (!label->chemical.init_flag)
        get_chemical_labels (&label->chemical);

      fold_keys (&label->chemical, &temporary, molecule);
      copy_keys (molecule, &temporary);
      return TRUE;
    }

    else
      return FALSE;
}


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

Routine to free chemical screening arrays
5/97 te

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

void free_screen
(
  MATCH         *match,
  LABEL         *label
)
{
  if (label->chemical.screen.pharmaco_flag)
  {
    free_match_site (match, label);
/*
    if (label->chemical.screen.fold_flag)
      return
        check_pharmacophore
        (
          &label->chemical,
          match->distance_tolerance,
          &match->receptor_site,
          molecule
        );

    else
    {
      get_ligand_keys (match, molecule);
      init_match_ligand (match, label);
      initialize_adjacency (match, label);

      if (match_distance (match) != EOF)
        return TRUE;

      else
        return FALSE;
    }
*/
  }


  else if (label->chemical.screen.similar_flag)
  {
    free_molecule (&match->receptor_site);
    free_table (&label->chemical, &label->chemical.screen_table);
  }

/*
  else if (label->chemical.screen.fold_flag)
  {
    if (!label->chemical.init_flag)
      get_chemical_labels (&label->chemical);

    fold_keys (&label->chemical, &temporary, molecule);
    copy_keys (molecule, &temporary);
    return TRUE;
  }
*/
}


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

Routine to prepare for similarity search
3/96 te

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

void init_similar_site
(
  MATCH *match,
  LABEL *label
)
{
  FILE *file;

  get_chemical_labels (&label->chemical);

/*
* Read in reference molecule for similarity search
* 11/96 te
*/
  file = efopen (match->receptor_file_name, "r", global.outfile);

  if (read_molecule
    (&match->receptor_site, NULL, match->receptor_file_name, file, FALSE)
    != TRUE)
    exit (fprintf (global.outfile,
      "ERROR init_similar_site: Error reading in reference molecule.\n"));

  fclose (file);
}


################################################################################
#################################### MOL.H #####################################
################################################################################
/*                                                                            */
/*                         Copyright UCSF, 1997                               */
/*                                                                            */

/*
Written by Todd Ewing
5/97
*/

/*
Structures to hold molecule data
5/97 te
*/

typedef struct info_struct
{
  int   input_id;               /* id from input file */
  int   output_id;              /* id for output file */
  char  *name;                  /* name string */
  char  *comment;               /* comment string */
  char  *molecule_type;         /* type string (from sybyl) */
  char  *charge_type;           /* charge string (from sybyl) */
  char  *status_bits;           /* status string (from sybyl) */
  char  *source_file;           /* name of source file (ptr format) */
  long  source_position;        /* file position in source (ptr format) */
  long  file_position;          /* position in input file */

  int   allocated;              /* flag for array allocation */
  int   assign_chem;            /* flag for chemical label assignment */
  int   assign_vdw;             /* flag for vdw label assignment */
  int   assign_flex;            /* flag for flex label assignment */

} INFO;

typedef struct transform_struct
{
  int   flag;                   /* flag for transformation */
  int   trans_flag;             /* flag for rigid body translation */
  int   rot_flag;               /* flag for rigid body rotation */
  int   refl_flag;              /* flag for chiral reflection */
  int   tors_flag;              /* flag for bond rotation */

  XYZ   translate;              /* translation from reference */
  XYZ   rotate;                 /* rotation from reference */
  int   conf_total;             /* Number of conformations */
  int   anchor_segment;         /* Anchor segment */
  int   anchor_atom;            /* Anchor atom */
  int   active_layer;           /* Current active layer */
  int   fold_flag;              /* flag for chemical key folding */

  XYZ   com;                    /* center of mass of current position */
  float rmsd;                   /* RMS deviation from reference */
  int   heavy_total;            /* Number of non-hydrogen atoms */

} TRANSFORM;

typedef struct score_part_struct
{
  int   current_flag;           /* Flag for whether score up-to-date */
  float electro;                /* Electrostatic component of score */
  float vdw;                    /* Van der Waals component of score */
  float total;                  /* Total score */

} SCORE_PART;

typedef struct mol_score_struct
{
  int   type;                   /* type of score method */
  int   bumpcount;              /* number of bumps */
  SCORE_PART    intra;          /* intramolecular score */
  SCORE_PART    inter;          /* intermolecular score */
  float total;                  /* total score */

} MOL_SCORE;

typedef struct arraysize_struct
{
  int   atoms;                  /* Number of atoms */
  int   bonds;                  /* Number of bonds */
  int   substs;                 /* Number of substructures */
  int   torsions;               /* Number of torsions */
  int   sets;                   /* Number of sets */
  int   segments;               /* Number of segments */
  int   layers;                 /* Number of layers */
  int   keys;                   /* Number of keys (along one dimension) */

} ARRAYSIZE;

typedef struct link_struct
{
  int   id;                     /* Link id */
  int   bond_id;                /* Bond id */
  int   out_flag;               /* Flag for whether outward or inward link */

} LINK;

typedef struct atom_struct
{
  int   number;                 /* Atom number */
  int   subst_id;               /* Substructure id */
  int   segment_id;             /* Molecular segment id */
  int   chem_id;                /* Chemical label id */
  int   vdw_id;                 /* VDW id */
  int   heavy_flag;             /* Heavy atom flag */
  int   centrality;             /* Proximity of most distant atom */
  int   flag;                   /* General purpose flag */
  float charge;                 /* Atomic partial charge */
  char  *name;                  /* Atom name */
  char  *type;                  /* Atom type */

  LINK  *neighbor;              /* Atom neighbor list */
  int   neighbor_total;         /* Number of neighbors */
  int   neighbor_max;           /* Maximum number of neighbors */

} ATOM;

typedef struct bond_struct
{
  int   id;                     /* Bond id */
  int   origin;                 /* Atom at bond origin */
  int   target;                 /* Atom at bond target */
```

```
  int    ring_flag;             /* Molecular ring id */
  int    flex_id;               /* Flexible label id */
  char   *type;                 /* Bond type */

} BOND;

typedef struct subst_struct
{
  int    number;                /* Substructure number */
  int    root_atom;             /* First atom in substructure */
  int    dict_type;             /* Dictionary type (sybyl) */
  int    inter_bonds;           /* Number of inter-substructure bonds */
  char   *name;                 /* Substructure name */
  char   *type;                 /* Substructure type */
  char   *chain;                /* Substructure chain identifier */
  char   *sub_type;             /* Sub-type */
  char   *status;               /* Status string (sybyl) */

} SUBST;

typedef struct torsion_struct
{
  int    flex_id;               /* Flexible label id */
  int    bond_id;               /* Bond id */
  int    segment_id;            /* Segment id */
  int    flag;                  /* General purpose flag */
  int    periph_flag;           /* Flag for peripheral torsion (eg. hydroxyl) */
  int    reverse_flag;          /* Flag for reversal of origin/target */
  int    origin;                /* Atom at bond origin */
  int    target;                /* Atom at bond target */
  int    origin_neighbor;       /* Neighbor to atom at bond origin */
  int    target_neighbor;       /* Neighbor to atom at bond origin */
  float  current_angle;         /* Current torsion angle */
  float  target_angle;          /* Target torsion angle */

} TORSION;

typedef struct set_struct
{
  char   *name;                 /* Set name */
  char   *type;                 /* Set type */
  char   *obj_type;             /* Set object type (ATOMS, BONDS, SUBSTS) */
  char   *sub_type;             /* Set subtype (Sybyl) */
  char   *status;               /* Set statis bits (Sybyl) */
  char   *comment;              /* Set comment */
  int    member_total;          /* Number of members in the set */
  int    *member;               /* Set members */

} SET;

typedef struct key_struct
{
  int    count;                 /* Number of distances stored (folded) */
  MASK   distance;              /* Distance keys between labels */

} KEY;

typedef struct segment_struct
{
  int    id;                    /* Segment id */
  int    torsion_id;            /* Torsion in this segment */
  int    layer_id;              /* Layer to which segment belongs */
  int    periph_flag;           /* Flag for peripheral segment */
  int    heavy_total;           /* Number of heavy atoms */
  int    active_flag;           /* Flag for whether to score, etc. */
  int    min_flag;              /* Flag for whether to minimize */
  int    conform_total;         /* Number of conformations in segment */
  int    conform_count;         /* Number of current conformation */
  int    conform_seed;          /* Random seed for conformation search */

  MOL_SCORE      score;         /* Score for segment */

  int    *atom;                 /* Atoms in this segment */
  int    atom_max;              /* Maximum number of atoms in segment */
  int    atom_total;            /* Number of atoms in segment */

  LINK   *neighbor;             /* Neighboring segments */
  int    neighbor_max;          /* Maximum number of neighboring segments */
  int    neighbor_total;        /* Number of neighboring segments */

} SEGMENT;

typedef struct layer_struct
{
  int    id;                    /* Layer id */
  int    active_flag;           /* Flag for whether to score, etc. */
  int    conform_total;         /* Total number of conformations for layer */
  MOL_SCORE      score;         /* Score for layer */

  int    *segment;              /* Segments in this layer */
  int    segment_max;           /* Maximum number of segments in layer */
  int    segment_total;         /* Number of segments in layer */

} LAYER;


typedef struct molecule_struct
{
  INFO           info;          /* General info */
  TRANSFORM      transform;     /* Transformation info */
  MOL_SCORE      score;         /* Molecule score info */
  ARRAYSIZE      total;         /* Current size of arrays */
  ARRAYSIZE      max;           /* Allocated size of arrays */

  ATOM           *atom;         /* Atom info */
  XYZ            *coord;        /* Atom coordinates */
  BOND           *bond;         /* Bond info */
  SUBST          *subst;        /* Sybyl substructure info */
  SET            *set;          /* Sybyl set info */
  TORSION        *torsion;      /* Rotatable torsion info */
  SEGMENT        *segment;      /* Rigid segment info */
  LAYER          *layer;        /* Segment layer info */
  KEY            **key;         /* Chemical keys */

} MOLECULE;

typedef struct linked_molecule
{
  struct linked_molecule *next_head, *next_member;
  MOLECULE *molecule;

} LINKED_MOLECULE;


/*
Routines used to manipulate molecule data structures
12/96 te
*/
```

```
void allocate_molecule               (MOLECULE *);
void allocate_info                   (MOLECULE *);
void allocate_transform              (MOLECULE *);
void allocate_score                  (MOL_SCORE *);
void allocate_atoms                  (MOLECULE *);
void allocate_atom_neighbors         (ATOM     *);
void allocate_bonds                  (MOLECULE *);
void allocate_torsions               (MOLECULE *);
void allocate_substs                 (MOLECULE *);
void allocate_segments               (MOLECULE *);
void allocate_segment_atoms          (SEGMENT  *);
void allocate_segment_neighbors      (SEGMENT  *);
void allocate_layers                 (MOLECULE *);
void allocate_layer_torsions         (LAYER    *);
void allocate_layer_segments         (LAYER    *);
void allocate_sets                   (MOLECULE *);
void allocate_keys                   (MOLECULE *);

void reset_molecule                  (MOLECULE *);
void reset_info                      (MOLECULE *);
void reset_transform                 (MOLECULE *);
void reset_score                     (MOL_SCORE *);
void reset_score_parts               (SCORE_PART *);
void reset_atoms                     (MOLECULE *);
void reset_atom                      (ATOM     *);
void reset_atom_neighbors            (ATOM     *);
void reset_bonds                     (MOLECULE *);
void reset_bond                      (BOND     *);
void reset_torsions                  (MOLECULE *);
void reset_torsion                   (TORSION  *);
void reset_substs                    (MOLECULE *);
void reset_subst                     (SUBST    *);
void reset_segments                  (MOLECULE *);
void reset_segment                   (SEGMENT  *);
void reset_segment_neighbors         (SEGMENT  *);
void reset_layers                    (MOLECULE *);
void reset_layer                     (MOLECULE *, int);
void reset_sets                      (MOLECULE *);
void reset_set                       (SET      *);
void reset_keys                      (MOLECULE *);

void free_molecule                   (MOLECULE *);
void free_info                       (MOLECULE *);
void free_atoms                      (MOLECULE *);
void free_atom                       (ATOM     *);
void free_atom_neighbors             (ATOM     *);
void free_bonds                      (MOLECULE *);
void free_bond                       (BOND     *);
void free_torsions                   (MOLECULE *);
void free_substs                     (MOLECULE *);
void free_subst                      (SUBST    *);
void free_segments                   (MOLECULE *);
void free_segment_atoms              (SEGMENT  *);
void free_segment_neighbors          (SEGMENT  *);
void free_layers                     (MOLECULE *);
void free_layer_segments             (LAYER    *);
void free_sets                       (MOLECULE *);
void free_set                        (SET      *);
void free_keys                       (MOLECULE *);

void reallocate_molecule             (MOLECULE *);
void reallocate_atoms                (MOLECULE *);
void reallocate_atom_neighbors       (ATOM     *);
void reallocate_bonds                (MOLECULE *);
void reallocate_torsions             (MOLECULE *);
void reallocate_substs               (MOLECULE *);
void reallocate_segments             (MOLECULE *);
void reallocate_segment_atoms        (SEGMENT  *);
void reallocate_segment_neighbors    (SEGMENT  *);
void reallocate_layers               (MOLECULE *);
void reallocate_layer_segments       (LAYER    *);
void reallocate_sets                 (MOLECULE *);
void reallocate_keys                 (MOLECULE *);

void copy_molecule                   (MOLECULE *, MOLECULE *);
void copy_info                       (MOLECULE *, MOLECULE *);
void copy_transform                  (MOLECULE *, MOLECULE *);
void copy_score                      (MOL_SCORE *, MOL_SCORE *);
void copy_atoms                      (MOLECULE *, MOLECULE *);
void copy_atom                       (ATOM     *, ATOM     *);
void copy_atom_neighbors             (ATOM     *, ATOM     *);
void copy_coords                     (MOLECULE *, MOLECULE *);
void copy_coord                      (XYZ       , XYZ       );
void copy_bonds                      (MOLECULE *, MOLECULE *);
void copy_bond                       (BOND     *, BOND     *);
void copy_torsions                   (MOLECULE *, MOLECULE *);
void copy_torsion                    (TORSION  *, TORSION  *);
void copy_substs                     (MOLECULE *, MOLECULE *);
void copy_subst                      (SUBST    *, SUBST    *);
void copy_segments                   (MOLECULE *, MOLECULE *);
void copy_segment                    (SEGMENT  *, SEGMENT  *);
void copy_layers                     (MOLECULE *, MOLECULE *);
void copy_layer                      (LAYER    *, LAYER    *);
void copy_sets                       (MOLECULE *, MOLECULE *);
void copy_set                        (SET      *, SET      *);
void copy_keys                       (MOLECULE *, MOLECULE *);

void save_molecule                   (MOLECULE *, FILE *);
void save_info                       (MOLECULE *, FILE *);
void save_transform                  (MOLECULE *, FILE *);
void save_score                      (MOL_SCORE *, FILE *);
void save_atoms                      (MOLECULE *, FILE *);
void save_atom                       (ATOM     *, FILE *);
void save_atom_neighbors             (ATOM     *, FILE *);
void save_bonds                      (MOLECULE *, FILE *);
void save_bond                       (BOND     *, FILE *);
void save_torsions                   (MOLECULE *, FILE *);
void save_substs                     (MOLECULE *, FILE *);
void save_subst                      (SUBST    *, FILE *);
void save_segments                   (MOLECULE *, FILE *);
void save_segment                    (SEGMENT  *, FILE *);
void save_layers                     (MOLECULE *, FILE *);
void save_layer                      (LAYER    *, FILE *);
void save_sets                       (MOLECULE *, FILE *);
void save_set                        (SET      *, FILE *);
void save_keys                       (MOLECULE *, FILE *);

void load_molecule                   (MOLECULE *, FILE *);
void load_info                       (MOLECULE *, FILE *);
void load_transform                  (MOLECULE *, FILE *);
void load_score                      (MOL_SCORE *, FILE *);
void load_atoms                      (MOLECULE *, FILE *);
void load_atom                       (ATOM     *, FILE *);
void load_atom_neighbors             (ATOM     *, FILE *);
void load_bonds                      (MOLECULE *, FILE *);
void load_bond                       (BOND     *, FILE *);
void load_torsions                   (MOLECULE *, FILE *);
void load_substs                     (MOLECULE *, FILE *);
void load_subst                      (SUBST    *, FILE *);
```

```c
void load_segments                  (MOLECULE *, FILE *);
void load_segment                   (SEGMENT  *, FILE *);
void load_layers                    (MOLECULE *, FILE *);
void load_layer                     (LAYER    *, FILE *);
void load_sets                      (MOLECULE *, FILE *);
void load_set                       (SET      *, FILE *);
void load_keys                      (MOLECULE *, FILE *);

void save_string                    (char **, FILE *);
void load_string                    (char **, FILE *);

int get_identifier                  (void);

void atom_neighbors                 (MOLECULE *);
void revise_atom_neighbors          (MOLECULE *);

void get_torsion_neighbors          (MOLECULE *);
void reverse_torsion                (TORSION  *);
float calculate_distances           (MOLECULE *, float ***, int *);
void free_distances                 (float ***, int *);

int get_atom_neighbor
(
  void *atom,
  int atom_id,
  int neighbor_id
);

void flag_atom_neighbor
(
  void  *atom_in,
  int   atom_id,
  int   neighbor,
  int   flag
);

void flag_segment_neighbor
(
  void  *segment_in,
  int   segment_id,
  int   neighbor,
  int   flag
);

int get_segment_neighbor
(
  void *segment,
  int segment_id,
  int neighbor_id
);

int get_atom_centrality
(
  MOLECULE *molecule,
  int atom_id
);

##############################################################################
##################################### MOL.C ##################################
##############################################################################
/*                                                                         */
/*                        Copyright UCSF, 1997                             */
/*                                                                         */

/*
Written by Todd Ewing
10/95
*/
#include <stdio.h>
#include "define.h"
#include "utility.h"
#include "global.h"
#include "search.h"
#include "mol.h"
#include "vector.h"

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

void allocate_molecule (MOLECULE *molecule)
{
  allocate_info           (molecule);
  allocate_transform      (molecule);
  allocate_score          (&molecule->score);
  allocate_atoms          (molecule);
  allocate_bonds          (molecule);
  allocate_torsions       (molecule);
  allocate_substs         (molecule);
  allocate_segments       (molecule);
  allocate_layers         (molecule);
  allocate_sets           (molecule);
  allocate_keys           (molecule);
}

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

void reset_molecule (MOLECULE *molecule)
{
  reset_info              (molecule);
  reset_transform         (molecule);
  reset_score             (&molecule->score);
  reset_atoms             (molecule);
  reset_bonds             (molecule);
  reset_torsions          (molecule);
  reset_substs            (molecule);
  reset_segments          (molecule);
  reset_layers            (molecule);
  reset_sets              (molecule);
  reset_keys              (molecule);
}

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

void free_molecule (MOLECULE *molecule)
{
  free_info               (molecule);
  free_atoms              (molecule);
  free_bonds              (molecule);
  free_torsions           (molecule);
  free_substs             (molecule);
  free_segments           (molecule);
  free_layers             (molecule);
  free_sets               (molecule);
  free_keys               (molecule);
}

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

void reallocate_molecule (MOLECULE *molecule)
{
  reallocate_atoms        (molecule);
  reallocate_bonds        (molecule);
  reallocate_torsions     (molecule);
  reallocate_substs       (molecule);
  reallocate_segments     (molecule);
  reallocate_layers       (molecule);
  reallocate_sets         (molecule);
  reallocate_keys         (molecule);
}

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

void copy_molecule (MOLECULE *copy, MOLECULE *original)
{
  copy_info               (copy, original);
  copy_transform          (copy, original);
  copy_score              (&copy->score, &original->score);
  copy_atoms              (copy, original);
  copy_bonds              (copy, original);
  copy_torsions           (copy, original);
  copy_substs             (copy, original);
  copy_segments           (copy, original);
  copy_layers             (copy, original);
  copy_sets               (copy, original);
  copy_keys               (copy, original);
}

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

void save_molecule (MOLECULE *molecule, FILE *file)
{
  efwrite (&molecule->max, sizeof (ARRAYSIZE), 1, file);
  efwrite (&molecule->total, sizeof (ARRAYSIZE), 1, file);

  save_info               (molecule, file);
  save_transform          (molecule, file);
  save_score              (&molecule->score, file);
  save_atoms              (molecule, file);
  save_bonds              (molecule, file);
  save_torsions           (molecule, file);
  save_substs             (molecule, file);
  save_segments           (molecule, file);
  save_layers             (molecule, file);
  save_sets               (molecule, file);
  save_keys               (molecule, file);
}

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

void load_molecule (MOLECULE *molecule, FILE *file)
{
  free_molecule (molecule);

  efread (&molecule->max, sizeof (ARRAYSIZE), 1, file);
  efread (&molecule->total, sizeof (ARRAYSIZE), 1, file);

  allocate_molecule       (molecule);

  load_info               (molecule, file);
  load_transform          (molecule, file);
  load_score              (&molecule->score, file);
  load_atoms              (molecule, file);
  load_bonds              (molecule, file);
  load_torsions           (molecule, file);
  load_substs             (molecule, file);
  load_segments           (molecule, file);
  load_layers             (molecule, file);
  load_sets               (molecule, file);
  load_keys               (molecule, file);
}

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

int get_identifier (void)
{
  static int identifier = 0;
  return ++identifier;
}

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

void allocate_info (MOLECULE *molecule)
{
  reset_info (molecule);
}

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

void reset_info (MOLECULE *molecule)
{
  efree ((void **) &molecule->info.name);
  efree ((void **) &molecule->info.comment);
  efree ((void **) &molecule->info.molecule_type);
  efree ((void **) &molecule->info.charge_type);
  efree ((void **) &molecule->info.status_bits);
  efree ((void **) &molecule->info.source_file);

  memset (&molecule->info, 0, sizeof (INFO));
  molecule->info.input_id = NEITHER;
}

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

void free_info (MOLECULE *molecule)
{
  efree ((void **) &molecule->info.name);
  efree ((void **) &molecule->info.comment);
  efree ((void **) &molecule->info.molecule_type);
  efree ((void **) &molecule->info.charge_type);
  efree ((void **) &molecule->info.status_bits);
  efree ((void **) &molecule->info.source_file);

  memset (&molecule->info, 0, sizeof (INFO));
  molecule->info.input_id = NEITHER;
}

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

void copy_info (MOLECULE *copy, MOLECULE *original)
{
  copy->info.input_id = original->info.input_id;
  copy->info.output_id = original->info.output_id;

  vstrcpy (&copy->info.name, original->info.name);
  vstrcpy (&copy->info.comment, original->info.comment);
  vstrcpy (&copy->info.molecule_type, original->info.molecule_type);
  vstrcpy (&copy->info.charge_type, original->info.charge_type);
  vstrcpy (&copy->info.status_bits, original->info.status_bits);
```

```c
  vstrcpy (&copy->info.source_file, original->info.source_file);

  copy->info.source_position = original->info.source_position;
  copy->info.file_position = original->info.file_position;

  copy->info.allocated = original->info.allocated;
  copy->info.assign_chem = original->info.assign_chem;
  copy->info.assign_flex = original->info.assign_flex;
  copy->info.assign_vdw = original->info.assign_vdw;
}

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

void save_info (MOLECULE *molecule, FILE *file)
{
  efwrite (&molecule->info.input_id, sizeof (int), 1, file);
  efwrite (&molecule->info.output_id, sizeof (int), 1, file);

  save_string (&molecule->info.name, file);
  save_string (&molecule->info.comment, file);
  save_string (&molecule->info.molecule_type, file);
  save_string (&molecule->info.charge_type, file);
  save_string (&molecule->info.status_bits, file);
  save_string (&molecule->info.source_file, file);

  efwrite (&molecule->info.source_position, sizeof (long), 1, file);
  efwrite (&molecule->info.file_position, sizeof (long), 1, file);

  efwrite (&molecule->info.allocated, sizeof (int), 1, file);
  efwrite (&molecule->info.assign_chem, sizeof (int), 1, file);
  efwrite (&molecule->info.assign_vdw, sizeof (int), 1, file);
  efwrite (&molecule->info.assign_flex, sizeof (int), 1, file);
}

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

void load_info (MOLECULE *molecule, FILE *file)
{
  efread (&molecule->info.input_id, sizeof (int), 1, file);
  efread (&molecule->info.output_id, sizeof (int), 1, file);

  load_string (&molecule->info.name, file);
  load_string (&molecule->info.comment, file);
  load_string (&molecule->info.molecule_type, file);
  load_string (&molecule->info.charge_type, file);
  load_string (&molecule->info.status_bits, file);
  load_string (&molecule->info.source_file, file);

  efread (&molecule->info.source_position, sizeof (long), 1, file);
  efread (&molecule->info.file_position, sizeof (long), 1, file);

  efread (&molecule->info.allocated, sizeof (int), 1, file);
  efread (&molecule->info.assign_chem, sizeof (int), 1, file);
  efread (&molecule->info.assign_vdw, sizeof (int), 1, file);
  efread (&molecule->info.assign_flex, sizeof (int), 1, file);
}

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

void allocate_transform (MOLECULE *molecule)
{
  reset_transform (molecule);
}

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

void reset_transform (MOLECULE *molecule)
{
  memset (&molecule->transform, 0, sizeof (TRANSFORM));
  molecule->transform.refl_flag = NEITHER;
  molecule->transform.tors_flag = NEITHER;
  molecule->transform.anchor_segment = NEITHER;
  molecule->transform.anchor_atom = NEITHER;
}

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

void copy_transform (MOLECULE *copy, MOLECULE *original)
{
  copy->transform.flag = original->transform.flag;
  copy->transform.trans_flag = original->transform.trans_flag;
  copy->transform.rot_flag = original->transform.rot_flag;
  copy->transform.refl_flag = original->transform.refl_flag;
  copy->transform.tors_flag = original->transform.tors_flag;

  copy->transform.translate[0] = original->transform.translate[0];
  copy->transform.translate[1] = original->transform.translate[1];
  copy->transform.translate[2] = original->transform.translate[2];

  copy->transform.rotate[0] = original->transform.rotate[0];
  copy->transform.rotate[1] = original->transform.rotate[1];
  copy->transform.rotate[2] = original->transform.rotate[2];

  copy->transform.conf_total = original->transform.conf_total;
  copy->transform.anchor_segment = original->transform.anchor_segment;
  copy->transform.anchor_atom = original->transform.anchor_atom;
  copy->transform.active_layer = original->transform.active_layer;
  copy->transform.fold_flag = original->transform.fold_flag;

  copy->transform.com[0] = original->transform.com[0];
  copy->transform.com[1] = original->transform.com[1];
  copy->transform.com[2] = original->transform.com[2];

  copy->transform.rmsd = original->transform.rmsd;
  copy->transform.heavy_total = original->transform.heavy_total;
}

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

void save_transform (MOLECULE *molecule, FILE *file)
{
  efwrite (&molecule->transform, sizeof (TRANSFORM), 1, file);
}

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

void load_transform (MOLECULE *molecule, FILE *file)
{
  efread (&molecule->transform, sizeof (TRANSFORM), 1, file);
}

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

void allocate_score (MOL_SCORE *score)
{
  reset_score (score);
}

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

void reset_score (MOL_SCORE *score)
{
  score->type = 0;
  score->bumpcount = 0;

  reset_score_parts (&score->intra);
  reset_score_parts (&score->inter);

  score->inter.total = INITIAL_SCORE;
  score->total = INITIAL_SCORE;
}

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

void reset_score_parts (SCORE_PART *part)
{
  part->current_flag = FALSE;
  part->total = 0;
  part->electro = 0;
  part->vdw = 0;
}

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

void copy_score (MOL_SCORE *copy, MOL_SCORE *original)
{
  copy->type = original->type;
  copy->bumpcount = original->bumpcount;
  copy->intra = original->intra;
  copy->inter = original->inter;
  copy->total = original->total;
}

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

void add_score_parts (SCORE_PART *sum, SCORE_PART *increment)
{
  sum->vdw += increment->vdw;
  sum->electro += increment->electro;
  sum->total += increment->total;
}

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

void save_score (MOL_SCORE *score, FILE *file)
{
  efwrite (score, sizeof (MOL_SCORE), 1, file);
}

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

void load_score (MOL_SCORE *score, FILE *file)
{
  efread (score, sizeof (MOL_SCORE), 1, file);
}

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

void allocate_atoms (MOLECULE *molecule)
{
  int i;

  if (molecule->max.atoms > 0)
  {
    ecalloc
    (
      (void **) &molecule->atom,
      molecule->max.atoms,
      sizeof (ATOM),
      molecule->info.name,
      global.outfile
    );

    ecalloc
    (
      (void **) &molecule->coord,
      molecule->max.atoms,
      sizeof (XYZ),
      molecule->info.name,
      global.outfile
    );

    for (i = 0; i < molecule->total.atoms; i++)
      reset_atom (&molecule->atom[i]);
  }
}

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

void allocate_atom_neighbors (ATOM *atom)
{
  if (atom->neighbor_max > 0)
  {
    ecalloc
    (
      (void **) &atom->neighbor,
      atom->neighbor_max,
      sizeof (LINK),
      "atom neighbors",
      global.outfile
    );

    reset_atom_neighbors (atom);
  }
}

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

void reset_atoms (MOLECULE *molecule)
{
  int i;

  if (molecule->max.atoms > 0)
  {
    for (i = 0; i < molecule->max.atoms; i++)
      reset_atom (&molecule->atom[i]);

    memset (molecule->coord, 0, molecule->max.atoms * sizeof (XYZ));
  }

  molecule->total.atoms = 0;
}

/* ////////////////////////////////////////////////////////////// */
```

```
void reset_atom (ATOM *atom)
{
  efree ((void **) &atom->name);
  efree ((void **) &atom->type);

  atom->number = 0;
  atom->subst_id = 0;
  atom->chem_id = 0;
  atom->vdw_id = 0;
  atom->heavy_flag = FALSE;
  atom->centrality = NEITHER;
  atom->segment_id = NEITHER;
  atom->flag = 0;
  atom->charge = 0;

  reset_atom_neighbors (atom);
}

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

void reset_atom_neighbors (ATOM *atom)
{
  int i;

  for (i = 0; i < atom->neighbor_max; i++)
  {
    atom->neighbor[i].id = NEITHER;
    atom->neighbor[i].bond_id = NEITHER;
    atom->neighbor[i].out_flag = FALSE;
  }

  atom->neighbor_total = 0;
}

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

void free_atoms (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.atoms; i++)
    free_atom (&molecule->atom[i]);

  efree ((void **) &molecule->atom);
  efree ((void **) &molecule->coord);
  molecule->max.atoms = 0;
}

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

void free_atom (ATOM *atom)
{
  efree ((void **) &atom->name);
  efree ((void **) &atom->type);

  free_atom_neighbors (atom);
}

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

void free_atom_neighbors (ATOM *atom)
{
  efree ((void **) &atom->neighbor);
  atom->neighbor_max = 0;
}

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

void reallocate_atoms (MOLECULE *molecule)
{
  if (molecule->total.atoms > molecule->max.atoms)
  {
    free_atoms (molecule);
    molecule->max.atoms = molecule->total.atoms;
    allocate_atoms (molecule);
  }
}

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

void reallocate_atom_neighbors (ATOM *atom)
{
  if (atom->neighbor_total > atom->neighbor_max)
  {
    free_atom_neighbors (atom);
    atom->neighbor_max = atom->neighbor_total;
    allocate_atom_neighbors (atom);
  }
}

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

void copy_atoms (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total.atoms = original->total.atoms;
  reallocate_atoms (copy);

  for (i = 0; i < original->total.atoms; i++)
  {
    copy_atom (&copy->atom[i], &original->atom[i]);
    copy_coord (copy->coord[i], original->coord[i]);
  }
}

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

void copy_atom (ATOM *copy, ATOM *original)
{
  copy->number = original->number;
  copy->subst_id = original->subst_id;
  copy->chem_id = original->chem_id;
  copy->vdw_id = original->vdw_id;
  copy->heavy_flag = original->heavy_flag;
  copy->centrality = original->centrality;
  copy->segment_id = original->segment_id;
  copy->flag = original->flag;
  copy->charge = original->charge;

  vstrcpy (&copy->name, original->name);
  vstrcpy (&copy->type, original->type);

  copy_atom_neighbors (copy, original);
}

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

void copy_atom_neighbors (ATOM *copy, ATOM *original)
{
  copy->neighbor_total = original->neighbor_total;
  reallocate_atom_neighbors (copy);

  memcpy
  (copy->neighbor, original->neighbor,
  original->neighbor_total * sizeof (LINK));

  copy->neighbor_total = original->neighbor_total;
}

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

void copy_coords (MOLECULE *copy, MOLECULE *original)
{
  int i;

  if (copy->info.input_id != original->info.input_id)
    exit (fprintf (global.outfile,
      "ERROR copy_coords: "
      "Error copying coordinates to out-of-date molecule structure.\n"));

  for (i = 0; i < original->total.atoms; i++)
    copy_coord (copy->coord[i], original->coord[i]);
}

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

void copy_coord (XYZ copy, XYZ original)
{
  copy[0] = original[0];
  copy[1] = original[1];
  copy[2] = original[2];
}

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

void save_atoms (MOLECULE *molecule, FILE *file)
{
  int i;

  if (molecule->max.atoms > 0)
  {
    for (i = 0; i < molecule->max.atoms; i++)
      save_atom (&molecule->atom[i], file);

    efwrite
    (
      molecule->coord,
      sizeof (XYZ),
      molecule->max.atoms,
      file
    );
  }
}

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

void save_atom (ATOM *atom, FILE *file)
{
  efwrite (atom, sizeof (ATOM), 1, file);

  save_string (&atom->name, file);
  save_string (&atom->type, file);

  save_atom_neighbors (atom, file);
}

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

void save_atom_neighbors (ATOM *atom, FILE *file)
{
  efwrite (&atom->neighbor_total, sizeof (int), 1, file);
  efwrite (&atom->neighbor_max, sizeof (int), 1, file);
  efwrite (atom->neighbor, sizeof (LINK), atom->neighbor_total, file);
}

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

void load_atoms (MOLECULE *molecule, FILE *file)
{
  int i;

  if (molecule->max.atoms > 0)
  {
    for (i = 0; i < molecule->max.atoms; i++)
      load_atom (&molecule->atom[i], file);

    efread
    (
      molecule->coord,
      sizeof (XYZ),
      molecule->max.atoms,
      file
    );
  }
}

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

void load_atom (ATOM *atom, FILE *file)
{
  efread (atom, sizeof (ATOM), 1, file);

  atom->name = NULL;
  atom->type = NULL;
  atom->neighbor = NULL;

  load_string (&atom->name, file);
  load_string (&atom->type, file);

  load_atom_neighbors (atom, file);
}

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

void load_atom_neighbors (ATOM *atom, FILE *file)
{
  efread (&atom->neighbor_total, sizeof (int), 1, file);
  efread (&atom->neighbor_max, sizeof (int), 1, file);

  allocate_atom_neighbors (atom);

  efread (atom->neighbor, sizeof (LINK), atom->neighbor_total, file);
}

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

int get_atom_neighbor
```

```
{
  void          *atom,
  int           atom_id,
  int           neighbor_id
)
{
  if (neighbor_id < ((ATOM *) atom)[atom_id].neighbor_total)
    return ((ATOM *) atom)[atom_id].neighbor[neighbor_id].id;

  else
    return EOF;
}

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

void allocate_bonds (MOLECULE *molecule)
{
  int i;

  if (molecule->max.bonds > 0)
    ecalloc
    (
      (void **) &molecule->bond,
      molecule->max.bonds,
      sizeof (BOND),
      molecule->info.name,
      global.outfile
    );

  for (i = 0; i < molecule->max.bonds; i++)
    reset_bond (&molecule->bond[i]);
}

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

void reset_bonds (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.bonds; i++)
    reset_bond (&molecule->bond[i]);

  molecule->total.bonds = 0;
}

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

void reset_bond (BOND *bond)
{
  bond->id = 0;
  bond->origin = NEITHER;
  bond->target = NEITHER;
  bond->ring_flag = 0;
  bond->flex_id = 0;
  efree ((void **) &bond->type);
}

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

void free_bonds (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.bonds; i++)
    free_bond (&molecule->bond[i]);

  efree ((void **) &molecule->bond);
  molecule->max.bonds = 0;
}

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

void free_bond (BOND *bond)
{
  efree ((void **) &bond->type);
}

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

void reallocate_bonds (MOLECULE *molecule)
{
  if (molecule->total.bonds > molecule->max.bonds)
  {
    free_bonds (molecule);
    molecule->max.bonds = molecule->total.bonds;
    allocate_bonds (molecule);
  }
}

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

void copy_bonds (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total.bonds = original->total.bonds;
  reallocate_bonds (copy);

  for (i = 0; i < original->total.bonds; i++)
    copy_bond (&copy->bond[i], &original->bond[i]);

  copy->total.bonds = original->total.bonds;
}

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

void copy_bond (BOND *copy, BOND *original)
{
  copy->id = original->id;
  copy->origin = original->origin;
  copy->target = original->target;
  copy->ring_flag = original->ring_flag;
  copy->flex_id = original->flex_id;
  vstrcpy (&copy->type, original->type);
}

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

void save_bonds (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.bonds; i++)
    save_bond (&molecule->bond[i], file);
}

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

void save_bond (BOND *bond, FILE *file)
{
  efwrite (bond, sizeof (BOND), 1, file);
  save_string (&bond->type, file);
}

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

void load_bonds (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.bonds; i++)
    load_bond (&molecule->bond[i], file);
}

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

void load_bond (BOND *bond, FILE *file)
{
  efread (bond, sizeof (BOND), 1, file);
  bond->type = NULL;
  load_string (&bond->type, file);
}

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

void allocate_torsions (MOLECULE *molecule)
{
  int i;

  if (molecule->max.torsions > 0)
    ecalloc
    (
      (void **) &molecule->torsion,
      molecule->max.torsions,
      sizeof (TORSION),
      molecule->info.name,
      global.outfile
    );

  for (i = 0; i < molecule->max.torsions; i++)
    reset_torsion (&molecule->torsion[i]);
}

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

void reset_torsions (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.torsions; i++)
    reset_torsion (&molecule->torsion[i]);

  molecule->total.torsions = 0;
}

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

void reset_torsion (TORSION *torsion)
{
  torsion->flex_id = 0;
  torsion->bond_id = NEITHER;
  torsion->segment_id = NEITHER;
  torsion->periph_flag = 0;
  torsion->reverse_flag = 0;
  torsion->origin = NEITHER;
  torsion->target = NEITHER;
  torsion->origin_neighbor = NEITHER;
  torsion->target_neighbor = NEITHER;
  torsion->current_angle = 0;
  torsion->target_angle = 0;
}

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

void free_torsions (MOLECULE *molecule)
{
  efree ((void **) &molecule->torsion);
  molecule->max.torsions = 0;
}

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

void reallocate_torsions (MOLECULE *molecule)
{
  if (molecule->total.torsions > molecule->max.torsions)
  {
    free_torsions (molecule);
    molecule->max.torsions = molecule->total.torsions;
    allocate_torsions (molecule);
  }
}

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

void copy_torsions (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total.torsions = original->total.torsions;
  reallocate_torsions (copy);

  for (i = 0; i < original->total.torsions; i++)
    copy_torsion (&copy->torsion[i], &original->torsion[i]);

  copy->total.torsions = original->total.torsions;
}

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

void copy_torsion (TORSION *copy, TORSION *original)
{
  copy->flex_id = original->flex_id;
  copy->bond_id = original->bond_id;
  copy->segment_id = original->segment_id;
  copy->periph_flag = original->periph_flag;
  copy->reverse_flag = original->reverse_flag;
  copy->origin = original->origin;
  copy->target = original->target;
  copy->origin_neighbor = original->origin_neighbor;
  copy->target_neighbor = original->target_neighbor;
  copy->current_angle = original->current_angle;
  copy->target_angle = original->target_angle;
}

/* //////////////////////////////////////////////////////////////// */
```

```
void save_torsions (MOLECULE *molecule, FILE *file)
{
  if (molecule->max.torsions > 0)
    efwrite (molecule->torsion, sizeof (TORSION), molecule->max.torsions, file);
}

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

void load_torsions (MOLECULE *molecule, FILE *file)
{
  if (molecule->max.torsions > 0)
    efread (molecule->torsion, sizeof (TORSION), molecule->max.torsions, file);
}

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

void allocate_substs (MOLECULE *molecule)
{
  int i;

  if (molecule->max.substs > 0)
    ecalloc
    (
      (void **) &molecule->subst,
      molecule->max.substs,
      sizeof (SUBST),
      molecule->info.name,
      global.outfile
    );

  for (i = 0; i < molecule->max.substs; i++)
    reset_subst (&molecule->subst[i]);
}

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

void reset_substs (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.substs; i++)
    reset_subst (&molecule->subst[i]);

  molecule->total.substs = 0;
}

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

void reset_subst (SUBST *subst)
{
  efree ((void **) &subst->name);
  efree ((void **) &subst->type);
  efree ((void **) &subst->chain);
  efree ((void **) &subst->sub_type);
  efree ((void **) &subst->status);

  memset (subst, 0, sizeof (SUBST));
}

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

void free_substs (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.substs; i++)
    free_subst (&molecule->subst[i]);

  efree ((void **) &molecule->subst);
  molecule->max.substs = 0;
}

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

void free_subst (SUBST *subst)
{
  efree ((void **) &subst->name);
  efree ((void **) &subst->type);
  efree ((void **) &subst->chain);
  efree ((void **) &subst->sub_type);
  efree ((void **) &subst->status);
}

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

void reallocate_substs (MOLECULE *molecule)
{
  if (molecule->total.substs > molecule->max.substs)
  {
    free_substs (molecule);
    molecule->max.substs = molecule->total.substs;
    allocate_substs (molecule);
  }
}

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

void copy_substs (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total.substs = original->total.substs;
  reallocate_substs (copy);

  for (i = 0; i < original->total.substs; i++)
    copy_subst (&copy->subst[i], &original->subst[i]);

  copy->total.substs = original->total.substs;
}

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

void copy_subst (SUBST *copy, SUBST *original)
{
  copy->number = original->number;
  copy->root_atom = original->root_atom;
  copy->dict_type = original->dict_type;
  copy->inter_bonds = original->inter_bonds;

  vstrcpy (&copy->name, original->name);
  vstrcpy (&copy->type, original->type);
  vstrcpy (&copy->chain, original->chain);
  vstrcpy (&copy->sub_type, original->sub_type);
  vstrcpy (&copy->status, original->status);
}

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

void save_substs (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.substs; i++)
    save_subst (&molecule->subst[i], file);
}

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

void save_subst (SUBST *subst, FILE *file)
{
  efwrite (subst, sizeof (SUBST), 1, file);

  save_string (&subst->name, file);
  save_string (&subst->type, file);
  save_string (&subst->chain, file);
  save_string (&subst->sub_type, file);
  save_string (&subst->status, file);
}

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

void load_substs (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.substs; i++)
    load_subst (&molecule->subst[i], file);
}

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

void load_subst (SUBST *subst, FILE *file)
{
  efread (subst, sizeof (SUBST), 1, file);

  subst->name = NULL;
  subst->type = NULL;
  subst->chain = NULL;
  subst->sub_type = NULL;
  subst->status = NULL;

  load_string (&subst->name, file);
  load_string (&subst->type, file);
  load_string (&subst->chain, file);
  load_string (&subst->sub_type, file);
  load_string (&subst->status, file);
}

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

void allocate_segments (MOLECULE *molecule)
{
  int i, j;

  ecalloc
  (
    (void **) &molecule->segment,
    molecule->max.segments,
    sizeof (SEGMENT),
    molecule->info.name,
    global.outfile
  );

  for (i = 0; i < molecule->max.segments; i++)
    reset_segment (&molecule->segment[i]);
}

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

void allocate_segment_atoms (SEGMENT *segment)
{
  if (segment->atom_max > 0)
  {
    ecalloc
    (
      (void **) &segment->atom,
      segment->atom_max,
      sizeof (int),
      "segment atoms",
      global.outfile
    );
  }
}

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

void allocate_segment_neighbors (SEGMENT *segment)
{
  int i;

  if (segment->neighbor_max > 0)
  {
    ecalloc
    (
      (void **) &segment->neighbor,
      segment->neighbor_max,
      sizeof (LINK),
      "segment neighbors",
      global.outfile
    );

    for (i = 0; i < segment->neighbor_max; i++)
    {
      segment->neighbor[i].id = NEITHER;
      segment->neighbor[i].bond_id = NEITHER;
      segment->neighbor[i].out_flag = FALSE;
    }
  }
}

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

void reset_segments (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.segments; i++)
    reset_segment (&molecule->segment[i]);

  molecule->total.segments = 0;
}

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

void reset_segment (SEGMENT *segment)
{
```

```
  segment->id = NEITHER;
  segment->torsion_id = NEITHER;
  segment->layer_id = NEITHER;
  segment->periph_flag = NEITHER;
  segment->heavy_total = 0;
  segment->active_flag = FALSE;
  segment->min_flag = FALSE;
  segment->conform_count = 0;
  segment->conform_total = 1;
  reset_score (&segment->score);

  memset (segment->atom, 0, segment->atom_max * sizeof (int));
  segment->atom_total = 0;

  reset_segment_neighbors (segment);
}

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

void reset_segment_neighbors (SEGMENT *segment)
{
  int i;

  for (i = 0; i < segment->neighbor_max; i++)
  {
    segment->neighbor[i].id = NEITHER;
    segment->neighbor[i].bond_id = NEITHER;
    segment->neighbor[i].out_flag = FALSE;
  }

  segment->neighbor_total = 0;
}

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

void free_segments (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.segments; i++)
  {
    free_segment_atoms (&molecule->segment[i]);
    free_segment_neighbors (&molecule->segment[i]);
  }

  efree ((void **) &molecule->segment);
  molecule->max.segments = 0;
}

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

void free_segment_atoms (SEGMENT *segment)
{
  efree ((void **) &segment->atom);
  segment->atom_max = 0;
}

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

void free_segment_neighbors (SEGMENT *segment)
{
  efree ((void **) &segment->neighbor);
  segment->neighbor_max = 0;
}

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

void reallocate_segments (MOLECULE *molecule)
{
  if (molecule->total.segments > molecule->max.segments)
  {
    free_segments (molecule);
    molecule->max.segments = molecule->total.segments;
    allocate_segments (molecule);
  }
}

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

void reallocate_segment_atoms (SEGMENT *segment)
{
  if (segment->atom_total > segment->atom_max)
  {
    free_segment_atoms (segment);
    segment->atom_max = segment->atom_total;
    allocate_segment_atoms (segment);
  }
}

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

void reallocate_segment_neighbors (SEGMENT *segment)
{
  if (segment->neighbor_total > segment->neighbor_max)
  {
    free_segment_neighbors (segment);
    segment->neighbor_max = segment->neighbor_total;
    allocate_segment_neighbors (segment);
  }
}

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

void copy_segments (MOLECULE *copy, MOLECULE *original)
{
  int i, j;
  int atom;

  copy->total.segments = original->total.segments;
  reallocate_segments (copy);

  for (i = 0; i < original->total.segments; i++)
  {
    copy_segment (&copy->segment[i], &original->segment[i]);

    for (j = 0; j < copy->segment[i].atom_total; j++)
    {
      atom = copy->segment[i].atom[j];

      if ((atom >= 0) && (atom < copy->total.atoms))
        copy->atom[atom].segment_id = i;

      else
        exit (fprintf (global.outfile,
          "ERROR copy_segments: unable to copy atoms\n"));
    }
  }

  if ((copy->total.atoms > 0) &&
    (copy->total.segments > 1) &&
    (copy->transform.anchor_atom != NEITHER))
    revise_atom_neighbors (copy);
/*
*/
}

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

void copy_segment (SEGMENT *copy, SEGMENT *original)
{
  copy->id = original->id;
  copy->torsion_id = original->torsion_id;
  copy->layer_id = original->layer_id;
  copy->periph_flag = original->periph_flag;
  copy->heavy_total = original->heavy_total;
  copy->active_flag = original->active_flag;
  copy->min_flag = original->min_flag;
  copy->conform_count = original->conform_count;
  copy->conform_total = original->conform_total;
  copy_score (&copy->score, &original->score);

  copy->atom_total = original->atom_total;
  reallocate_segment_atoms (copy);
  memcpy
    (copy->atom, original->atom,
    original->atom_total * sizeof (int));

  copy->neighbor_total = original->neighbor_total;
  reallocate_segment_neighbors (copy);
  memcpy
    (copy->neighbor, original->neighbor,
    original->neighbor_total * sizeof (LINK));
}

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

void save_segments (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.segments; i++)
    save_segment (&molecule->segment[i], file);
}

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

void save_segment (SEGMENT *segment, FILE *file)
{
  efwrite (&segment->id, sizeof (int), 1, file);
  efwrite (&segment->torsion_id, sizeof (int), 1, file);
  efwrite (&segment->layer_id, sizeof (int), 1, file);
  efwrite (&segment->heavy_total, sizeof (int), 1, file);
  efwrite (&segment->periph_flag, sizeof (int), 1, file);
  efwrite (&segment->active_flag, sizeof (int), 1, file);
  efwrite (&segment->min_flag, sizeof (int), 1, file);
  efwrite (&segment->conform_count, sizeof (int), 1, file);
  efwrite (&segment->conform_total, sizeof (int), 1, file);
  save_score (&segment->score, file);

  efwrite (&segment->atom_total, sizeof (int), 1, file);
  efwrite (&segment->atom_max, sizeof (int), 1, file);
  efwrite (segment->atom, sizeof (int), segment->atom_max, file);

  efwrite (&segment->neighbor_total, sizeof (int), 1, file);
  efwrite (&segment->neighbor_max, sizeof (int), 1, file);
  efwrite (segment->neighbor, sizeof (LINK), segment->neighbor_max, file);
}

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

void load_segments (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.segments; i++)
    load_segment (&molecule->segment[i], file);
}

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

void load_segment (SEGMENT *segment, FILE *file)
{
  efread (&segment->id, sizeof (int), 1, file);
  efread (&segment->torsion_id, sizeof (int), 1, file);
  efread (&segment->layer_id, sizeof (int), 1, file);
  efread (&segment->heavy_total, sizeof (int), 1, file);
  efread (&segment->periph_flag, sizeof (int), 1, file);
  efread (&segment->active_flag, sizeof (int), 1, file);
  efread (&segment->min_flag, sizeof (int), 1, file);
  efread (&segment->conform_count, sizeof (int), 1, file);
  efread (&segment->conform_total, sizeof (int), 1, file);
  load_score (&segment->score, file);

  efread (&segment->atom_total, sizeof (int), 1, file);
  efread (&segment->atom_max, sizeof (int), 1, file);
  allocate_segment_atoms (segment);
  efread (segment->atom, sizeof (int), segment->atom_max, file);

  efread (&segment->neighbor_total, sizeof (int), 1, file);
  efread (&segment->neighbor_max, sizeof (int), 1, file);
  allocate_segment_neighbors (segment);
  efread (segment->neighbor, sizeof (LINK), segment->neighbor_max, file);
}

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

int get_segment_neighbor
(
  void          *segment,
  int           segment_id,
  int           neighbor_id
)
{
  if (neighbor_id < ((SEGMENT *) segment)[segment_id].neighbor_total)
    return ((SEGMENT *) segment)[segment_id].neighbor[neighbor_id].id;

  else
    return EOF;
}

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

void allocate_layers (MOLECULE *molecule)
{
  int i;

  if (molecule->max.layers > 0)
```

```
  {
    ecalloc
    (
      (void **) &molecule->layer,
      molecule->max.layers,
      sizeof (LAYER),
      molecule->info.name,
      global.outfile
    );
  }

  for (i = 0; i < molecule->max.layers; i++)
    reset_layer (molecule, i);
}

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

void allocate_layer_segments (LAYER *layer)
{
  if (layer->segment_max > 0)
  {
    ecalloc
    (
      (void **) &layer->segment,
      layer->segment_max,
      sizeof (int),
      "layer atoms",
      global.outfile
    );
  }
}

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

void reset_layers (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.layers; i++)
    reset_layer (molecule, i);

  molecule->total.layers = 0;
}

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

void reset_layer (MOLECULE *molecule, int layer)
{
  int segment;
  int segment_id;

  for
  (
    segment_id = 0;
    segment_id < molecule->layer[layer].segment_total;
    segment_id++
  )
  {
    segment = molecule->layer[layer].segment[segment_id];
    molecule->segment[segment].layer_id = NEITHER;
  }

  molecule->layer[layer].id = NEITHER;
  molecule->layer[layer].active_flag = FALSE;
  molecule->layer[layer].conform_total = 1;
  reset_score (&molecule->layer[layer].score);

  molecule->layer[layer].segment_total = 0;
  memset
  (
    molecule->layer[layer].segment,
    0,
    molecule->layer[layer].segment_max * sizeof (int)
  );
}

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

void free_layers (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.layers; i++)
    free_layer_segments (&molecule->layer[i]);

  efree ((void **) &molecule->layer);
  molecule->max.layers = 0;
}

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

void free_layer_segments (LAYER *layer)
{
  efree ((void **) &layer->segment);
  layer->segment_max = 0;
}

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

void reallocate_layers (MOLECULE *molecule)
{
  if (molecule->total.layers > molecule->max.layers)
  {
    free_layers (molecule);
    molecule->max.layers = molecule->total.layers;
    allocate_layers (molecule);
  }
}

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

void reallocate_layer_segments (LAYER *layer)
{
  if (layer->segment_total > layer->segment_max)
  {
    free_layer_segments (layer);
    layer->segment_max = layer->segment_total;
    allocate_layer_segments (layer);
  }
}

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

void copy_layers (MOLECULE *copy, MOLECULE *original)
{
  int i, j;
  int segment;

  copy->total.layers = original->total.layers;
```

```
  reallocate_layers (copy);

  for (i = 0; i < original->total.layers; i++)
  {
    copy_layer (&copy->layer[i], &original->layer[i]);

    for (j = 0; j < copy->layer[i].segment_total; j++)
    {
      segment = copy->layer[i].segment[j];

      if ((segment >= 0) && (segment < copy->total.segments))
        copy->segment[segment].layer_id = i;

      else
        exit (fprintf (global.outfile,
          "ERROR copy_layers: unable to update segments\n"));
    }
  }
}

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

void copy_layer (LAYER *copy, LAYER *original)
{
  copy->id = original->id;
  copy->active_flag = original->active_flag;
  copy->conform_total = original->conform_total;
  copy_score (&copy->score, &original->score);

  copy->segment_total = original->segment_total;
  reallocate_layer_segments (copy);
  memcpy
    (copy->segment, original->segment,
    original->segment_total * sizeof (int));
}

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

void save_layers (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.layers; i++)
    save_layer (&molecule->layer[i], file);
}

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

void save_layer (LAYER *layer, FILE *file)
{
  efwrite (&layer->id, sizeof (int), 1, file);
  efwrite (&layer->active_flag, sizeof (int), 1, file);
  efwrite (&layer->conform_total, sizeof (int), 1, file);
  save_score (&layer->score, file);

  efwrite (&layer->segment_total, sizeof (int), 1, file);
  efwrite (&layer->segment_max, sizeof (int), 1, file);
  efwrite (layer->segment, sizeof (int), layer->segment_max, file);
}

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

void load_layers (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.layers; i++)
    load_layer (&molecule->layer[i], file);
}

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

void load_layer (LAYER *layer, FILE *file)
{
  efread (&layer->id, sizeof (int), 1, file);
  efread (&layer->active_flag, sizeof (int), 1, file);
  efread (&layer->conform_total, sizeof (int), 1, file);
  load_score (&layer->score, file);

  efread (&layer->segment_total, sizeof (int), 1, file);
  efread (&layer->segment_max, sizeof (int), 1, file);
  allocate_layer_segments (layer);
  efread (layer->segment, sizeof (int), layer->segment_max, file);
}


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

void allocate_sets (MOLECULE *molecule)
{
  int i;

  if (molecule->max.sets > 0)
    ecalloc
    (
      (void **) &molecule->set,
      molecule->max.sets,
      sizeof (SET),
      molecule->info.name,
      global.outfile
    );

  for (i = 0; i < molecule->max.sets; i++)
    reset_set (&molecule->set[i]);
}

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

void reset_sets (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.sets; i++)
    reset_set (&molecule->set[i]);

  molecule->total.sets = 0;
}

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

void reset_set (SET *set)
{
  efree ((void **) &set->name);
  efree ((void **) &set->type);
  efree ((void **) &set->obj_type);
  efree ((void **) &set->sub_type);
  efree ((void **) &set->status);
  efree ((void **) &set->comment);
```

```
  efree ((void **) &set->member);

  memset (set, 0, sizeof (SET));
}

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

void free_sets (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.sets; i++)
    free_set (&molecule->set[i]);

  efree ((void **) &molecule->set);
  molecule->max.sets = 0;
}

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

void free_set (SET *set)
{
  efree ((void **) &set->name);
  efree ((void **) &set->type);
  efree ((void **) &set->obj_type);
  efree ((void **) &set->sub_type);
  efree ((void **) &set->status);
  efree ((void **) &set->comment);
  efree ((void **) &set->member);

  set->member_total = 0;
}

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

void reallocate_sets (MOLECULE *molecule)
{
  if (molecule->total.sets > molecule->max.sets)
  {
    free_sets (molecule);
    molecule->max.sets = molecule->total.sets;
    allocate_sets (molecule);
  }
}

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

void copy_sets (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total sets = original->total.sets;
  reallocate_sets (copy);

  for (i = 0; i < original->total.sets; i++)
    copy_set (&copy->set[i], &original->set[i]);

  copy->total.sets = original->total.sets;
}

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

void copy_set (SET *copy, SET *original)
{
  int i;

  vstrcpy (&copy->name, original->name);
  vstrcpy (&copy->type, original->type);
  vstrcpy (&copy->obj_type, original->obj_type);
  vstrcpy (&copy->sub_type, original->sub_type);
  vstrcpy (&copy->status, original->status);
  vstrcpy (&copy->comment, original->comment);

  if (copy->member_total != original->member_total)
  {
    efree ((void **) &copy->member);
    ecalloc
    (
      (void **) &copy->member,
      original->member_total,
      sizeof (int),
      "set members",
      global.outfile
    );
    copy->member_total = original->member_total;
  }

  for (i = 0; i < original->member_total; i++)
    copy->member[i] = original->member[i];
}

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

void save_sets (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.sets; i++)
    save_set (&molecule->set[i], file);
}

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

void save_set (SET *set, FILE *file)
{
  efwrite (set, sizeof (SET), 1, file);

  save_string (&set->name, file);
  save_string (&set->type, file);
  save_string (&set->obj_type, file);
  save_string (&set->sub_type, file);
  save_string (&set->status, file);
  save_string (&set->comment, file);

  efwrite (set->member, sizeof (int), set->member_total, file);
}

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

void load_sets (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.sets; i++)
    load_set (&molecule->set[i], file);
}

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

void load_set (SET *set, FILE *file)
{
  efread (set, sizeof (SET), 1, file);

  set->name = NULL;
  set->type = NULL;
  set->obj_type = NULL;
  set->sub_type = NULL;
  set->status = NULL;
  set->comment = NULL;
  set->member = NULL;

  load_string (&set->name, file);
  load_string (&set->type, file);
  load_string (&set->obj_type, file);
  load_string (&set->sub_type, file);
  load_string (&set->status, file);
  load_string (&set->comment, file);

  emalloc
  (
    (void **) &set->member,
    set->member_total * sizeof (int),
    "loaded set members",
    global.outfile
  );

  efread (set->member, sizeof (int), set->member_total, file);
}

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

void allocate_keys (MOLECULE *molecule)
{
  int i;

  if (molecule->max.keys > 0)
  {
    ecalloc
    (
      (void **) &molecule->key,
      molecule->max.keys,
      sizeof (KEY *),
      molecule->info.name,
      global.outfile
    );

    for (i = 0; i < molecule->max.keys; i++)
      ecalloc
      (
        (void **) &molecule->key[i],
        molecule->max.keys,
        sizeof (KEY),
        molecule->info.name,
        global.outfile
      );
  }
}

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

void reset_keys (MOLECULE *molecule)
{
  int i, j;

  for (i = 0; i < molecule->max.keys; i++)
    for (j = 0; j < molecule->max.keys; j++)
    {
      molecule->key[i][j].count = 0;
      molecule->key[i][j].distance = (MASK) 0;
    }

  molecule->total.keys = 0;
}

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

void free_keys (MOLECULE *molecule)
{
  int i;

  for (i = 0; i < molecule->max.keys; i++)
    efree ((void **) &molecule->key[i]);

  efree ((void **) &molecule->key);
  molecule->max.keys = 0;
}

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

void reallocate_keys (MOLECULE *molecule)
{
  if (molecule->total.keys > molecule->max.keys)
  {
    free_keys (molecule);
    molecule->max.keys = molecule->total.keys;
    allocate_keys (molecule);
  }
}

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

void copy_keys (MOLECULE *copy, MOLECULE *original)
{
  int i;

  copy->total.keys = original->total.keys;
  reallocate_keys (copy);

  for (i = 0; i < original->total.keys; i++)
    memcpy
    (
      copy->key[i],
      original->key[i],
      original->total.keys * sizeof (KEY)
    );
}

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

void save_keys (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.keys; i++)
    efwrite (molecule->key[i], sizeof (KEY), molecule->max.keys, file);
}
```

```
/* ///////////////////////////////////////////////////////////////// */

void load_keys (MOLECULE *molecule, FILE *file)
{
  int i;

  for (i = 0; i < molecule->max.keys; i++)
    efread (molecule->key[i], sizeof (KEY), molecule->max.keys, file);
}

/* ///////////////////////////////////////////////////////////////// */
/*
Routine to save the contents of a dynamically allocated character
string.
2/96 te
*/

void save_string (char **string, FILE *file)
{
  int length;

/*
* If the string has been allocated, then find the length + 1 (for
* the null character)
* 2/96 te
*/
  if (*string)
    length = strlen (*string) + 1;

  else
    length = 0;

  efwrite (&length, sizeof (int), 1, file);

  if (length > 0)
    efwrite (*string, sizeof (char), length, file);
}

/* ///////////////////////////////////////////////////////////////// */
/*
Routine to load the contents of a dynamically allocated character
string.
2/96 te
*/

void load_string (char **string, FILE *file)
{
  int length;

  efread (&length, sizeof (int), 1, file);

  if (length > 0)
  {
    ecalloc
    (
      (void **) string,
      length,
      sizeof (char),
      "molecule string",
      global.outfile
    );

    efread (*string, sizeof (char), length, file);
  }
}


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

void atom_neighbors (MOLECULE *molecule)
{
  int i;
  int origin;
  int target;
  int neighbor;

  for (i = 0; i < molecule->total.atoms; i++)
  {
    molecule->atom[i].neighbor_total = 0;
    molecule->atom[i].flag = TRUE;
  }

  if (molecule->total.bonds > 0)
  {
/*
*   Compute the total neighbors for each atom
*   12/96 te
*/
    for (i = 0; i < molecule->total.bonds; i++)
    {
      molecule->atom[molecule->bond[i].origin].neighbor_total++;
      molecule->atom[molecule->bond[i].target].neighbor_total++;
    }

/*
*   Allocate space for atom neighbor lists and reset totals
*   12/96 te
*/
    for (i = 0; i < molecule->total.atoms; i++)
    {
      reallocate_atom_neighbors (&molecule->atom[i]);
      molecule->atom[i].neighbor_total = 0;
    }

/*
*   Record bond information into atom neighbor lists
*   12/96 te
*/
    for (i = 0; i < molecule->total.bonds; i++)
    {
      origin = molecule->bond[i].origin;
      target = molecule->bond[i].target;

      if ((origin < 0) || (origin >= molecule->total.atoms))
        exit (fprintf (global.outfile,
          "ERROR atom_neighbors: origin atom incorrect for bond %d\n", i + 1));

      if ((target < 0) || (target >= molecule->total.atoms))
        exit (fprintf (global.outfile,
          "ERROR atom_neighbors: target atom incorrect for bond %d\n", i + 1));

      neighbor = molecule->atom[origin].neighbor_total;
      molecule->atom[origin].neighbor[neighbor].id = target;
      molecule->atom[origin].neighbor[neighbor].bond_id = i;

      if (molecule->atom[target].flag == TRUE)
        molecule->atom[origin].neighbor[neighbor].out_flag = TRUE;

      molecule->atom[target].flag = FALSE;

      molecule->atom[origin].neighbor_total++;

      neighbor = molecule->atom[target].neighbor_total;
      molecule->atom[target].neighbor[neighbor].id = origin;
      molecule->atom[target].neighbor[neighbor].bond_id = i;
      molecule->atom[target].neighbor[neighbor].out_flag = FALSE;
      molecule->atom[target].neighbor_total++;
    }
  }
}


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

void flag_atom_neighbor
(
  void  *atom,
  int   atom_id,
  int   neighbor_id,
  int   flag
)
{
  ((ATOM *) atom)[atom_id].neighbor[neighbor_id].out_flag = flag;
}


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

void flag_segment_neighbor
(
  void  *segment,
  int   segment_id,
  int   neighbor_id,
  int   flag
)
{
  ((SEGMENT *) segment)[segment_id].neighbor[neighbor_id].out_flag = flag;
}


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

void revise_atom_neighbors (MOLECULE *molecule)
{
  SEARCH search = {0};
  int i, j;
  int bond;

/*
* Perform breadth-first atom search, starting from anchor segment.
* Also, flag atom neighbors during this search run.
* 11/96 te
*/
  for
  (
    i = 0;
    breadth_search
    (
      &search,
      molecule->atom,
      molecule->total.atoms,
      get_atom_neighbor,
      flag_atom_neighbor,
      &molecule->transform.anchor_atom, 1,
      NEITHER, i
    ) != EOF;
    i++
  );

/*
* Update the origin and target entries in bond records
* 1/97 te
*/
  for (i = 0; i < molecule->total.atoms; i++)
    for (j = 0; j < molecule->atom[i].neighbor_total; j++)
      if (molecule->atom[i].neighbor[j].out_flag == TRUE)
      {
        bond = molecule->atom[i].neighbor[j].bond_id;

        if (molecule->bond[bond].target == i)
        {
          molecule->bond[bond].origin = i;
          molecule->bond[bond].target = molecule->atom[i].neighbor[j].id;
        }
      }

  free_search (&search);
}

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

Subroutine to compute the centrality of an atom.  This is done by performing
a breadth-first search to identify the bond distance of the most distant atom
from each atom.

11/96 te

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

int get_atom_centrality (MOLECULE *molecule, int atom_id)
{
  int i;
  static SEARCH search = {0};

  if (molecule->atom[atom_id].centrality != NEITHER)
    return molecule->atom[atom_id].centrality;

  else
  {
    for
    (
      i = molecule->atom[atom_id].centrality = 0;
      breadth_search
      (
        &search,
        molecule->atom,
        molecule->total.atoms,
        get_atom_neighbor,
        NULL,
        &atom_id, 1,
        NEITHER,
        i
      ) != EOF;
      i++
    )
      molecule->atom[atom_id].centrality += search.radius;

    return molecule->atom[atom_id].centrality;
/*
```

```
    return
      molecule->atom[atom_id].centrality =
        get_search_radius (&search, NEITHER, NEITHER);
*/
  }
}


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

Subroutine that determines the neighboring atoms to the origin and target
atoms for each torsion.

11/96 te

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

void get_torsion_neighbors (MOLECULE *molecule)
{
  int i, j, k;
  int bond_id;
  int origin;
  int target;
  int neighbor;
  int central_neighbor;
  int central;
  static SEARCH search = {0};

  for (i = 0; i < molecule->total.torsions; i++)
  {
    bond_id = molecule->torsion[i].bond_id;
    origin = molecule->torsion[i].origin = molecule->bond[bond_id].origin;
    target = molecule->torsion[i].target = molecule->bond[bond_id].target;

/*
*   Determine which neighbor of the origin atom is most central
*   11/96 te
*/
    if (molecule->atom[origin].neighbor_total > 1)
    {
      for
      (
        j = central_neighbor = 0, central = INT_MIN;
        (neighbor = get_atom_neighbor (molecule->atom, origin, j)) != EOF;
        j++
      )
      {
        if (neighbor != target)
        {
          for
          (
            k = 0;
            breadth_search
            (
              &search,
              molecule->atom,
              molecule->total.atoms,
              get_atom_neighbor,
              NULL,
              &neighbor, 1,
              origin,
              k
            ) != EOF;
            k++
          );

          if (search.radius > central)
          {
            central = search.radius;
            central_neighbor = neighbor;
          }
        }
      }

      molecule->torsion[i].origin_neighbor = central_neighbor;
    }

    else
      molecule->torsion[i].origin_neighbor = NEITHER;

/*
*   Determine which neighbor of the target atom is most central
*   11/96 te
*/
    if (molecule->atom[target].neighbor_total > 1)
    {
      for
      (
        j = central_neighbor = 0, central = INT_MIN;
        (neighbor = get_atom_neighbor (molecule->atom, target, j)) != EOF;
        j++
      )
      {
        if (neighbor != origin)
        {
          for
          (
            k = 0;
            breadth_search
            (
              &search,
              molecule->atom,
              molecule->total.atoms,
              get_atom_neighbor,
              NULL,
              &neighbor, 1,
              target,
              k
            ) != EOF;
            k++
          );

          if (search.radius > central)
          {
            central = search.radius;
            central_neighbor = neighbor;
          }
        }
      }

      molecule->torsion[i].target_neighbor = central_neighbor;
    }

    else
      molecule->torsion[i].target_neighbor = NEITHER;
  }
}


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

Subroutine that switches the origin and target atoms in a torsion.

11/96 te

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

void reverse_torsion (TORSION *torsion)
{
  int atom;

  atom = torsion->origin;
  torsion->origin = torsion->target;
  torsion->target = atom;

  atom = torsion->origin_neighbor;
  torsion->origin_neighbor = torsion->target_neighbor;
  torsion->target_neighbor = atom;
}


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

Recursive subroutine that counts up the number of atoms directly
and indirectly bonded to a given atom.

10/95 te

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

int bonded_atoms
(
  MOLECULE *molecule,
  int current_atom
)
{
  int i;
  int total;            /* Number of atoms seen on linkage path */

  if (molecule->atom[current_atom].flag)
    return 0;

  else
  {
    molecule->atom[current_atom].flag = TRUE;
    total = 1;

    for (i = 0; i < molecule->atom[current_atom].neighbor_total; i++)
      total +=
        bonded_atoms
        (
          molecule,
          molecule->atom[current_atom].neighbor[i].id
        );

    return total;
  }
}


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

Routine to construct a distance matrix.
3/97 te

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

float calculate_distances (MOLECULE *molecule, float ***distances, int *size)
{
  int i, j;
  float distance_max;

/*
* Either allocate space for distance matrix, or reset previous space
* 3/97 te
*/
  if (*size < molecule->total.atoms)
  {
    free_distances (distances, size);

    *size = molecule->total.atoms;

    ecalloc
    (
      (void **) distances,
      *size,
      sizeof (float *),
      "distance matrix",
      global.outfile
    );

    for (i = 0; i < *size; i++)
      ecalloc
      (
        (void **) &(*distances)[i],
        *size,
        sizeof (float),
        "distance matrix",
        global.outfile
      );
  }

  else
  {
    for (i = 0; i < *size; i++)
      memset ((*distances)[i], 0, *size * sizeof (float));
  }

/*
* Compute the distance between all pairs of atoms
* 3/97 te
*/
  for (i = distance_max = 0; i < molecule->total.atoms; i++)
  {
    for (j = i + 1; j < molecule->total.atoms; j++)
    {
      (*distances)[i][j] =
        (*distances)[j][i] =
          dist3 (molecule->coord[i], molecule->coord[j]);

      if ((*distances)[i][j] > distance_max)
        distance_max = (*distances)[i][j];
    }
  }

  return distance_max;
}
```

```
/* ////////////////////////////////////////////////////////////////

Routine to free distance matrix.
5/97 te

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

void free_distances (float ***distances, int *size)
{
  int i;

  for (i = 0; i < *size; i++)
    efree ((void **) &(*distances)[i]);

  efree ((void **) distances);
  *size = 0;
}

******************************************************************************
********************************** MOL_PREP.H ********************************
******************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
2/96
*/

/* Routines used to read and write molecule data */

int prepare_molecule
(
  MOLECULE      *molecule,
  FILE_NAME     molecule_file_name,
  FILE          *molecule_file,
  LABEL         *label,
  int           atom_model,
  int           need_bonds,
  int           assign_chemical,
  int           assign_vdw
);

void deduce_bonds       (LABEL_VDW *, int, MOLECULE *);

******************************************************************************
********************************** MOL_PREP.C ********************************
******************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label.h"
#include "mol_prep.h"

typedef struct link_bond
{
  BOND bond;
  struct link_bond *previous, *next;
} LINK_BOND;


int prepare_molecule
(
  MOLECULE *molecule,
  FILE_NAME molecule_file_name,
  FILE *molecule_file,
  LABEL *label,
  int atom_model,
  int need_bonds,
  int assign_chemical,
  int assign_vdw
)
{
  int return_value;

/*
* If bonds are needed, but were not read in, then deduce them
* 10/95 te
*/
  if (need_bonds && (molecule->total.bonds == 0))
    deduce_bonds (&label->vdw, atom_model, molecule);

/*
* Update atom neighbor information
* 9/95 te
*/
  if (molecule->total.bonds > 0)
    atom_neighbors (molecule);

/*
  fprintf (global.outfile, "Breadth search\n");
  for
  (
    i = 0;
    (atom_id = breadth_search
      (
        &search,
        molecule->atom,
        molecule->total atoms,
        get_atom_neighbor,
        NULL, &i, 1,
        NEITHER,
        i
      )) != EOF;
    i++
  )
  {
    origin = get_search_origin (&search, NEITHER, NEITHER, NEITHER);
    target = get_search_target (&search, NEITHER, NEITHER, NEITHER);
    thread = get_search_thread (&search, NEITHER, NEITHER);

    fprintf (global.outfile, "%d %d %s %d %s %d %s %d\n",
      get_search_radius (&search, NEITHER, NEITHER),
      atom_id, molecule->atom[atom_id].name,
      origin, (origin >= 0) ? molecule->atom[origin].name : "UNK",
      target, (target >= 0) ? molecule->atom[target].name : "UNK",
      thread);
  }
*/

/*
  fprintf (global.outfile, "Depth search\n");
  for
  (
    i = 0;
    (atom_id = depth_search
      (
        &search,
        molecule->atom,
        molecule->total.atoms,
        get_atom_neighbor,
        0,
        NEITHER,
        i
      )) != EOF;
    i++
  )
  {
    origin = get_search_origin (&search, NEITHER, NEITHER, NEITHER);
    target = get_search_target (&search, NEITHER, NEITHER, NEITHER);

    fprintf (global.outfile, "%d %d %s %d %s %d %s\n",
      get_search_radius (&search, NEITHER, NEITHER),
      atom_id, molecule->atom[atom_id] name,
      origin, (origin >= 0) ? molecule->atom[origin].name : "UNK",
      target, (target >= 0) ? molecule->atom[target].name : "UNK");
  }
*/

/*
* Assign new chemical labels.  If the input file was not SPH format
* 10/95 te
*/
  if (assign_chemical)
  {
    if (strcmp (strrchr (molecule_file_name, '.'), ".sph"))
    {
      if ((return_value =
        assign_chemical_labels (&label->chemical, molecule)) != TRUE)
        return return_value;
    }

    else
      read_chemical_labels
       (&label->chemical, molecule, molecule_file_name, molecule_file);
  }

/*
* Assign vdw label types to atoms
* 10/95 te
*/
  if (assign_vdw)
    if ((return_value =
      assign_vdw_labels (&label->vdw, atom_model, molecule)) != TRUE)
      return return_value;

  return TRUE;
}

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

void deduce_bonds
(
  LABEL_VDW *label_vdw,
  int atom_model,
  MOLECULE *molecule
)
{
  int i, j;
  LINK_BOND *first_bond = NULL;
  LINK_BOND *previous_bond = NULL;
  LINK_BOND *current_bond = NULL;
  float cutoff;

  float square_distance (XYZ, XYZ);

/*
* Make preliminary vdw type assignments
* 10/95 te
*/
  if (!assign_vdw_labels (label_vdw, atom_model, molecule))
    exit (EXIT_FAILURE);

/*
* Loop through all possible pairs of atoms, checking interatomic distance.
* If the distance is below some cutoff, then assume the two atoms are bonded.
* 10/95 te
*/
  molecule->total.bonds = 0;

  for (i = 0; i < molecule->total.atoms; i++)
    for (j = i + 1; j < molecule->total.atoms; j++)
    {
      cutoff = 0.53 *
        (label_vdw->member[molecule->atom[i].vdw_id].radius +
        label_vdw->member[molecule->atom[j].vdw_id].radius);

      if (square_distance (molecule->coord[i], molecule->coord[j]) <
        SQR (cutoff))
      {
/*
*       Allocate memory for this BOND link
*       6/95 te
*/
        previous_bond = current_bond;
        current_bond = NULL;

        ecalloc
        (
          (void **) &current_bond,
          1,
          sizeof (LINK_BOND),
          "linked bond list",
          global.outfile
        );

/*
*       Update pointers between BOND links, initialize first link
*       6/95 te
*/
        current_bond->previous = previous_bond;

        if (previous_bond)
          previous_bond->next = current_bond;
        else
          first_bond = current_bond;
```

```
/*
*       Deposit bond info in this BOND link
*       6/95 te
*/
        current_bond->bond.origin = i;
        current_bond->bond.target = j;
        vstrcpy (&current_bond->bond.type, "1");
        molecule->total.bonds++;
      }
    }

  molecule->max.bonds = molecule->total.bonds;
  allocate_bonds (molecule);

/*
* Copy bond info into molecule structure
* 6/95 te
*/
  for (i = 0, current_bond = first_bond;
    (i < molecule->total.bonds) && (current_bond != NULL);
    i++, current_bond = current_bond->next)
    copy_bond (&molecule->bond[i], &current_bond->bond);

  if (i != molecule->total.bonds)
    exit (fprintf (global.outfile,
      "ERROR deduce_bonds: Error reading bond linked list.\n"));

/*
* Free up memory allocated to linked list
* 6/95 te
*/
  for (previous_bond = first_bond;
    previous_bond != NULL;
    previous_bond = current_bond)
  {
    current_bond = previous_bond->next;
    efree ((void **) &previous_bond);
  }
}

*************************************************************************
************************************ ORIENT.H ***************************
*************************************************************************
/*                                                                     */
/*                     Copyright UCSF, 1997                            */
/*                                                                     */

/*
Written by Todd Ewing
10/95
*/

typedef struct orient_struct
{
  int flag;                     /* Flag for orienting molecule */
  int init_flag;                /* Flag for initializing orient variables */
  int random_flag;              /* Flag for random orientation sampling */
  int uniform_flag;             /* Flag for uniform orientation sampling */
  int max;                      /* Maximum number of orientations */

  XYZ span;                     /* Half width of box */
  XYZ center;                   /* Center of box */

  int *orient_list;
  int *target_list;

  MOLECULE orient;
  MOLECULE target;

  MATCH match;

} ORIENT;


/*
Routines defined in transform.c, that are called by outside functions
*/

int get_orientation
(
  DOCK          *dock,
  ORIENT        *orient,
  LABEL         *label,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_conf,
  MOLECULE      *mol_ori,
  int           molecule_id,
  int           conformation_id,
  int           orientation_id
);

void free_orients
(
  LABEL         *label,
  ORIENT        *orient
);

int get_orient
(
  ORIENT        *orient,
  LABEL         *label,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_ori,
  int           molecule,
  int           conformation,
  int           orientation
);

void free_orient
(
  ORIENT        *orient
);

*************************************************************************
************************************ ORIENT.C ***************************
*************************************************************************
/*                                                                     */
/*                     Copyright UCSF, 1997                            */
/*                                                                     */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "vector.h"
```

```
#include "search.h"
#include "transform.h"
#include "dock.h"
#include "label.h"
#include "score.h"
#include "score_dock.h"
#include "match.h"
#include "orient.h"


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

int get_orientation
(
  DOCK          *dock,
  ORIENT        *orient,
  LABEL         *label,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_conf,
  MOLECULE      *mol_ori,
  int           molecule_id,
  int           conformation_id,
  int           orientation_id
)
{
/*
* Match molecule to site points
* 10/96 te
*/
  if (orient->match.flag)
  {
    if (orient->random_flag)
      get_random_match
      (
        &orient->match,
        label,
        mol_conf,
        molecule_id,
        conformation_id,
        orientation_id
      );

    else if (get_match
      (
        dock,
        &orient->match,
        label,
        mol_conf,
        conformation_id,
        orientation_id
      ) == EOF)
      return EOF;

    extract_clique (&orient->match, mol_ori);

    orient_molecule
    (
      &orient->match.receptor_clique,
      &orient->match.ligand_clique,
      mol_ref, mol_conf, mol_ori
    );

    return TRUE;
  }

  else if (orient->random_flag)
  {
    return
      get_orient
      (
        orient,
        label,
        mol_ref,
        mol_ori,
        molecule_id,
        conformation_id,
        orientation_id
      );
  }

  else
  {
    if (orientation_id == 0)
      return TRUE;

    else
      return EOF;
  }
}

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

void free_orients
(
  LABEL         *label,
  ORIENT        *orient
)
{
  if (orient->match.flag)
  {
    if (orient->random_flag)
      free_random_matches (&orient->match);

    else
      free_matches (label, &orient->match);
  }

  else if (orient->random_flag)
    free_orient (orient);
}

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

Routine to randomly orient a ligand in a rectangular box
2/97 te

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

int get_orient
(
  ORIENT        *orient,
  LABEL         *label,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_ori,
  int           molecule,
  int           conformation,
  int           orientation
```

```
}
{
  int i, j;
  XYZ max = {FLT_MIN, FLT_MIN, FLT_MIN},
  XYZ min = {FLT_MAX, FLT_MAX, FLT_MAX};

  if (!molecule && !conformation && !orientation)
  {
    get_site (&orient->match, label);
    get_centers (&orient->match, label);

    for (i = 0; i < orient->match.receptor_site.total.atoms; i++)
    {
      for (j = 0; j < 3; j++)
      {
        min[j] = MIN (min[j], orient->match receptor_site.coord[i][j]);
        max[j] = MAX (max[j], orient->match.receptor_site.coord[i][j]);
      }
    }

    for (j = 0; j < 3; j++)
    {
      orient->center[j] = (max[j] + min[j]) / 2.0;
      orient->span[j] = (max[j] - min[j]) / 2.0;
    }
  }

  for (i = 0; i < 3; i++)
  {
    mol_ori->transform translate[i] =
      orient->center[i] +
      orient->span[i] * (1.0 - 2.0 * (float) rand() / (float) RAND_MAX) -
      (orient->match.centers_flag
        ? orient->match ligand_center.transform.com[i]
        : mol_ref->transform.com[i]);

    mol_ori->transform.rotate[i] =
      1.0 - 2.0 * (float) rand() / (float) RAND_MAX;
  }

  mol_ori->transform.trans_flag =
    mol_ori->transform.rot_flag = TRUE;

  transform_molecule (mol_ori, mol_ref);

  return TRUE;
}

void free_orient (ORIENT *orient)
{
  free_molecule (&orient->match.receptor_site);
  free_molecule (&orient->match.ligand_center);
}
******************************************************************************
************************************* PARM.H *********************************
******************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

typedef struct parm_struct
{
  STRING200 *line;
  int total;

} PARM;

enum VARIABLE_TYPE {Boolean, Integer, Real, Character, String};

void read_parameters (PARM *parm);

void get_parameter
{
  void                  *variable,
  PARM                  *parm,
  enum VARIABLE_TYPE    variable_type,
  char                  name[],
  char                  suggest[],
  int                   query
);

int yes_or_no (char *value);
******************************************************************************
************************************* PARM.C *********************************
******************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/
#include <time.h>
#include <sys/resource.h>
#include "define h"
#include "utility.h"
#include "global.h"
#include "parm.h"

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

void read_parameters (PARM *parm)
{
  STRING200 line;

  if (global.infile == stdin)
    return;

/*
* Count the number of lines in the parameter file
* 3/96 te
*/
  for
  (
    parm->total = 0;
    fgets (line, sizeof (STRING200), global.infile) != NULL;
    parm->total++
  );

  fseek (global.infile, 0, SEEK_SET);

/*
* Allocate memory for storing each line and keyword read from parameter file
* 3/96 te
*/
  if (parm->total > 0)
    ecalloc
    (
      (void **) &parm->line,
      parm->total,
      sizeof (STRING200),
      "parm line buffer",
      global.outfile
    );

  for
  (
    parm->total = 0;
    fgets (parm->line[parm->total], sizeof (STRING200), global.infile) != NULL;
    parm->total++
  );

  fseek (global.infile, 0, SEEK_END);
}


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

void get_parameter
(
  void *variable,
  PARM *parm,
  enum VARIABLE_TYPE variable_type,
  char name[],
  char suggest[],
  int query
)
{
  int i;
  int find_value;
  int get_value;
  STRING200 value = "";

  find_value = FALSE;
  get_value = FALSE;

/*
* Get a value from the user
* 3/96 te
*/
  if (query)
  {
/*
*   If we have buffered input lines, then scan them for a value
*   3/96 te
*/
    if ((global.infile != stdin) && (parm->total > 0))
    {
      for (i = 0; i < parm->total; i++)
        if (!strncmp (parm->line[i], name, strlen (name)))
        {
          if (sscanf (parm->line[i], "%*s %s", value) == 1)
          {
            find_value = TRUE;
            get_value = TRUE;
          }

          break;
        }
    }

/*
*   If we don't have buffered input lines, or the value wasn't found,
*   then try asking for the value interactively.
*   3/96 te
*/
    if (((global.infile == stdin) || !find_value) &&
      (global.outfile == stdout))
    {
      fprintf (global.outfile, "%-30s [%s] ", name, suggest);

      if (!strcmp (gets (value), ""))
        sscanf (suggest, "%s", value);

      get_value = TRUE;
    }
  }

/*
* If user not queried, the use the suggested value
* 3/96 te
*/
  else
  {
    sscanf (suggest, "%s", value);
    find_value = TRUE;
    get_value = TRUE;
  }

/*
* Check if a value was read
* 3/96 te
*/
  if (!get_value)
    exit (fprintf (global.outfile,
      "ERROR get_parameter: Unable to get value for %s parameter.\n", name));

/*
* Store the value in the variable
* 3/96 te
*/
  get_value = FALSE;

  switch (variable_type)
  {
    case Boolean:
      if (tolower (value[0]) == 'y')
      {
        *(int *) variable = TRUE;
        strcpy (value, "yes");
      }

      else
      {
        *(int *) variable = FALSE;
        strcpy (value, "no");
      }

      get_value = TRUE;

      break;
```

```
    case Integer:
      if (!strcmp (value, "<infinity>"))
      {
        *(int *) variable = INT_MAX;
        get_value = TRUE;
      }

      else if (sscanf (value, "%d", variable) == 1)
      {
        sprintf (value, "%d", *(int *) variable);
        get_value = TRUE;
      }

      break;

    case Real:
      if (!strcmp (value, "<infinity>"))
      {
        *(float *) variable = FLT_MAX;
        get_value = TRUE;
      }

      else if (sscanf (value, "%f", variable) == 1)
      {
        sprintf (value, "%g", *(float *) variable);
        get_value = TRUE;
      }

      break;

    case Character:
      if (*(int *) variable = (int) tolower (value[0]))
      {
        if (*(int *) variable == 'y')
          sprintf (value, "yes");

        else if (*(int *) variable == 'n')
          sprintf (value, "no");

        else
          sprintf (value, "%c", *(int *) variable);

        get_value = TRUE;
      }

      break;

    case String:
      if (sscanf (value, "%s", variable) == 1)
        get_value = TRUE;

      break;

    default:
      exit (fprintf (global.outfile,
        "ERROR get_parameter: Unknown data type for parameter.\n"));
  }

/*
* Check if value was stored in variable
* 3/96 te
*/
  if (!get_value)
    exit (fprintf (global.outfile,
      "ERROR get_parameter: Unable to store value for %s parameter.\n", name));

/*
* Update the input file if:
*   An input file is being read AND either
*     The parameter was missing from the file, but was read interactively, OR
*     The parameter was NOT read from the file but running in verbose mode
* 3/96 te
*/
  if ((global.infile != stdin) && query && !find_value)
    fprintf (global.infile, "%-30s %s\n", name, value);

/*
* Report value stored in variable to output file, if:
*   The value was queried, but not entered interactively, or
*   The value was not queried and verbose mode
* 3/96 te
*/
  if ((query && find_value) || (!query && (global.output_volume == 'v')))
    fprintf (global.outfile, "%-30s %s\n", name, value);
}


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

Set limit on how much memory can be dynamically allocated for this run
11/95 te

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

void set_memory_limit (void)
{
  time_t tp;
  struct rlimit total, resident;

/*
* Set the total memory limit to the resident amount (help prevent swapping)
* 11/95 te
*/
  getrlimit (RLIMIT_RSS, &resident);
  getrlimit (RLIMIT_DATA, &total);

  resident.rlim_cur = total.rlim_cur = MIN (resident.rlim_cur, total.rlim_cur);

  setrlimit (RLIMIT_RSS, &resident);
  setrlimit (RLIMIT_DATA, &total);

/*
* Get the current time
* 11/95 te
*/
  time (&tp);

/*
* Output the information about this job
* 11/95 te
*/
  fprintf (global.outfile,
    "\n________________Job_Information________________\n");

  fprintf (global.outfile, "%-30s %s", "launch_time", ctime (&tp));
  fprintf (global.outfile, "%-30s %s\n", "host_name",
    getenv("HOST") ? getenv("HOST") : "unknown");
  fprintf (global.outfile, "%-30s %d\n", "memory_limit",
    total.rlim_cur);
  fprintf (global.outfile, "%-30s %s\n", "working_directory",
    getenv("PWD") ? getenv("PWD") : "unknown");
  fprintf (global.outfile, "%-30s %s\n", "user_name",
    getenv("USER") ? getenv("USER") : "unknown");
}

################################################################################
################################## PARM_DOCK.H #################################
################################################################################
/*                                                                            */
/*                         Copyright UCSF, 1997                               */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

void get_parameters
(
  DOCK *,
  ORIENT *,
  SCORE *,
  LABEL *
);

int process_commands (DOCK *, int argc, char *argv[]);

################################################################################
################################## PARM_DOCK.C #################################
################################################################################
/*                                                                            */
/*                         Copyright UCSF, 1997                               */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "io.h"
#include "score.h"
#include "score_dock.h"
#include "match.h"
#include "orient.h"
#include "parm.h"
#include "parm_dock.h"

void get_parameters
(
  DOCK *dock,
  ORIENT *orient,
  SCORE *score,
  LABEL *label
)
{
  int i, j;

  PARM parm = {0};

  STRING40 parameter_name;
  STRING100 parameter_value;

  enum FILE_FORMAT format;

/*
* Read parameters into buffer
* 8/96 te
*/
  read_parameters (&parm);

/*
* Begin reading in individual parameters
* 8/96 te
*/
  fprintf (global.outfile,
    "\n_______________General_Parameters_______________\n");

  get_parameter
  (
    (void *) &label->chemical.screen.flag,
    &parm, Boolean, "chemical_screen",
    "NO yes",
    TRUE
  );

  label->chemical.flag = label->chemical.screen.flag;

  get_parameter
  (
    (void *) &label->flex.flag,
    &parm, Boolean, "flexible_ligand",
    "NO yes",
    TRUE
  );

  get_parameter
  (
    (void *) &orient->flag,
    &parm, Boolean, "orient_ligand",
    "NO yes",
    !label->chemical.screen.flag
  );

  dock->multiple_orients =
    dock->multiple_orients || orient->flag;

  get_parameter
  (
    (void *) &dock->multiple_ligands,
    &parm, Boolean, "multiple_ligands",
    "NO yes",
    TRUE
  );

  get_parameter
  (
    (void *) &score->flag,
    &parm, Boolean, "score_ligand",
    "NO yes",
    !label->chemical.screen.flag || label->flex.flag
  );
```

```
label->vdw.flag =
  orient->flag || label->flex.flag || score->flag || dock->multiple_ligands;

get_parameter
(
  (void *) &score->minimize.flag,
  &parm, Boolean, "minimize_ligand",
  "NO yes",
  score->flag
);

get_parameter
(
  (void *) &dock->parallel.flag,
  &parm, Boolean, "parallel_jobs",
  "NO yes",
  TRUE
);

get_parameter
(
  (void *) &i,
  &parm, Integer, "random_seed",
  "0",
  label->flex.flag || score->minimize.flag || orient->flag
);

srand (i);


if (label->chemical.screen.flag || (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n____________Chemical_Screen_Parameters____________\n");

get_parameter
(
  (void *) &label->chemical.screen.construct_flag,
  &parm, Boolean, "construct_screen",
  "NO yes",
  label->chemical.screen.flag
);

label->vdw.flag = label->vdw.flag || label->chemical.screen.flag;

get_parameter
(
  (void *) &label->chemical.screen.process_flag,
  &parm, Boolean, "screen_ligands",
  "NO yes",
  label->chemical.screen.flag && !label->chemical.screen.construct_flag
);

if (label->chemical.screen.flag &&
  !label->chemical.screen.process_flag &&
  !label->chemical.screen.construct_flag)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: No chemical screen options selected!\n"));

get_parameter
(
  (void *) &label->chemical.screen.pharmaco_flag,
  &parm, Boolean, "pharmacophore_screen",
  "NO yes",
  label->chemical.screen.process_flag
);

get_parameter
(
  (void *) &label->chemical.screen similar_flag,
  &parm, Boolean, "similarity_screen",
  "NO yes",
  label->chemical.screen.process_flag &&
    !label->chemical.screen.pharmaco_flag
);

get_parameter
(
  (void *) &label->chemical.screen.fold_flag,
  &parm, Boolean, "fold_keys",
  "YES",
  FALSE
);

if (label->chemical.screen.process_flag &&
  !label->chemical.screen.pharmaco_flag &&
  !label->chemical.screen.similar_flag)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: No screen ligand options selected!\n"));

get_parameter
(
  (void *) &label->chemical.screen.dissimilar_maximum,
  &parm, Real, "dissimilarity_maximum",
  "0.25",
  label->chemical.screen.similar_flag
);

get_parameter
(
  (void *) &label->chemical.screen.distance_minimum,
  &parm, Real, "distance_begin",
  (label->chemical.screen.construct_flag ||
    label->chemical.screen.pharmaco_flag ||
    label->chemical.screen.similar_flag) ? "2" : "0",
  label->chemical.screen.construct_flag ||
    label->chemical.screen.pharmaco_flag ||
    label->chemical.screen.similar_flag
);

get_parameter
(
  (void *) &label->chemical.screen.distance_maximum,
  &parm, Real, "distance_end",
  (label->chemical.screen.construct_flag ||
    label->chemical.screen.pharmaco_flag ||
    label->chemical.screen.similar_flag) ? "17" : "0",
  label->chemical screen.construct_flag ||
    label->chemical screen.pharmaco_flag ||
    label->chemical.screen.similar_flag
);

get_parameter
(
  (void *) &label->chemical.screen.distance_interval,
  &parm, Real, "distance_interval",
  (label->chemical.screen.construct_flag ||
    label->chemical.screen.pharmaco_flag ||
    label->chemical.screen.similar_flag) ? "0.5" : "0",
  label->chemical.screen.construct_flag ||
    label->chemical.screen.pharmaco_flag ||
    label->chemical.screen.similar_flag
);

label->chemical.screen.interval_total = (int)
  ((label->chemical.screen.distance_maximum -
  label->chemical.screen.distance_minimum) /
  label->chemical.screen.distance_interval) + 2;

if (label->chemical.screen.interval_total > MASK_LENGTH)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: Chemical screen intervals > %d!\n",
    MASK_LENGTH - 2));


if (label->flex.flag || (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n____________Flexible_Ligand_Parameters____________\n");

get_parameter
(
  (void *) &label->flex.anchor_flag,
  &parm, Boolean, "anchor_search",
  "NO yes",
  score->flag && label->flex.flag
);

get_parameter
(
  (void *) &label->flex.multiple_anchors,
  &parm, Boolean, "multiple_anchors",
  "NO yes",
  label->flex.anchor_flag
);

get_parameter
(
  (void *) &label->flex.anchor_size,
  &parm, Integer, "anchor_size",
  label->flex multiple_anchors ? "10" : "0",
  label->flex.multiple_anchors
);

get_parameter
(
  (void *) &label->flex.periph_flag,
  &parm, Boolean, "peripheral_search",
  "NO yes",
  label->flex.anchor_flag
);

get_parameter
(
  (void *) &label->flex.periph_seeds,
  &parm, Integer, "peripheral_seeds",
  label->flex.periph_flag ? "25" : "1",
  label->flex.periph_flag
);

get_parameter
(
  (void *) &label->flex.drive_flag,
  &parm, Boolean, "torsion_drive",
  "NO yes",
  label->flex.flag && (!label->flex.anchor_flag || label->flex.periph_flag)
);

dock->multiple_conforms = label->flex.drive_flag;

get_parameter
(
  (void *) &label->flex.clash_overlap,
  &parm, Real, "clash_overlap",
  label->flex.drive_flag ? "0.5" : "0",
  label->flex.drive_flag
);

get_parameter
(
  (void *) &label->flex.max_conforms,
  &parm, Integer, "maximum_conformations",
  label->flex.drive_flag && !label->flex.anchor_flag ? "1000" : "1",
  label->flex.drive_flag && !label->flex.anchor_flag
);

get_parameter
(
  (void *) &label->flex.minimize_flag,
  &parm, Boolean, "torsion_minimize",
  "NO yes",
  label->flex.flag && score->minimize.flag &&
    (!label->flex.anchor_flag || label->flex.periph_flag)
);

get_parameter
(
  (void *) &label->flex.reminimize_layers,
  &parm, Integer, "reminimize_layer_number",
  label->flex.minimize_flag && label->flex.periph_flag ? "2" : "0",
  label->flex.minimize_flag && label->flex.periph_flag
);

get_parameter
(
  (void *) &label->flex.minimize_anchor_flag,
  &parm, Boolean, "minimize_anchor",
  score->minimize.flag && !label->flex.periph_flag ? "YES" : "NO yes",
  score->minimize.flag && label->flex periph_flag
);

get_parameter
(
  (void *) &label->flex.reminimize_anchor_flag,
  &parm, Boolean, "reminimize_anchor",
  "NO yes",
  score->minimize.flag && label->flex.periph_flag
);

get_parameter
(
  (void *) &label->flex.reminimize_ligand_flag,
  &parm, Boolean, "reminimize_ligand",
  "NO yes",
  score->minimize.flag && label->flex.periph_flag
);

get_parameter
(
```

```
  (void *) &label->flex.max_torsions,
  &parm, Integer, "flexible_bond_maximum",
  label->flex.flag && dock->multiple_ligands ? "10" : "<infinity>",
  label->flex.flag && dock->multiple_ligands
);


if (orient->flag || (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n____________Orient_Ligand_Parameters____________\n");

get_parameter
(
  (void *) &orient->match.flag,
  &parm, Boolean, "match_receptor_sites",
  label->chemical.screen.pharmaco_flag &&
    label->chemical.screen.fold_flag ? "YES" : "NO yes",
  orient->flag
);

get_parameter
(
  (void *) &orient->match.centers_flag,
  &parm, Boolean, "match_ligand_centers",
  "NO yes",
  orient->flag && !dock->multiple_ligands
);

get_parameter
(
  (void *) &orient->random_flag,
  &parm, Boolean, "random_search",
  "NO yes",
  orient->flag
);

if (orient->flag &&
  !orient->match.flag &&
  !orient->random_flag)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: No orient_ligand options selected\n"));

get_parameter
(
  (void *) &orient->uniform_flag,
  &parm, Boolean, "uniform_sampling",
  orient->flag ? "YES no" : "NO yes",
  orient->flag && !orient->random_flag
);

orient->match.uniform_flag = orient->match.flag && orient->uniform_flag;

get_parameter
(
  (void *) &orient->max,
  &parm, Integer, "total_orientations",
  orient->uniform_flag ? (dock->multiple_ligands ? "100" : "500") : "0",
  orient->uniform_flag
);

get_parameter
(
  (void *) &dock->write_orients,
  &parm, Boolean, "write_orientations",
  "NO yes",
  orient->flag
);

get_parameter
(
  (void *) &dock->rank_orients,
  &parm, Boolean, "rank_orientations",
  "NO yes",
  score->flag && dock->write_orients
);

get_parameter
(
  (void *) &dock->rank_orient_total,
  &parm, Integer, "rank_orientation_total",
  dock->rank_orients ? "100" : "1",
  dock->rank_orients
);

if (label->flex.periph_flag == TRUE)
{
  dock->rank_anchors = TRUE;
  dock->rank_anchor_total = orient->flag ? orient->max : 1;
}

else
{
  dock->rank_anchors = dock->rank_orients;
  dock->rank_anchor_total = dock->rank_orient_total;
}


if ((orient->match.flag && !orient->random_flag) ||
  (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n________________Match_Parameters________________\n");

get_parameter
(
  (void *) &orient->match.clique_size_min,
  &parm, Integer, "nodes_minimum",
  orient->match.flag ? "3" : "0",
  orient->match.flag && !orient->match.uniform_flag
);

get_parameter
(
  (void *) &orient->match.clique_size_max,
  &parm, Integer, "nodes_maximum",
  orient->match.flag ? (orient->match.uniform_flag ? "3" : "10") : "0",
  orient->match.flag && !orient->match.uniform_flag
);

get_parameter
(
  (void *) &orient->match.distance_tolerance,
  &parm, Real, "distance_tolerance",
  orient->match.flag ? "0.25" : "0",
  orient->match.flag && !orient->match.uniform_flag
);

get_parameter
(
  (void *) &orient->match.distance_minimum,
  &parm, Real, "distance_minimum",
  orient->flag && orient->match.flag ? "2.0" : "0",
  orient->flag && orient->match.flag && !orient->match.uniform_flag
);

get_parameter
(
  (void *) &orient->match.degeneracy_flag,
  &parm, Boolean, "check_degeneracy",
  "NO yes",
  orient->flag && orient->match.flag && !orient->uniform_flag
);

get_parameter
(
  (void *) &orient->match.reflect_flag,
  &parm, Boolean, "reflect_ligand",
  "NO yes",
  orient->flag && orient->match.flag && !orient->uniform_flag &&
    (orient->match.clique_size_max > 3)
);

get_parameter
(
  (void *) &orient->match.critical_flag,
  &parm, Boolean, "critical_points",
  "NO yes",
  orient->flag && orient->match.flag && !orient->random_flag
);

get_parameter
(
  (void *) &orient->match.multiple_flag,
  &parm, Boolean, "multiple_points",
  "NO yes",
  orient->match.critical_flag
);

get_parameter
(
  (void *) &orient->match.chemical_flag,
  &parm, Boolean, "chemical_match",
  label->chemical.screen.pharmaco_flag ? "YES" : "NO yes",
  orient->flag && orient->match.flag && !orient->random_flag
);

label->chemical.flag =
  label->chemical.flag || orient->match.chemical_flag;


if (dock->multiple_ligands || (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n___________Multiple_Ligand_Parameters____________\n");

get_parameter
(
  (void *) &dock->max_ligands,
  &parm, Integer, "ligands_maximum",
  dock->multiple_ligands ? "1000" : "1",
  dock->multiple_ligands
);

get_parameter
(
  (void *) &dock->initial_skip,
  &parm, Integer, "initial_skip",
  "0",
  dock->multiple_ligands
);

get_parameter
(
  (void *) &dock->interval_skip,
  &parm, Integer, "interval_skip",
  "0",
  dock->multiple_ligands
);

get_parameter
(
  (void *) &dock->min_heavies,
  &parm, Integer, "heavy_atoms_minimum",
  "0",
  dock->multiple_ligands
);

get_parameter
(
  (void *) &dock->max_heavies,
  &parm, Integer, "heavy_atoms_maximum",
  orient->flag && dock->multiple_ligands ? "100" : "<infinity>",
  dock->multiple_ligands
);

get_parameter
(
  (void *) &dock->rank_ligands,
  &parm, Boolean,
  "rank_ligands",
  "NO yes",
  dock->multiple_ligands &&
    (score->flag || label->chemical.screen.similar_flag) &&
    (!dock->write_orients || dock->rank_orients)
);

get_parameter
(
  (void *) &dock->rank_ligand_total,
  &parm, Integer,
  "rank_ligand_total",
  dock->rank_ligands ? "100" : "1",
  dock->rank_ligands
);

get_parameter
(
  (void *) &dock->restart_interval,
  &parm, Integer, "restart_interval",
  dock->rank_ligands ? "100" : "0",
  dock->rank_ligands
);


if (score->flag || (global.output_volume == 'v'))
  fprintf (global.outfile,
    "\n________________Scoring_Parameters_______________\n");

get_parameter
(
```

```
  (void *) &score->intra_flag,
  &parm, Boolean, "intramolecular_score",
  label->flex.flag && score->flag ? "YES no" : "NO yes",
  label->flex.flag && score->flag
);

get_parameter
(
  (void *) &score->inter_flag,
  &parm, Boolean, "intermolecular_score",
  score->flag ? "YES no" : "NO yes",
  score->flag
);

if (score->flag && !score->intra_flag && !score->inter_flag)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: No scoring options selected\n"));

get_parameter
(
  (void *) &score->grid.flag,
  &parm, Boolean, "gridded_score",
  score->inter_flag ? "YES no" : "NO yes",
  score->inter_flag
);

get_parameter
(
  (void *) &score->grid.version,
  &parm, Real, "grid_version",
  "4.0",
  score->grid.flag
);

if ((score->grid.version < 3.0) || (score->grid.version > 4.0))
  exit (fprintf (global.outfile,
    "ERROR get_parameters: grid_version selection not supported\n"));

get_parameter
(
  (void *) &score->grid.size,
  &parm, Integer, "grid_points",
  (score->grid.version < 4) ? "1000000" : "0",
  score->grid.version < 4
);

get_parameter
(
  (void *) &score->grid.spacing,
  &parm, Real, "receptor_atom_grid_spacing",
  score->inter_flag && !score->grid.flag ? "2.0" : "0.0",
  FALSE
);

if (score->inter_flag && !score->grid.flag && (score->grid.spacing <= 0.0))
  exit (fprintf (global.outfile,
    "ERROR get_parameters: Innappropriate grid_spacing value.\n"));

get_parameter
(
  (void *) &score->bump.flag,
  &parm, Boolean, "bump_filter",
  "NO yes",
  orient->flag && score->grid.flag
);

get_parameter
(
  (void *) &score->bump.maximum,
  &parm, Integer, "bump_maximum",
  "0",
  score->bump.flag
);

score->type[NONE].flag = !score->flag;
strcpy (score->type[NONE].name, "none");
strcpy (score->type[NONE].abbrev, "out");

get_parameter
(
  (void *) &score->contact.flag,
  &parm, Boolean, "contact_score",
  "NO yes",
  score->flag &&
    (!score->grid.flag || (score->grid.version >= 4.0))
);

score->type[CONTACT].flag = score->contact.flag;
strcpy (score->type[CONTACT].name, "contact");
strcpy (score->type[CONTACT].abbrev, "cnt");

get_parameter
(
  (void *) &score->contact.distance,
  &parm, Real, "contact_cutoff_distance",
  score->contact.flag && (!score->grid.flag || score->intra_flag) ?
    "4.0" : "0.0",
  score->contact.flag && (!score->grid.flag || score->intra_flag)
);

get_parameter
(
  (void *) &score->contact.clash_overlap,
  &parm, Real, "contact_clash_overlap",
  score->contact.flag && (score->intra_flag || !score->grid.flag)
    ? "0.75" : "0.0",
  score->contact.flag && (score->intra_flag || !score->grid.flag)
);

get_parameter
(
  (void *) &score->contact.clash_penalty,
  &parm, Real, "contact_clash_penalty",
  score->contact.flag ? "50" : "0",
  score->contact.flag
);

get_parameter
(
  (void *) &score->chemical.flag,
  &parm, Boolean, "chemical_score",
  "NO yes",
  score->flag &&
    (!score->grid.flag || (score->grid.version >= 4.0))
);

label->chemical.flag =
  label->chemical.flag || score->chemical.flag;

score->type[CHEMICAL].flag = score->chemical.flag;
strcpy (score->type[CHEMICAL].name, "chemical");
strcpy (score->type[CHEMICAL].abbrev, "chm");

get_parameter
(
  (void *) &score->energy.flag,
  &parm, Boolean, "energy_score",
  "NO yes",
  score->flag
);

score->type[ENERGY].flag = score->energy.flag;
strcpy (score->type[ENERGY].name, "energy");
strcpy (score->type[ENERGY].abbrev, "nrg");

get_parameter
(
  (void *) &score->energy.distance,
  &parm, Real, "energy_cutoff_distance",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag) ? "10.0" : "0.0",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag)
);

get_parameter
(
  (void *) &score->energy.distance_dielectric,
  &parm, Boolean, "distance_dielectric",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag) ? "YES no" : "NO",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag)
);

get_parameter
(
  (void *) &score->energy.dielectric_factor,
  &parm, Real, "dielectric_factor",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag) ? "4.0" : "0.0",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag)
);

get_parameter
(
  (void *) &score->energy.attractive_exponent,
  &parm, Integer, "attractive_exponent",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag) ? "6" : "0",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag)
);

get_parameter
(
  (void *) &score->energy.repulsive_exponent,
  &parm, Integer, "repulsive_exponent",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag) ? "12" : "0",
  (!score->grid.flag || score->intra_flag) &&
    (score->energy.flag || score->chemical.flag)
);

get_parameter
(
  (void *) &score->energy.atom_model,
  &parm, Character, "atom_model",
  score->energy.flag || score->chemical.flag ? "UNITED all" : "0",
  score->energy.flag || score->chemical.flag
);

get_parameter
(
  (void *) &score->energy.scale_vdw,
  &parm, Real, "vdw_scale",
  score->energy.flag || score->chemical.flag ? "1" : "0",
  score->energy.flag || score->chemical.flag
);

get_parameter
(
  (void *) &score->energy.scale_electro,
  &parm, Real, "electrostatic_scale",
  score->energy.flag || score->chemical.flag ? "1" : "0",
  score->energy.flag || score->chemical.flag
);

get_parameter
(
  (void *) &score->energy.decomp_flag,
  &parm, Boolean, "output_atom_scores",
  "NO yes",
  FALSE
);

get_parameter
(
  (void *) &score->rmsd.flag,
  &parm, Boolean, "rmsd_score",
  "NO yes",
  score->inter_flag && !score->intra_flag &&
    !score->grid.flag && !orient->flag
);

score->type[RMSD].flag = score->rmsd.flag;
strcpy (score->type[RMSD].name, "rmsd");
strcpy (score->type[RMSD].abbrev, "rmsd");

if
(
  score->inter_flag &&
  !score->contact.flag &&
  !score->chemical.flag &&
  !score->energy.flag &&
  !score->rmsd.flag
)
  exit (fprintf (global.outfile,
    "ERROR get_parameters: no intermolecular scoring options selected.\n"));

for (i = 1; i < SCORE_TOTAL; i++)
{
  sprintf (parameter_name, "%s_maximum", score->type[i].name);

  get_parameter
  (
    (void *) &score->type[i].maximum,
```

```
      &parm, Real, parameter_name,
      "0",
      score->type[i].flag &&
        ((dock->write_orients && !dock->rank_orients) ||
        (!dock->multiple_orients &&
          dock->multiple_ligands && !dock->rank_ligands))
    );

    sprintf (parameter_name, "%s_size_penalty", score->type[i].name);

    get_parameter
    (
      (void *) &score->type[i].size_penalty,
      &parm, Real, parameter_name,
      "0",
      score->type[i].flag && dock->rank_ligands && (i != RMSD)
    );
  }

  get_parameter
  (
    (void *) &score->rmsd_override,
    &parm, Real, "rmsd_override",
    "0.0",
    score->flag && !dock->rank_ligands &&
      dock->write_orients && !dock->rank_orients
  );


  if (score->minimize.flag || (global.output_volume == 'v'))
    fprintf (global.outfile,
      "\n______________Minimization_Parameters_____________\n");

  for (i = 1, j = 0; i < SCORE_TOTAL; i++)
  {
    sprintf (parameter_name, "%s_minimize", score->type[i].name);

    get_parameter
    (
      (void *) &score->type[i].minimize,
      &parm, Boolean, parameter_name,
      "NO yes",
      score->type[i].flag && score->minimize.flag
    );

    if (score->type[i].minimize == TRUE) j++;
  }

  if
  (
    (score->minimize.flag == TRUE) &&
    ((j == 0) ||
    ((score->inter_flag == FALSE) &&
      (label->flex.minimize_flag == FALSE)))
  )
    exit (fprintf (global.outfile,
      "ERROR get_parameters: no minimization selected\n"));

  get_parameter
  (
    (void *) &score->minimize.translation,
    &parm, Real, "initial_translation",
    score->inter_flag && score->minimize.flag ? "1.0" : "0.0",
    score->inter_flag && score->minimize.flag
  );

  get_parameter
  (
    (void *) &score->minimize.rotation,
    &parm, Real, "initial_rotation",
    score->inter_flag && score->minimize.flag ? "0.1" : "0.0",
    score->inter_flag && score->minimize.flag
  );

  get_parameter
  (
    (void *) &score->minimize.torsion,
    &parm, Real, "initial_torsion",
    label->flex.minimize_flag ? "10.0" : "0.0",
    label->flex.minimize_flag
  );

  get_parameter
  (
    (void *) &score->minimize.iteration,
    &parm, Integer, "maximum_iterations",
    score->minimize.flag ? "100" : "0",
    score->minimize.flag
  );

  if ((score->minimize.flag == TRUE) && (score->minimize.iteration < 1))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: maximum_iterations < 1\n"));

  for (i = 1; i < SCORE_TOTAL; i++)
  {
    sprintf (parameter_name, "%s_convergence", score->type[i].name);

    get_parameter
    (
      (void *) &score->type[i].convergence,
      &parm, Real, parameter_name,
      score->type[i].minimize && (score->minimize.iteration > 1) ? "0.1" : "0",
      score->type[i].minimize && (score->minimize.iteration > 1)
    );
  }

  get_parameter
  (
    (void *) &score->minimize.cycle,
    &parm, Integer, "maximum_cycles",
    score->minimize.flag ? "1" : "0",
    score->minimize.flag
  );

  if ((score->minimize.flag == TRUE) && (score->minimize.cycle < 1))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: maximum_cycles < 1\n"));

  get_parameter
  (
    (void *) &score->minimize.cycle_converge,
    &parm, Real, "cycle_convergence",
    (score->minimize.cycle > 1) ? "1.0" : "0",
    score->minimize.cycle > 1
  );

  if ((score->minimize.cycle > 1) && (score->minimize.cycle_converge <= 0))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: cycle_convergence <= 0\n"));

  for (i = 1; i < SCORE_TOTAL; i++)
  {
    sprintf (parameter_name, "%s_termination", score->type[i].name);

    get_parameter
    (
      (void *) &score->type[i].termination,
      &parm, Real, parameter_name,
      score->type[i].minimize && (score->minimize.cycle > 1) ? "1.0" : "0",
      score->type[i].minimize && (score->minimize.cycle > 1)
    );
  }


  if (dock->parallel.flag || (global.output_volume == 'v'))
    fprintf (global.outfile,
      "\n______________Parallel_Job_Parameters_____________\n");

  get_parameter
  (
    (void *) &dock->parallel.server,
    &parm, Boolean, "parallel_server",
    "NO yes",
    dock->parallel.flag
  );

/*
* Make sure only a single type of scoring has been requested
* 10/96 te
*/
  if (dock->parallel.server)
  {
    for (i = j = 0; i < SCORE_TOTAL; i++)
      if (score->type[i].flag) j++;

    if (j != 1)
      exit (fprintf (global.outfile,
        "ERROR get_parameters: "
        "Parallel job server must perform single type of scoring.\n"));
  }

  get_parameter
  (
    (void *) &dock->parallel.server_name,
    &parm, String, "server_name",
    dock->parallel.server ? global.job_name : "server",
    dock->parallel.flag
  );

  get_parameter
  (
    (void *) &dock->parallel.client_total,
    &parm, Integer, "client_total",
    dock->parallel.server ? "5" : (dock->parallel.flag ? "1" : "0"),
    dock->parallel.server
  );

  if (dock->parallel.flag)
  {
    if (dock->parallel.client_total < 1)
      exit (fprintf (global.outfile,
        "ERROR get_parameters: Inappropriate value for client_total.\n"));

    ecalloc
    (
      (void **) &dock->parallel.client_name,
      dock->parallel.client_total,
      sizeof (STRING20),
      "parallel client names",
      global.outfile
    );

    for (i = 0; i < dock->parallel.client_total; i++)
    {
      if (dock->parallel.server)
      {
        sprintf (parameter_name, "client_name_%d", i + 1);
        sprintf (parameter_value, "client%d", i + 1);
      }

      else
      {
        strcpy (parameter_name, "client_name");
        strcpy (parameter_value, global.job_name);
      }

      get_parameter
      (
        (void *) &dock->parallel.client_name[i],
        &parm, String, parameter_name,
        parameter_value,
        TRUE
      );
    }
  }


  fprintf (global.outfile,
    "\n__________________File_Input___________________\n");

  get_parameter
  (
    (void *) &dock->ligand_file_name,
    &parm, String, "ligand_atom_file",
    "ligand.mol2",
    !dock->parallel.flag || dock->parallel.server
  );

  format = check_file_extension (dock->ligand_file_name, TRUE);

  if (format == Unknown)
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "Unrecognized file extension of ligand_atom_file\n"));

  else if (label->chemical.screen.process_flag && (format != Ptr))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "PTR file required to chemical screen ligands.\n"));

  get_parameter
  (
    (void *) &orient->match.ligand_file_name,
    &parm, String, "ligand_center_file",
    "ligand_center.sph",
    orient->match.centers_flag
  );
```

```
  if (check_file_extension (orient->match.ligand_file_name, TRUE) == Unknown)
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "Unrecognized file extension of ligand_center_file\n"));

  get_parameter
  (
    (void *) &orient->match.receptor_file_name,
    &parm, String, "receptor_site_file",
    "receptor_site.sph",
    orient->flag || label->chemical.screen.process_flag
  );

  if (check_file_extension (orient->match.receptor_file_name, TRUE) == Unknown)
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "Unrecognized file extension of receptor_site_file\n"));

  get_parameter
  (
    (void *) &score->grid.file_prefix,
    &parm, String, "score_grid_prefix",
    "score_grid",
    score->inter_flag && score->grid.flag
  );

  get_parameter
  (
    (void *) &score->grid.receptor_file_name,
    &parm, String, "receptor_atom_file",
    "receptor.mol2",
    score->inter_flag && !score->grid.flag
  );

  if (!check_file_extension (score->grid.receptor_file_name, TRUE))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "Unrecognized file extension of receptor_atom_file\n"));

  get_parameter
  (
    (void *) &label->vdw.file_name,
    &parm, String, "vdw_definition_file",
    PARAMETER_PATH "vdw.defn",
    label->vdw.flag
  );

  get_parameter
  (
    (void *) &label->chemical.file_name,
    &parm, String, "chemical_definition_file",
    PARAMETER_PATH "chem.defn",
    label->chemical.flag
  );

  get_parameter
  (
    (void *) &label->chemical.match_file_name,
    &parm, String, "chemical_match_file",
    PARAMETER_PATH "chem_match.tbl",
    orient->match.chemical_flag
  );

  get_parameter
  (
    (void *) &label->chemical.score_file_name,
    &parm, String, "chemical_score_file",
    PARAMETER_PATH "chem_score.tbl",
    score->chemical.flag
  );

  get_parameter
  (
    (void *) &label->chemical.screen_file_name,
    &parm, String, "chemical_screen_file",
    PARAMETER_PATH "chem_screen.tbl",
    label->chemical.screen.similar_flag
  );

  get_parameter
  (
    (void *) &label->flex.file_name,
    &parm, String, "flex_definition_file",
    PARAMETER_PATH "flex.defn",
    label->flex.flag
  );

  get_parameter
  (
    (void *) &label->flex.search_file_name,
    &parm, String, "flex_drive_file",
    PARAMETER_PATH "flex_drive.tbl",
    label->flex.drive_flag
  );

  strcat (strcpy (parameter_value, global.job_name), ".quit");

  get_parameter
  (
    (void *) &dock->quit_file_name,
    &parm, String, "quit_file",
    parameter_value,
    dock->rank_ligands || (dock->multiple_ligands && dock->parallel.flag)
  );

  strcat (strcpy (parameter_value, global.job_name), ".dump");

  get_parameter
  (
    (void *) &dock->dump_file_name,
    &parm, String, "dump_file",
    parameter_value,
    dock->rank_ligands
  );

  fprintf (global.outfile,
    "\n_____________________File_Output___________________\n");

  for (i = 0; i < SCORE_TOTAL; i++)
  {
    sprintf (parameter_name, "ligand_%s_file", i ? score->type[i].name : "out");
    sprintf (parameter_value, "%s_%s.%s", global.job_name,
      score->type[i].abbrev,
      dock->parallel.flag ? "ptr" : "mol2");

    get_parameter
    (
      (void *) &score->type[i].file_name,
      &parm, String, parameter_name,
      parameter_value,
      score->type[i].flag && !dock->parallel.server
    );

    if (score->type[i].flag)
    {
      if (dock->parallel.flag && !dock->parallel.server)
      {
        if (check_file_extension (score->type[i].file_name, TRUE) != Ptr)
          exit (fprintf (global.outfile,
            "ERROR get_parameters: "
            "Parallel client must write in PTR format\n"));
      }

      else if (label->chemical.screen.construct_flag)
      {
        if (check_file_extension (score->type[i].file_name, TRUE) != Ptr)
          exit (fprintf (global.outfile,
            "ERROR get_parameters: "
            "Constructed screen must be written in PTR format\n"));
      }

      else
      {
        if (check_file_extension (score->type[i].file_name, TRUE) == Unknown)
          exit (fprintf (global.outfile,
            "ERROR get_parameters: "
            "Unrecognized file extension of ligand output file\n"));
      }
    }
  }

  strcat (strcpy (parameter_value, global.job_name), ".info");

  get_parameter
  (
    (void *) &dock->info_file_name,
    &parm, String, "info_file",
    parameter_value,
    score->flag && dock->multiple_ligands
  );

  strcat (strcpy (parameter_value, global.job_name), ".rst");

  get_parameter
  (
    (void *) &dock->restart_file_name,
    &parm, String, "restart_file",
    parameter_value,
    dock->rank_ligands
  );

/*
* Flush output and clean up memory
* 11/96 te
*/
  fprintf (global.outfile, "\n\n");
  fflush (global.outfile);

  if (global.infile != stdin)
  {
    fclose (global.infile);
    efree ((void **) &parm.line);
  }
}

/* ////////////////////////////////////////////////////////////////// */
int process_commands
(
  DOCK *dock,
  int argc,
  char *argv[]
)
{
  int i = 1, j;
  int molecule_io = FALSE;
  FILE_NAME infile_name = "", outfile_name = "";

/*
* Extract name of global.executable from command line,
* ignoring any path description if present
* 6/95 te
*/
  strcpy (global.executable,
    strrchr (argv[0], '/') ? strrchr (argv[0], '/') + 1 : argv[0]);
  strcpy (global.job_name, "dock");

/*
* Step through each command line argument
* 6/95 te
*/
  while (i < argc)
  {

/*
*   Look for command flag prefix (-)
*   6/95 te
*/
    if (argv[i][0] == '-')
    {

/*
*     For "-i" flag read in next field as the input file name
*     6/95 te
*/
      if (argv[i][1] == 'i')
      {
        if ((i + 1 < argc) && (argv[i + 1][0] != '-'))
        {
          strncpy (infile_name, argv[i + 1], 80);
          i++;

/*
*         Set the job name to the root name of the the "*.in" file
*         10/95 te
*/
          if (strrchr (infile_name, '.') &&
            !strcmp (strrchr (infile_name, '.'), ".in"))
          {
            for (j = 0; j < strlen (infile_name); j++)
            {
              if (!strcmp (&infile_name[j], ".in"))
              {
                global.job_name[j] = 0;
                break;
              }
```

```
            else
              global.job_name[j] = infile_name[j];
          }
        }

        else
          strcpy (global.job_name, infile_name);
      }

      else
        sprintf (infile_name, "%s.in", global.job_name);
    }

/*
*       For "-o" flag read in next field as the output file name.
*       If the next field does not exist or is another flag, then
*       use default output file name.
*       6/95 te
*/
    else if (argv[i][1] == 'o')
    {
      if ((i + 1 < argc) && (argv[i + 1][0] != '-'))
      {
        strncpy (outfile_name, argv[i + 1], 80);
        i++;
      }

      else
        sprintf (outfile_name, "%s.out", global.job_name);

      if (!strcmp (infile_name, ""))
        break;
    }

/*
*       For "-p" flag, set performance flag
*       1/97 te
*/
    else if (argv[i][1] == 'p')
      dock->performance_flag = TRUE;

/*
*       For "-r" flag, turn on flag to perform restart run
*       6/95 te
*/
    else if (argv[i][1] == 'r')
    {
      if (strcmp (infile_name, ""))
        dock->restart = TRUE;

      else
        break;
    }

/*
*       For "-s" flag, allow all input and output through standard streams
*       6/95 te
*/
    else if (argv[i][1] == 's')
    {
      if ((strcmp (infile_name, "")) || (strcmp (outfile_name, "")))
        break;
    }

/*
*       For "-t" flag, set output volume to terse
*       6/95 te
*/
    else if (argv[i][1] == 't')
      global.output_volume = 't';

/*
*       For "-v" flag, set output volume to verbose
*       6/95 te
*/
    else if (argv[i][1] == 'v')
      global.output_volume = 'v';

    else
      break;

    i++;
  }

  else
    break;
}

/*
* Check if no arguments were given, to conform to old style input/output
* 10/95 te
*/
  if (argc == 1)
  {
/*
*   If INDOCK file exists, then assume user wants output directed to OUTDOCK
*   10/95 te
*/
    if (global.infile = fopen ("INDOCK", "r"))
    {
      fclose (global.infile);

      if (global.infile = fopen ("OUTDOCK", "r"))
      {
        fclose (global.infile);
        fprintf (stderr,
          "ERROR get_parameters: \n"
          "       Presence of INDOCK file implies OUTDOCK to receive output.\n"
          "       but OUTDOCK file already exists.  For reverse compatibility\n"
          "       reasons, this process will stop.\n");
        exit (EXIT_FAILURE);
      }

      strcpy (infile_name, "INDOCK");
      strcpy (outfile_name, "OUTDOCK");
    }

/*
*     If it doesn't, then trigger an error condition
*     10/95 te
*/
    else
      i = 0;
  }

/*
* If any error was made in the command line, then quit
* 6/95 te
*/
  if (i < argc)
  {
    fprintf (stderr,
      "  \n"
      "?[33mUsage: %s [-i [input_file]] [-o [output_file]] ...\n"
      "  [-restart] [-standard_i/o] [-terse] [-verbose]?[0m\n"
      "  \n"
      "  -i: read from %s.in or input_file, standard_in otherwise\n"
      "  -o: write to %s.out or output_file (-i required), \n"
      "      standard_out otherwise\n"
      "  -p: monitor performance\n"
      "  -r: restart run (-i required)\n"
      "  -s: read from and write to standard streams (-i and/or -o illegal)\n"
      "  -t: terse program output\n"
      "  -v: verbose program output\n"
      "  \n",
      global.executable, global.executable, global.executable);
    exit (EXIT_FAILURE);
  }

/*
* Open the input file
* 6/95 te
*/
  if (strcmp (infile_name, ""))
  {
    if (strcmp (outfile_name, ""))
    {
/*
*     Check to see if user has supplied molecule files as input/output
*     6/95 te
*/
      if ((check_file_extension (infile_name, FALSE) != Unknown) &&
        (check_file_extension (outfile_name, FALSE) != Unknown))
      {
        molecule_io = TRUE;

/*
*       Open up scratch file and write necessary parameters to it
*       6/95 te
*/
        global.infile = tmpfile ();

        fprintf (global.infile, "chemical_screen          no\n");
        fprintf (global.infile, "orient_ligand            no\n");
        fprintf (global.infile, "flexible_ligand          no\n");
        fprintf (global.infile, "multiple_ligands         yes\n");
        fprintf (global.infile, "score_ligand             no\n");
        fprintf (global.infile, "parallel_jobs            no\n");
        fprintf (global.infile, "ligands_maximum          <infinity>\n");
        fprintf (global.infile, "initial_skip             0\n");
        fprintf (global.infile, "interval_skip            0\n");
        fprintf (global.infile, "atoms_maximum            <infinity>\n");
        fprintf (global.infile, "heavy_atoms_minimum      0\n");
        fprintf (global.infile, "heavy_atoms_maximum      <infinity>\n");
        fprintf (global.infile, "ligand_atom_file         %s\n", infile_name);
        fprintf (global.infile, "vdw_definition_file      %s\n",
          PARAMETER_PATH "vdw.defn");
        fprintf (global.infile, "ligand_out_file         %s\n", outfile_name);

        rewind (global.infile);
      }

      else
        global.infile = efopen (infile_name, "r", stdout);
    }

    else
      global.infile = efopen (infile_name, "a+", stdout);
  }

  else
    global.infile = stdin;

/*
* Open the output file
* 6/95 te
*/
  if ((strcmp (outfile_name, "")) && (!molecule_io))
  {
    if (dock->restart)
      global.outfile = efopen (outfile_name, "a", stdout);

    else
      global.outfile = efopen (outfile_name, "w", stdout);
  }

  else
    global.outfile = stdout;

  return TRUE;
}

********************************************************************************
********************************** PARM_GRID.H *********************************
********************************************************************************
/*                                                                           */
/*                          Copyright UCSF, 1997                             */
/*                                                                           */

/*
Written by Todd Ewing
10/95
*/

int get_parameters
(
  GRID *,
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL *
);

int process_commands (int argc, char *argv[]);

********************************************************************************
********************************** PARM_GRID.C *********************************
********************************************************************************
/*                                                                           */
/*                          Copyright UCSF, 1997                             */
/*                                                                           */

/*
Written by Todd Ewing
10/95
*/
```

```
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "parm.h"
#include "label.h"
#include "score.h"
#include "score_grid.h"
#include "grid.h"
#include "io.h"

int get_parameters
(
  GRID *grid,
  SCORE_GRID *score_grid,
  SCORE_BUMP *score_bump,
  SCORE_CONTACT *score_contact,
  SCORE_CHEMICAL *score_chemical,
  SCORE_ENERGY *score_energy,
  LABEL *label
)
{
  PARM parm = {0};

  read_parameters (&parm);

  fprintf (global.outfile,
    "\n________________General_Parameters________________\n");

  get_parameter
  (
    (void *) &score_grid->flag,
    &parm, Boolean, "compute_grids",
    "NO yes",
    TRUE
  );

  if (score_grid->flag)
    label->vdw.flag = TRUE;

  get_parameter
  (
    (void *) &score_grid->spacing,
    &parm, Real, "grid_spacing",
    score_grid->flag ? "0.3" : "0.0",
    score_grid->flag
  );

  get_parameter
  (
    (void *) &grid->output_molecule,
    &parm, Boolean, "output_molecule",
    "NO yes",
    TRUE
  );

  if (!score_grid->flag && !grid->output_molecule)
    exit (fprintf (global.outfile,
      "WARNING get_parameters: "
      "No processing requested.  Execution terminated.\n"));

  if (score_grid->flag)
    fprintf (global.outfile,
      "\n________________Scoring_Parameters________________\n");

  get_parameter
  (
    (void *) &score_contact->flag,
    &parm, Boolean, "contact_score",
    "NO yes",
    score_grid->flag
  );

  get_parameter
  (
    (void *) &score_contact->distance,
    &parm, Real, "contact_cutoff_distance",
    score_contact->flag ? "4.0" : "0.0",
    score_contact->flag
  );

  get_parameter
  (
    (void *) &score_chemical->flag,
    &parm, Boolean, "chemical_score",
    "NO yes",
    score_grid->flag
  );

  if (score_chemical->flag)
    label->chemical.flag = TRUE;

  get_parameter
  (
    (void *) &score_energy->flag,
    &parm, Boolean, "energy_score",
    "NO yes",
    score_grid->flag
  );

  if (score_chemical->flag && !score_energy->flag)
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "energy scoring is required with chemical scoring.\n"));

  get_parameter
  (
    (void *) &score_energy->distance,
    &parm, Real, "energy_cutoff_distance",
    score_energy->flag ? "10.0" : "0.0",
    score_energy->flag
  );

  get_parameter
  (
    (void *) &score_energy->atom_model,
    &parm, Character, "atom_model",
    score_energy->flag ? "UNITED all" : "0",
    score_energy->flag
  );

  if (score_energy->flag &&
    (score_energy->atom_model != 'a') &&
    (score_energy->atom_model != 'u'))
    return FALSE;

  get_parameter
  (
    (void *) &score_energy->attractive_exponent,
    &parm, Integer, "attractive_exponent",
    score_energy->flag ? "6" : "0",
    score_energy->flag
  );

  get_parameter
  (
    (void *) &score_energy->repulsive_exponent,
    &parm, Integer, "repulsive_exponent",
    score_energy->flag ? "12" : "0",
    score_energy->flag
  );

  if (score_energy->flag &&
    (score_energy->attractive_exponent >= score_energy->repulsive_exponent))
    exit (fprintf (global.outfile,
      "ERROR get_parameters: Require EXP(repulse) > EXP(attract).\n"));

  get_parameter
  (
    (void *) &score_energy->distance_dielectric,
    &parm, Boolean, "distance_dielectric",
    score_energy->flag ? "YES no" : "NO",
    score_energy->flag
  );

  get_parameter
  (
    (void *) &score_energy->dielectric_factor,
    &parm, Real, "dielectric_factor",
    score_energy->flag ? "4.0" : "0.0",
    score_energy->flag
  );

  get_parameter
  (
    (void *) &score_bump->flag,
    &parm, Boolean, "bump_filter",
    "NO yes",
    score_grid->flag
  );

  if
  (
    score_grid->flag &&
    !score_bump->flag &&
    !score_contact->flag &&
    !score_chemical->flag &&
    !score_energy->flag
  )
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "At least ONE type of scoring must be selected.\n"));

  if
  (
    !score_bump->flag &&
    (score_contact->flag ||
      score_chemical->flag)
  )
    exit (fprintf (global.outfile,
      "ERROR get_parameters: "
      "Contact and chemical scoring require a bump grid.\n"));

  get_parameter
  (
    (void *) &score_bump->clash_overlap,
    &parm, Real, "bump_overlap",
    score_bump->flag ? "0.75" : "0.0",
    score_bump->flag
  );


  fprintf (global.outfile,
    "\n____________________File_Input____________________\n");

  get_parameter
  (
    (void *) &grid->in_file_name,
    &parm, String, "receptor_file",
    "receptor.mol2",
    TRUE
  );

  if (!check_file_extension (grid->in_file_name, TRUE))
    return FALSE;

  get_parameter
  (
    (void *) &grid->box_file_name,
    &parm, String, "box_file",
    "site_box.pdb",
    score_grid->flag
  );

  get_parameter
  (
    (void *) &label->vdw.file_name,
    &parm, String, "vdw_definition_file",
    PARAMETER_PATH "vdw.defn",
    label->vdw.flag
  );

  get_parameter
  (
    (void *) &label->chemical.file_name,
    &parm, String, "chemical_definition_file",
    PARAMETER_PATH "chem.defn",
    label->chemical.flag
  );


  fprintf (global.outfile,
    "\n____________________File_Output____________________\n");

  get_parameter
  (
    (void *) &score_grid->file_prefix,
    &parm, String, "score_grid_prefix",
    global.job_name,
    score_grid->flag
  );

  get_parameter
  (
    (void *) &grid->out_file_name,
    &parm, String, "receptor_out_file",
    "receptor_out.mol2",
```

```
    grid->output_molecule
  );

  if (!check_file_extension (grid->out_file_name, TRUE))
    return FALSE;

  if (global.infile != stdin)
    fclose (global.infile);

  fprintf (global.outfile, "\n\n");
  return TRUE;
}


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

int process_commands
(
  int argc,
  char *argv[]
)
{
  int i = 1, j;
  FILE_NAME infile_name = "", outfile_name = "";
  FILE *tmpfile ();

/*
* Extract name of global executable from command line,
* ignoring any path description if present
* 6/95 te
*/
  strcpy (global.executable,
    strrchr (argv[0], '/') ? strrchr (argv[0], '/') + 1 : argv[0]);
  strcpy (global.job_name, global.executable);

/*
* Step through each command line argument
* 6/95 te
*/
  while (i < argc)
  {

/*
*   Look for command flag prefix (-)
*   6/95 te
*/
    if (argv[i][0] == '-')
    {

/*
*     For "-i" flag read in next field as the input file name
*     6/95 te
*/
      if (argv[i][1] == 'i')
      {
        if ((i + 1 < argc) && (argv[i + 1][0] != '-'))
        {
          strncpy (infile_name, argv[i + 1], 80);
          i++;

/*
*         Set the job name to the root name of the the "*.in" file
*         10/95 te
*/
          if (strrchr (infile_name, '.') &&
            !strcmp (strrchr (infile_name, '.'), ".in"))
          {
            for (j = 0; j < strlen (infile_name); j++)
            {
              if (!strcmp (&infile_name[j], ".in"))
              {
                global.job_name[j] = 0;
                break;
              }

              else
                global.job_name[j] = infile_name[j];
            }
          }

          else
            strcpy (global.job_name, infile_name);
        }

        else
          sprintf (infile_name, "%s.in", global.job_name);
      }

/*
*     For "-o" flag read in next field as the output file name.
*     If the next field does not exist or is another flag, then
*     use default output file name.
*     6/95 te
*/
      else if (argv[i][1] == 'o')
      {
        if ((i + 1 < argc) && (argv[i + 1][0] != '-'))
        {
          strncpy (outfile_name, argv[i + 1], 80);
          i++;
        }

        else
          sprintf (outfile_name, "%s.out", global.job_name);

        if (!strcmp (infile_name, ""))
          break;
      }

/*
*     For "-s" flag, allow all input and output through standard streams
*     6/95 te
*/
      else if (argv[i][1] == 's')
      {
        if ((strcmp (infile_name, "")) || (strcmp (outfile_name, "")))
          break;
      }

/*
*     For "-t" flag, set output volume to terse
*     6/95 te
*/
      else if (argv[i][1] == 't')
        global.output_volume = 't';

/*
*     For "-v" flag, set output volume to verbose
*     6/95 te
*/
      else if (argv[i][1] == 'v')
        global.output_volume = 'v';

      else
        break;

      i++;
    }

    else
      break;
  }

/*
* Check if no arguments were given, to conform to old style input/output
* 10/95 te
*/
  if (argc == 1)
  {
    strcpy (infile_name, "INCHEM");

/*
*   If INCHEM file exists, then assume user wants output directed to OUTCHEM
*   10/95 te
*/
    if (global.infile = fopen (infile_name, "r"))
    {
      fclose (global.infile);
      strcpy (outfile_name, "OUTCHEM");
    }

/*
*   If it doesn't, then trigger an error condition
*   10/95 te
*/
    else
      i = 0;
  }

/*
* If any error was made in the command line, then quit
* 6/95 te
*/
  if (i < argc)
  {
    fprintf (stderr,
      "  \n"
      "?[33mUsage: %s [-i [input_file]] [-o [output_file]] ...\n"
      "  [-s?[0mstandard_i/o?[33m] [-t?[0merse?[33m] [-v?[0merbose?[33m]?[0m\n"
      "  \n"
      "  -i: read from %s.in or input_file, standard_in otherwise\n"
      "  -o: write to %s.out or output_file (-i required), \n"
      "      standard_out otherwise\n"
      "  -s: read from and write to standard streams (-i and/or -o illegal)\n"
      "  -t: terse program output\n"
      "  -v: verbose program output\n"
      "  \n",
      global.executable, global.executable, global.executable);
    exit (EXIT_FAILURE);
  }

/*
* Open the input file
* 6/95 te
*/
  if (strcmp (infile_name, ""))
    global.infile = efopen (infile_name, "a+", stdout);

  else
    global.infile = stdin;

/*
* Open the output file
* 6/95 te
*/
  if (strcmp (outfile_name, ""))
      global.outfile = efopen (outfile_name, "w", stdout);

  else
    global.outfile = stdout;

  return TRUE;
}

********************************************************************************
************************************ RANK.H ************************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

int write_info
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int ligand_read_num,
  int ligand_dock_num,
  int ligand_skip_num,
  float time
);

int write_topscorers
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  MOLECULE *mol_ref,
  MOLECULE *mol_ori
);

int write_restartinfo
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int ligand_read_num,
  int ligand_dock_num,
  int ligand_skip_num,
  float clock_elapsed
);

int read_restartinfo
```

```c
{
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int *ligand_read_num,
  int *ligand_dock_num,
  int *ligand_skip_num,
  float *clock_elapsed
);

############################################################################
################################### RANK.C #################################
############################################################################
/*                                                                   */
/*                        Copyright UCSF, 1997                       */
/*                                                                   */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "score_dock.h"
#include "io.h"
#include "io_ligand.h"
#include "transform.h"
#include "rank.h"


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

int write_info
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int ligand_read_num,
  int ligand_dock_num,
  int ligand_skip_num,
  float time
)
{
  int i, j;
  FILE *infofile = NULL;

  infofile = rfopen (dock->info_file_name, "w", global.outfile);

  fprintf (infofile, "%-35s : %10d\n", "Compounds read", ligand_read_num);
  fprintf (infofile, "%-35s : %10d\n", "Compounds docked", ligand_dock_num);
  fprintf (infofile, "%-35s : %10d\n", "Compounds skipped", ligand_skip_num);
  fprintf (infofile, "%-35s : %10.2f\n", "Elapsed CPU time (sec)", time);
  fprintf (infofile, "%-35s : %10.2f\n", "Time per docked compound (sec)",
    time / (float) ligand_dock_num);

  if (dock->rank_ligands)
    for (i = 0; i < SCORE_TOTAL; i++)
    {
      if (score->type[i].flag)
      {
        fprintf (infofile,
          "\n\nCurrent best %s scorers: \n", score->type[i].name);

        for (j = 0; j < list->total[i]; j++)
          fprintf (infofile, "%2d: %7.2f %s %s\n",
            j + 1,
            list->member[i][j]->score.total,
            list->member[i][j]->info.name,
            (list->member[i][j]->transform.refl_flag == 1
              ? "(REFLECTED)" : ""));
      }
    }

  fclose (infofile);
  return TRUE;
}

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

int write_topscorers
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  MOLECULE *mol_ref,
  MOLECULE *mol_ori
)
{
  int i, j;
  long file_position;

/*
* Save the current position in the ligand input file
* 7/95 te
*/
  if (!dock->parallel.flag || dock->parallel.server)
    file_position = ftell (dock->ligand_file);

/*
* Loop through all requested scoring types
* 7/95 te
*/
  for (i = 0; i < SCORE_TOTAL; i++)
  {
    if (!score->type[i].flag)
      continue;

/*
*   Open up the output file and reset output numbering
*   6/97 te
*/
    if (dock->rank_ligands)
      score->type[i].file =
        rfopen (score->type[i].file_name, "w", global.outfile);

    for (j = 0; j < list->total[i]; j++)
    {
/*
*     If coordinates are needed for output (output file is NOT ptr format),
*     then reread them and transform them
*     6/95 te
*/
      if (check_file_extension (score->type[i].file_name, ".") != Ptr)
```

```c
      {
        reset_molecule (mol_ref);

        fseek
          (dock->ligand_file,
          list->member[i][j]->info.file_position,
          SEEK_SET);

        read_molecule
          (mol_ref, NULL, dock->ligand_file_name, dock->ligand_file, TRUE);

        if (strcmp
          (list->member[i][j]->info.name, mol_ref->info.name) != 0)
        {
          fprintf (global.outfile,
            "ERROR write_topscorers: Read incorrect ligand from input file:\n");
          fprintf (global.outfile, "      Intended : %s\n",
            list->member[i][j]->info.name);
          fprintf (global.outfile, "      Actual   : %s\n",
            mol_ref->info.name);
          exit (EXIT_FAILURE);
        }

        copy_molecule (mol_ori, mol_ref);
        copy_member (list->lite_flag, mol_ori, list->member[i][j]);

        if ((mol_ori->total.torsions > 0) && (mol_ori->transform.tors_flag))
        {
          revise_atom_neighbors (mol_ori);
          get_torsion_neighbors (mol_ori);
        }

        if (mol_ori->transform.flag)
          transform_molecule (mol_ori, mol_ref);

        write_ligand
        (
          dock,
          score,
          mol_ori,
          score->type[i].file_name,
          score->type[i].file
        );
      }

      else
        write_ligand
        (
          dock,
          score,
          list->member[i][j],
          score->type[i].file_name,
          score->type[i].file
        );
    }

    if (dock->rank_ligands)
      fclose (score->type[i].file);
  }

  if (!dock->parallel.flag || dock->parallel.server)
    fseek (dock->ligand_file, file_position, SEEK_SET);

  return TRUE;
}

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

int write_restartinfo
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int ligand_read_num,
  int ligand_dock_num,
  int ligand_skip_num,
  float clock_elapsed
)
{
  long file_position;
  FILE *restart_file;

  void efwrite (void *, size_t, size_t, FILE *);

  file_position = ftell (dock->ligand_file);

  restart_file = efopen (dock->restart_file_name, "w", global.outfile);

  efwrite (&file_position, sizeof (long), 1, restart_file);
  efwrite (&ligand_read_num, sizeof (int), 1, restart_file);
  efwrite (&ligand_dock_num, sizeof (int), 1, restart_file);
  efwrite (&ligand_skip_num, sizeof (int), 1, restart_file);
  efwrite (&clock_elapsed, sizeof (float), 1, restart_file);

  save_lists (score, list, restart_file);

  fclose (restart_file);
  return TRUE;
}

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

int read_restartinfo
(
  DOCK *dock,
  SCORE *score,
  LIST *list,
  int *ligand_read_num,
  int *ligand_dock_num,
  int *ligand_skip_num,
  float *clock_elapsed
)
{
  long file_position;
  FILE *restart_file;

  void efread (void *, size_t, size_t, FILE *);

  restart_file = efopen (dock->restart_file_name, "r", global.outfile);

  efread (&file_position, sizeof (long), 1, restart_file);
  efread (ligand_read_num, sizeof (int), 1, restart_file);
  efread (ligand_dock_num, sizeof (int), 1, restart_file);
  efread (ligand_skip_num, sizeof (int), 1, restart_file);
  efread (clock_elapsed, sizeof (float), 1, restart_file);

  if (!load_lists (score, list, restart_file))
    exit (fprintf (global.outfile,
      "ERROR read_restartinfo: Error reading restart information.\n"));
```

```
fclos
fseek
retur
}

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

/*
Written
9-96
*/

typede
{
  int
  int
  floa

  int
  int
  floa
  floa
  floa

  FILE
  FILE

  mcl
  sli

  int

} sco


typed
{
  in
  in
  ch

  fl
  fl
  f

  in

} s


typ
{
  i
  i
  s
  f
  f
  f

} 

ty
{



}

t
```

USF LIBRARY

```
  fclose (restart_file);
  fseek (dock->ligand_file, file_position, SEEK_SET);
  return TRUE;
}

***********************************************************************************
************************************ SCORE.H **************************************
***********************************************************************************
/*                                                                               */
/*                         Copyright UCSF, 1997                                  */
/*                                                                               */

/*
Written by Todd Ewing
9/96
*/

typedef struct score_grid_struct
{
  int flag;                      /* Flag for precomputed grid use */
  int init_flag;                 /* Flag for grid initialization */
  float version;                 /* Chemgrid version format */

  int size;                      /* Number of grid points */
  int span[3];                   /* Number of points along each grid edge */
  float origin[3];               /* Coordinates of origin of grids */
  float spacing;                 /* Spacing in angstroms between grid points */
  float distance;                /* Longest interaction distance */

  FILE_NAME file_prefix;         /* Filename prefix for grid files */
  FILE_NAME receptor_file_name;  /* Filename of receptor atoms */

  MOLECULE receptor;             /* Receptor atoms for when grids not used */
  SLINT **atom;                  /* 3D grid of receptor atoms */

  int grid_cutoff;               /* Cutoff converted to integer grid units */

} SCORE_GRID;


typedef struct score_bump_struct
{
  int flag;                      /* Flag for bump checking */
  int init_flag;                 /* Flag for bump check initialization */
  char *grid;                    /* Bump grid */

  float clash_energy;            /* Maximum VDW energy for grid point */
  float clash_overlap;           /* VDW overlap allowed of probe with receptor */
  float distance;                /* Longest distance between bumping atoms */

  int maximum;                   /* Maximum bumps allowed for orientation */

} SCORE_BUMP;


typedef struct score_contact_struct
{
  int flag;                      /* Flag for contact score use */
  int init_flag;                 /* Flag for contact score initialization */
  short int *grid;               /* Contact score grid */
  float distance;                /* Contact distance between heavy atoms */
  float clash_overlap;           /* VDW overlap allowed of probe with receptor */
  float clash_penalty;           /* Contact score penalty for each atom clash */

} SCORE_CONTACT;


typedef struct score_chemical_struct
{
  int flag;                      /* Flag for chemical score use */
  int init_flag;                 /* Flag for chemical score initialization */

  float **grid;                  /* Chemical score grids */
  float distance;                /* Longest chemical interaction distance */

} SCORE_CHEMICAL;


typedef struct score_energy_struct
{
  int flag;                      /* Flag for energy score use */
  int init_flag;                 /* Flag for energy score initialization */
  int decomp_flag;               /* Flag to decompose energy by atom */

  float *avdw, *bvdw, *es;       /* VDW and electrostatic potential grids */

  float distance;                /* Cutoff distance for NB interaction */

  int distance_dielectric;       /* Flag for distance dependent dielectric */
  float dielectric_factor;       /* Factor to multiply dielectric function */

  int repulsive_exponent;        /* Exponent of repulsive LJ-pot'l term */
  int attractive_exponent;       /* Exponent of attractive LJ-pot'l term */
  int atom_model;                /* Flag for all atom or united atom model */

  float scale_electro;           /* Scaling factor for electrostatic term */
  float scale_vdw;               /* Scaling factor for Van der Waals term */

  int vdw_init_flag;             /* Flag for vdw initialization */
  float *vdwA;                   /* Array of VDW repulsive terms */
  float *vdwB;                   /* Array of VDW repulsive terms */

} SCORE_ENERGY;

typedef struct score_rmsd_struct
{
  int flag;                      /* Flag for rmsd score use */
  int init_flag;                 /* Flag for rmsd score initialization */

} SCORE_RMSD;


/* ///////////////////// Score Routines /////////////////////////// */

void make_receptor_grid
(
  SCORE_GRID          *grid,
  SCORE_ENERGY        *energy,
  LABEL               *label
);

void free_receptor_grid (SCORE_GRID *grid);

int get_grid_index
(
  SCORE_GRID    *grid,
  XYZ           coord
);

int get_grid_coordinate
(
  SCORE_GRID    *grid,
  XYZ           coord,
  int           grid_coord[3]
);

void calc_inter_score_cont
(
  SCORE_GRID    *grid,
  void          *score,
  float         distance_cutoff,
  void          calc_inter_score
                  (SCORE_GRID *, void *, LABEL *, MOLECULE *,
                  int, int, SCORE_PART *),
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
);

int check_bump
(
  SCORE_GRID *,
  SCORE_BUMP *,
  LABEL *,
  MOLECULE *
);

void calc_inter_contact
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void calc_inter_contact_grid
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void calc_inter_contact_cont
(
  SCORE_GRID *,
  SCORE_CONTACT *,
  LABEL *,
  MOLECULE *,
  int,
  int,
  SCORE_PART *
);

void calc_pairwise_contact
(
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *origin,
  MOLECULE      *target,
  int           origin_atom,
  int           target_atom,
  float         *score
);

void calc_inter_energy
(
  SCORE_GRID *,
  SCORE_ENERGY *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void initialize_vdw_energy
(
  SCORE_ENERGY *energy,
  LABEL_VDW *vdw
);

void free_vdw_energy (SCORE_ENERGY *energy);

void calc_inter_energy_grid
(
  SCORE_GRID *,
  SCORE_ENERGY *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void calc_inter_energy_cont
(
  SCORE_GRID *,
  SCORE_ENERGY *,
  LABEL *,
  MOLECULE *,
  int,
  int,
  SCORE_PART *
);

void calc_pairwise_energy
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *origin,
  MOLECULE      *target,
  int           origin_atom,
  int           target_atom,
  float         *vdwA,
  float         *vdwB,
  float         *electro
);

void calc_inter_chemical
(
  SCORE_GRID *,
```

USF LIBRARY

```
SCORE_
SCORE_
LABEL
MOLECU
int
SCORE_
};

void cal
{
  SCORE_
  SCORE_
  SCORE_
  LABEL
  MOLECU
  int
  SCORE_
};

void ca
{
  SCORE
  SCORE
  LABEL
  MOLEC
  int
  int
  SCORE
};

void c
{
  SCOR
  MOLE
  int
  SCOR
};

float
{
  MOL
  MOL
};

float
{
  MO
  MO
};

floa
{
  MO
  MO
  in
};

voi
{
};
```

```
  SCORE_ENERGY *,
  SCORE_CHEMICAL *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void calc_inter_chemical_grid
(
  SCORE_GRID *,
  SCORE_ENERGY *,
  SCORE_CHEMICAL *,
  LABEL *,
  MOLECULE *,
  int,
  SCORE_PART *
);

void calc_inter_chemical_cont
(
  SCORE_GRID *,
  SCORE_ENERGY *,
  LABEL *,
  MOLECULE *,
  int,
  int,
  SCORE_PART *
);

void calc_inter_rmsd
(
  SCORE_GRID *,
  MOLECULE *,
  int,
  SCORE_PART *
);

float calc_rmsd
(
  MOLECULE       *mol_ori,
  MOLECULE       *mol_ref
);

float calc_layer_rmsd
(
  MOLECULE       *mol_ori,
  MOLECULE       *mol_ref
);

float calc_segment_rmsd
(
  MOLECULE       *mol_ori,
  MOLECULE       *mol_ref,
  int            segment
);


void calc_intra_contact
(
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
);

void calc_intra_energy
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
);

void calc_intra_chemical
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
);

void sum_contact
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
);

void sum_energy
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
);

void sum_chemical
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
);

void sum_rmsd
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
);


##############################################################################
################################### SCORE.C ##################################
##############################################################################
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Written by Todd Ewing
10/95
*/

#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
```

```
#include "label.h"
#include "score.h"
#include "io_grid.h"
#include "io_receptor.h"
#include "vector.h"

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

Routine to read in receptor atoms and partition them into a
3-D grid for rapid evaluation of continuous scores.

2/97 te

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

void make_receptor_grid
(
  SCORE_GRID             *grid,
  SCORE_ENERGY           *energy,
  LABEL                  *label
)
{
  FILE *receptor_file;
  XYZ span;
  int i, j, index, grid_coord[3];
  SLINT **grid_current = NULL;
  SLINT **grid_previous = NULL;

  grid->init_flag = TRUE;

/*
* Open and read receptor atom file
* 9/96 te
*/
  receptor_file = efopen (grid->receptor_file_name, "r", global.outfile);

  read_receptor
  (
    energy,
    label,
    &grid->receptor,
    grid->receptor_file_name,
    receptor_file,
    label->vdw.flag || label->chemical.flag,
    label->chemical.flag,
    label->vdw.flag
  );

/*
* Determine size of box enclosing receptor
* 9/96 te
*/
  for (j = 0; j < 3; j++)
  {
    grid->origin[j] = grid->receptor.coord[0][j];
    span[j] = grid->receptor.coord[0][j];
  }

  for (i = 1; i < grid->receptor.total.atoms; i++)
    for (j = 0; j < 3; j++)
    {
      if (grid->receptor.coord[i][j] < grid->origin[j])
        grid->origin[j] = grid->receptor.coord[i][j];

      if (grid->receptor.coord[i][j] > span[j])
        span[j] = grid->receptor.coord[i][j];
    }

  grid->size = 1;

  for (j = 0; j < 3; j++)
  {
    grid->span[j] = NINT ((span[j] - grid->origin[j]) / grid->spacing) + 1;
    grid->size *= grid->span[j];
  }

  ecalloc
  (
    (void **) &grid->atom,
    grid->size,
    sizeof (SLINT *),
    "receptor atom grid",
    global.outfile
  );

  ecalloc
  (
    (void **) &grid_current,
    grid->size,
    sizeof (SLINT *),
    "receptor atom ptr grid",
    global.outfile
  );

  ecalloc
  (
    (void **) &grid_previous,
    grid->size,
    sizeof (SLINT *),
    "receptor atom ptr grid",
    global.outfile
  );

  for (i = 0; i < grid->receptor.total.atoms; i++)
  {
    for (j = 0; j < 3; j++)
      grid_coord[j] = NINT ((grid->receptor.coord[i][j] - grid->origin[j])
        / grid->spacing);

    index =
      grid->span[0] * grid->span[1] * grid_coord[2] +
      grid->span[0] * grid_coord[1] +
      grid_coord[0];

    ecalloc
    (
      (void **) &grid_current[index],
      1,
      sizeof (SLINT),
      "next atom in receptor atom grid",
      global.outfile
    );

    grid_current[index]->value = i;

    if (grid->atom[index])
      grid_previous[index]->next = grid_current[index];

    else
```

```
      grid->atom[index] = grid_current[index];

    grid_previous[index] = grid_current[index];
    grid_current[index] = NULL;
  }

  efree ((void **) &grid_current);
  efree ((void **) &grid_previous);
}


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

Routine to free receptor grid

5/97 te

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

void free_receptor_grid (SCORE_GRID *grid)
{
  int i;
  SLINT *previous = NULL;
  SLINT *current = NULL;

  free_molecule (&grid->receptor);

  if (grid->atom)
  {
    for (i = 0; i < grid->size; i++)
    {
      for
      (
        previous = grid->atom[i];
        previous != NULL;
        previous = current
      )
      {
        current = previous->next;
        efree ((void **) &previous);
      }
    }

    efree ((void **) &grid->atom);
  }
}


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

Routine to compute the 1D array index given 3D coordinates.

Return values:
        POSITIVE INTEGER        grid index
        NEITHER (-1)            coordinates outside grid

2/97 te

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

int get_grid_index
(
  SCORE_GRID    *grid,
  XYZ           coord
)
{
  int grid_coord[3];

  if (get_grid_coordinate (grid, coord, grid_coord))
    return
      grid->span[0] * grid->span[1] * grid_coord[2] +
      grid->span[0] * grid_coord[1] +
      grid_coord[0];

  else
    return NEITHER;
}

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

Routine to compute the 3D integer grid coordinates given
3D real coordinates.

Return values:
        TRUE            coordinates inside grid
        FALSE           coordinates outside grid

2/97 te

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

int get_grid_coordinate
(
  SCORE_GRID    *grid,
  XYZ           coord,
  int           grid_coord[3]
)
{
  int i;
  int inside_flag = TRUE;

  for (i = 0; i < 3; i++)
  {
    grid_coord[i] = NINT ((coord[i] - grid->origin[i]) / grid->spacing);

    if
    (
      (grid_coord[i] < 0) ||
      (grid_coord[i] > grid->span[i] - 1)
    )
      inside_flag = FALSE;
  }

  return inside_flag;
}


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

Routine to identify all receptor atoms near the ligand atom
and compute a continuous intermolecular score.

2/97 te

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

void calc_inter_score_cont
(
  SCORE_GRID    *grid,
  void          *score,
  float         distance_cutoff,
  void          calc_inter_score
                  (SCORE_GRID *, void *, LABEL *, MOLECULE *,
                  int, int, SCORE_PART *),
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  int i, j, k;
  int ilo, jlo, klo;
  int ihi, jhi, khi;
  int grid_cutoff;
  int grid_coord[3];
  int index;
  int rec_atom;
  SLINT *ptr;

  get_grid_coordinate (grid, molecule->coord[atom], grid_coord);
  grid_cutoff = (int) (distance_cutoff / grid->spacing) + 1;

  ilo = MAX (0, (grid_coord[0] - grid_cutoff));
  jlo = MAX (0, (grid_coord[1] - grid_cutoff));
  klo = MAX (0, (grid_coord[2] - grid_cutoff));
  ihi = MIN (grid->span[0], (grid_coord[0] + grid_cutoff + 1));
  jhi = MIN (grid->span[1], (grid_coord[1] + grid_cutoff + 1));
  khi = MIN (grid->span[2], (grid_coord[2] + grid_cutoff + 1));

  for (i = ilo; i < ihi; i++)
    for (j = jlo; j < jhi; j++)
      for (k = klo; k < khi; k++)
      {
        index =
          grid->span[0] * grid->span[1] * k +
          grid->span[0] * j + i;

        for (ptr = grid->atom[index]; ptr; ptr = ptr->next)
        {
          rec_atom = ptr->value;

          calc_inter_score
          (
            grid,
            score,
            label,
            molecule,
            atom,
            rec_atom,
            inter
          );
        }
      }
}


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

Routine to evaluate the number of intermolecular bumps between
a ligand and receptor

2/97 te

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

int check_bump
(
  SCORE_GRID *grid,
  SCORE_BUMP *bump,
  LABEL *label,
  MOLECULE *molecule
)
{
  int atom;
  int segment;
  int index;

  if (bump->grid == NULL)
    exit (fprintf (global.outfile, "ERROR check_bump: No bump grid loaded\n"));

  for (atom = molecule->score.bumpcount = 0;
    (molecule->score.bumpcount <= bump->maximum) &&
    (atom < molecule->total.atoms); atom++)
  {
    if
    (
      ((segment = molecule->atom[atom].segment_id) != NEITHER) &&
      (segment < molecule->total.segments) &&
      (molecule->segment[segment].active_flag != TRUE)
    )
      continue;

    if (molecule->atom[atom].heavy_flag == TRUE)
    {
      index = get_grid_index (grid, molecule->coord[atom]);

      if (index != NEITHER)
      {
        if (grid->version < 3.99)
        {
          if (bump->grid[index] == 'T')
            molecule->score.bumpcount++;
        }

        else
        {
          if (label->vdw.member[molecule->atom[atom].vdw_id].bump_id >=
            bump->grid[index])
            molecule->score.bumpcount++;
        }
      }
    }
  }

  return molecule->score.bumpcount;
}


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

Routine to compute the intermolecular contact score between
a ligand atom and the receptor.

2/97 te
```

```
///////////////////////////////////////////////////////////////// */

void calc_inter_contact
(
  SCORE_GRID    *grid,
  SCORE_BUMP    *bump,
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  if (molecule->atom[atom].heavy_flag == TRUE)
  {
    if (grid->flag)
      calc_inter_contact_grid
        (grid, bump, contact, label, molecule, atom, inter);

    else
      calc_inter_score_cont
      (
        grid,
        (void *) contact,
        contact->distance,
        (void (*)()) calc_inter_contact_cont,
        label,
        molecule,
        atom,
        inter
      );
  }
}

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

Routine to compute the intermolecular contact score using a
precomputed grid.

2/97 te

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

void calc_inter_contact_grid
(
  SCORE_GRID    *grid,
  SCORE_BUMP    *bump,
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  int index;

  index = get_grid_index (grid, molecule->coord[atom]);

  if (index != NEITHER)
  {
    if (label->vdw.member[molecule->atom[atom].vdw_id].bump_id >=
      bump->grid[index])
      inter->total += contact->clash_penalty;

    else
      inter->total += (float) contact->grid[index];
  }
}

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

Routine to compute the intermolecular contact score in a continuous
fashion given a ligand atom and a receptor atom.

2/97 te

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

void calc_inter_contact_cont
(
  SCORE_GRID    *grid,
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  int           rec_atom,
  SCORE_PART    *inter
)
{
  float score;

  if (grid->receptor.atom[rec_atom].heavy_flag == TRUE)
  {
    calc_pairwise_contact
    (
      contact,
      label,
      molecule,
      &grid->receptor,
      atom,
      rec_atom,
      &score
    );

    inter->total += score;
  }
}

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

Routine to compute the intramolecular contact score
between two ligand atoms.

2/97 te

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

void calc_intra_contact
(
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
)
{
  float score;

  calc_pairwise_contact
  (
    contact,
    label,
    molecule,
    molecule,
    atomi,
    atomj,
    &score
  );

  intra->total += score;
}

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

Routine to compute the contact score between any two atoms.

2/97 te

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

void calc_pairwise_contact
(
  SCORE_CONTACT *contact,
  LABEL         *label,
  MOLECULE      *origin,
  MOLECULE      *target,
  int           origin_atom,
  int           target_atom,
  float         *score
)
{
  float reference;
  float sq_distance;

  if
  (
    origin->atom[origin_atom].heavy_flag &&
    target->atom[target_atom].heavy_flag &&
    ((sq_distance = square_distance
      (origin->coord[origin_atom], target->coord[target_atom]))
        < SQR (contact->distance))
  )
  {
    reference = contact->clash_overlap *
      (label->vdw.member[origin->atom[origin_atom].vdw_id].radius +
      label->vdw.member[target->atom[target_atom].vdw_id].radius);

    if (sq_distance < SQR (reference))
      *score = contact->clash_penalty;

    else
      *score = -1;
  }

  else
    *score = 0;
}

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

Routine to add contact score components.

2/97 te

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

void sum_contact
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
)
{
  sum->total += increment->total;
}


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

Routine to compute the intermolecular energy score between
a ligand atom and the receptor.

2/97 te

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

void calc_inter_energy
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  if (!energy->vdw_init_flag)
    initialize_vdw_energy (energy, &label->vdw);

  if (label->vdw.member[molecule->atom[atom].vdw_id].well_depth != 0.0)
  {
    if (grid->flag)
      calc_inter_energy_grid (grid, energy, label, molecule, atom, inter);

    else
      calc_inter_score_cont
      (
        grid,
        (void *) energy,
        energy->distance,
        (void (*)()) calc_inter_energy_cont,
        label,
        molecule,
        atom,
        inter
      );
  }
}

/* /////////////////////////////////////////////////////////////////
```

```
Routine to initialize vdw parameters.

2/97 te

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

void initialize_vdw_energy
(
  SCORE_ENERGY  *energy,
  LABEL_VDW     *label_vdw
)
{
  int i;

  if (!label_vdw->init_flag)
    get_vdw_labels (label_vdw);

  ecalloc
  (
    (void **) &energy->vdwA,
    label_vdw->total,
    sizeof (float),
    "energy vdwA terms",
    global.outfile
  );

  ecalloc
  (
    (void **) &energy->vdwB,
    label_vdw->total,
    sizeof (float),
    "energy vdwB terms",
    global.outfile
  );

  for (i = 0; i < label_vdw->total; i++)
  {
    energy->vdwA[i] = sqrt (label_vdw->member[i].well_depth *
      (float) energy->attractive_exponent /
      (float) (energy->repulsive_exponent - energy->attractive_exponent) *
      pow (2 * label_vdw->member[i].radius, energy->repulsive_exponent));

    energy->vdwB[i] = sqrt (label_vdw->member[i].well_depth *
      (float) energy->repulsive_exponent /
      (float) (energy->repulsive_exponent - energy->attractive_exponent) *
      pow (2 * label_vdw->member[i].radius, energy->attractive_exponent));
  }

  energy->dielectric_factor = 332.0 / energy->dielectric_factor;
  energy->vdw_init_flag = TRUE;
}


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

void free_vdw_energy (SCORE_ENERGY *energy)
{
  efree ((void **) &energy->vdwA);
  efree ((void **) &energy->vdwB);
}


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

Routine to compute the intermolecular energy score using a
precomputed grid.

2/97 te

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

void calc_inter_energy_grid
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  int index[8];
  int out_flag;
  XYZ cube_coord;

  void get_index_ (XYZ, XYZ, float *, int *, int *, int *, float *);
  float get_value_ (float *, XYZ, int *);

  if (label->vdw.member[molecule->atom[atom].vdw_id].well_depth != 0.0)
  {
    get_index_
    (
      molecule->coord[atom],
      grid->origin,
      &grid->spacing,
      grid->span,
      &out_flag,
      index,
      cube_coord
    );

    if (out_flag == 0)
    {
      inter->vdw +=
        (
          energy->vdwA[molecule->atom[atom].vdw_id] *
            get_value_(energy->avdw, cube_coord, index) -
          energy->vdwB[molecule->atom[atom].vdw_id] *
            get_value_(energy->bvdw, cube_coord, index)
        ) * energy->scale_vdw;

      inter->electro +=
        molecule->atom[atom].charge *
        get_value_(energy->es, cube_coord, index) *
        energy->scale_electro;

      inter->total = inter->vdw + inter->electro;
    }
  }
}


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

Routine to compute the intermolecular energy score in a continuous
fashion given a ligand atom and a receptor atom.

2/97 te

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

void calc_inter_energy_cont
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  int           rec_atom,
  SCORE_PART    *inter
)
{
  float vdwA, vdwB, electro;

  if (label->vdw.member[grid->receptor.atom[rec_atom].vdw_id].well_depth != 0.0)
  {
    calc_pairwise_energy
    (
      energy,
      label,
      molecule,
      &grid->receptor,
      atom,
      rec_atom,
      &vdwA,
      &vdwB,
      &electro
    );

    inter->vdw += vdwA - vdwB;
    inter->electro += electro;
    inter->total = inter->vdw + inter->electro;
  }
}


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

Routine to compute the intramolecular energy score
between two ligand atoms.

2/97 te

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

void calc_intra_energy
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
)
{
  float vdwA, vdwB, electro;

  calc_pairwise_energy
  (
    energy,
    label,
    molecule,
    molecule,
    atomi,
    atomj,
    &vdwA,
    &vdwB,
    &electro
  );

  intra->vdw += vdwA - vdwB;
  intra->electro += electro;
  intra->total = intra->vdw + intra->electro;

/*
  fprintf (global.outfile, "%s %s %f\n",
    molecule->atom[atomi].name,
    molecule->atom[atomj].name,
    vdwA - vdwB + electro);
*/
}


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

Routine to compute the energy score between any two atoms.

2/97 te

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

void calc_pairwise_energy
(
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *origin,
  MOLECULE      *target,
  int           origin_atom,
  int           target_atom,
  float         *vdwA,
  float         *vdwB,
  float         *electro
)
{
  int power_exponent;
  float power_distance;
  float distance;
  int square_flag = TRUE;

  if (!energy->vdw_init_flag)
    initialize_vdw_energy (energy, &label->vdw);

  if
  (
    (label->vdw.member[origin->atom[origin_atom].vdw_id].well_depth != 0.0) &&
    (label->vdw.member[target->atom[target_atom].vdw_id].well_depth != 0.0) &&
    ((distance = square_distance
      (origin->coord[origin_atom], target->coord[target_atom]))
      <= SQR (energy->distance))
  )
  {
    distance = 1.0 / MAX (distance, DISTANCE_MIN);

    if (energy->repulsive_exponent % 2)
    {
      power_exponent = energy->repulsive_exponent;
      distance = sqrt (distance);
      square_flag = FALSE;
```

```
    }

    else
      power_exponent = energy->repulsive_exponent / 2;

    POWER (distance, power_exponent, power_distance);

    *vdwA =
      energy->vdwA[origin->atom[origin_atom].vdw_id] *
      energy->vdwA[target->atom[target_atom].vdw_id] *
      power_distance;

    if (square_flag == TRUE)
    {
      if (energy->repulsive_exponent % 2)
      {
        power_exponent = energy->attractive_exponent;
        distance = sqrt (distance);
        square_flag = FALSE;
      }

      else
        power_exponent = energy->attractive_exponent / 2;
    }

    else
      power_exponent = energy->attractive_exponent;

    POWER (distance, power_exponent, power_distance);

    *vdwB =
      energy->vdwB[origin->atom[origin_atom].vdw_id] *
      energy->vdwB[target->atom[target_atom].vdw_id] *
      power_distance;

    *electro =
      energy->dielectric_factor *
      origin->atom[origin_atom].charge *
      target->atom[target_atom].charge *
      (energy->distance_dielectric == TRUE
          ? (square_flag == TRUE ? distance : SQR (distance))
          : (square_flag == TRUE ? sqrt (distance) : distance));
  }

  else
    *vdwA = *vdwB = *electro = 0;
}

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

Routine to add energy score components.

2/97 te

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

void sum_energy
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
)
{
  sum->vdw += increment->vdw;
  sum->electro += increment->electro;
  sum->total = sum->vdw + sum->electro;
}


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

Routine to compute the intermolecular chemical score between
a ligand atom and the receptor.

2/97 te

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

void calc_inter_chemical
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  SCORE_CHEMICAL *chemical,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  if (!energy->vdw_init_flag)
    initialize_vdw_energy (energy, &label->vdw);

  if (label->chemical.score_table == NULL)
    get_table
    (
      &label->chemical,
      label->chemical.score_file_name,
      &label->chemical.score_table
    );

  if (label->vdw.member[molecule->atom[atom].vdw_id].well_depth != 0.0)
  {
    if (grid->flag)
      calc_inter_chemical_grid
        (grid, energy, chemical, label, molecule, atom, inter);

    else
      calc_inter_score_cont
      (
        grid,
        (void *) energy,
        energy->distance,
        (void (*)()) calc_inter_chemical_cont,
        label,
        molecule,
        atom,
        inter
      );
  }
}

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

Routine to compute the intermolecular chemical score using a
precomputed grid.

2/97 te
```

```
/////////////////////////////////////////////////////////////////// */

void calc_inter_chemical_grid
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  SCORE_CHEMICAL *chemical,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  int i;
  int index[8];
  int out_flag;
  XYZ cube_coord;
  float vdw, electro;

  void get_index_ (XYZ, XYZ, float *, int *, int *, int *, float *);
  float get_value_ (float *, XYZ, int *);

  if (label->vdw.member[molecule->atom[atom].vdw_id].well_depth != 0.0)
  {
    get_index_
    (
      molecule->coord[atom],
      grid->origin,
      &grid->spacing,
      grid->span,
      &out_flag,
      index,
      cube_coord
    );

    if (out_flag == 0)
    {
      vdw =
        energy->vdwA[molecule->atom[atom].vdw_id] *
        get_value_(energy->avdw, cube_coord, index);

      for (i = 0; i < label->chemical.total; i++)
      {
        if (chemical->grid[i] != NULL)
          vdw -=
            label->chemical.score_table[molecule->atom[atom].chem_id][i] *
            energy->vdwB[molecule->atom[atom].vdw_id] *
            get_value_(chemical->grid[i], cube_coord, index);
      }

      electro =
        molecule->atom[atom].charge *
        get_value_(energy->es, cube_coord, index);
    }

    else
    {
      vdw = 10.0;
      electro = 10.0;
    }

    inter->vdw += vdw *= energy->scale_vdw;
    inter->electro += electro *= energy->scale_electro;
    inter->total = inter->vdw + inter->electro;
  }
}

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

Routine to compute the intermolecular chemical score in a continuous
fashion given a ligand atom and a receptor atom.

2/97 te

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

void calc_inter_chemical_cont
(
  SCORE_GRID    *grid,
  SCORE_ENERGY  *energy,
  LABEL         *label,
  MOLECULE      *molecule,
  int           atom,
  int           rec_atom,
  SCORE_PART    *inter
)
{
  float vdwA, vdwB, electro;

  if (label->vdw.member[grid->receptor.atom[rec_atom].vdw_id].well_depth != 0.0)
  {
    calc_pairwise_energy
    (
      energy,
      label,
      molecule,
      &grid->receptor,
      atom,
      rec_atom,
      &vdwA,
      &vdwB,
      &electro
    );

    inter->vdw += vdwA - vdwB *
      label->chemical.score_table
        [molecule->atom[atom].chem_id]
        [grid->receptor.atom[rec_atom].chem_id];

    inter->electro += electro;
    inter->total = inter->vdw + inter->electro;
  }
}

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

Routine to compute the intramolecular chemical score
between two ligand atoms.

2/97 te

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

void calc_intra_chemical
(
  SCORE_ENERGY  *energy,
```

```
  LABEL         *label,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
)
{
  float vdwA, vdwB, electro;

  calc_pairwise_energy
  (
    energy,
    label,
    molecule,
    molecule,
    atomi, atomj,
    &vdwA,
    &vdwB,
    &electro
  );

  intra->vdw += vdwA - vdwB *
    label->chemical.score_table
      [molecule->atom[atomi].chem_id]
      [molecule->atom[atomj].chem_id];

  intra->electro += electro;
  intra->total = intra->vdw + intra->electro;
}


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

Routine to add chemical score components.

2/97 te

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

void sum_chemical
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
)
{
  sum->vdw += increment->vdw;
  sum->electro += increment->electro;
  sum->total = sum->vdw + sum->electro;
}


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

Routine to compute the intermolecular rmsd score between
a ligand atom and the receptor.

2/97 te

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

void calc_inter_rmsd
(
  SCORE_GRID    *grid,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
/*
* Check if input is alright
* 1/97 te
*/
  if (molecule->total.atoms != grid->receptor.total.atoms)
    exit (fprintf (global.outfile,
      "ERROR calc_rmsd_score: molecules have different number of atoms\n"));

  if (molecule->atom[atom].heavy_flag == TRUE)
  {
    inter->vdw += 1;
    inter->electro +=
      square_distance (molecule->coord[atom], grid->receptor.coord[atom]);
  }
}


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

Routine to add rmsd score components.

2/97 te

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

void sum_rmsd
(
  SCORE_PART    *sum,
  SCORE_PART    *increment
)
{
  sum->vdw += increment->vdw;
  sum->electro += increment->electro;

  if (sum->vdw > 0)
    sum->total = sqrt (sum->electro / sum->vdw);

  else
    sum->total = 0;
}


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

Routine to compute the rms deviation between two configurations
of the same molecule.

2/97 te

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

float calc_rmsd
(
  MOLECULE      *mol_ori,
  MOLECULE      *mol_ref
)
{
  int atom;
  int layer_ref;
  int layer_ori;
  int heavy_total = 0;
  float rmsd = 0;

  if (mol_ori->total.atoms != mol_ref->total.atoms)
    exit (fprintf (global.outfile,
      "ERROR calc_rmsd: molecules have different number of atoms\n"));

/*
* If both molecules have an anchor layer, but the anchors are
* different, then set the rmsd arbitrarily high
* 2/97 te
  if ((mol_ori->total.layers > 1) &&
    (mol_ref->total.layers > 1))
  {
    if (mol_ori->layer[0].segment_total != mol_ref->layer[0].segment_total)
      return FLT_MAX;

    for (segment = 0; segment < mol_ref->layer[0].segment_total; segment++)
      if (mol_ori->layer[0].segment[segment] !=
        mol_ref->layer[0].segment[segment])
        return FLT_MAX;
  }
*/

/*
* Loop through atoms
* 2/97 te
*/
  for (atom = 0; atom < mol_ori->total.atoms; atom++)
  {
    if (mol_ori->atom[atom].heavy_flag != TRUE)
      continue;

    if ((mol_ori->total.segments > 0) &&
      (mol_ori->segment[mol_ori->atom[atom].segment_id].active_flag == FALSE))
      continue;

    if ((mol_ref->total.layers > 1) && (mol_ori->total.layers > 1))
    {
      layer_ref = mol_ref->segment[mol_ref->atom[atom].segment_id].layer_id;
      layer_ori = mol_ori->segment[mol_ori->atom[atom].segment_id].layer_id;

      if
      (
        ((layer_ref == NEITHER) || (layer_ori == NEITHER)) &&
        (layer_ref != layer_ori)
      )
        return FLT_MAX;
    }

    rmsd += square_distance (mol_ori->coord[atom], mol_ref->coord[atom]);
    heavy_total++;
  }

  if (heavy_total > 0)
    rmsd = sqrt (rmsd / (float) heavy_total);

  else
    rmsd = 0;

  return rmsd;
}


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

Routine to compute the layer-weighted rms deviation between two configurations
of the same molecule.

3/97 te

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

float calc_layer_rmsd
(
  MOLECULE      *mol_ori,
  MOLECULE      *mol_ref
)
{
  int atom;
  int layer_ref;
  int layer_ori;
  int heavy_total = 0;
  float rmsd = 0;

  if (mol_ori->total.atoms != mol_ref->total.atoms)
    exit (fprintf (global.outfile,
      "ERROR calc_rmsd: molecules have different number of atoms\n"));

/*
* Loop through atoms
* 3/97 te
*/
  for (atom = 0; atom < mol_ori->total.atoms; atom++)
  {
/*
    if (mol_ori->atom[atom].heavy_flag != TRUE)
      continue;
*/

    if ((mol_ori->total.segments > 0) &&
      (mol_ori->segment[mol_ori->atom[atom].segment_id].active_flag == FALSE))
      continue;

    layer_ref = mol_ref->segment[mol_ref->atom[atom].segment_id].layer_id;
    layer_ori = mol_ori->segment[mol_ori->atom[atom].segment_id].layer_id;

    if
    (
      ((layer_ref == NEITHER) || (layer_ori == NEITHER)) &&
      (layer_ref != layer_ori)
    )
      return FLT_MAX;

    rmsd +=
      square_distance (mol_ori->coord[atom], mol_ref->coord[atom]) *
      (layer_ori + 1);
    heavy_total += layer_ori + 1;
  }

  if (heavy_total > 0)
    rmsd = sqrt (rmsd / (float) heavy_total);

  else
    rmsd = 0;

  return rmsd;
```

```
}

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

Routine to compute the rms deviation of a segment from two configurations
of the same molecule.

3/97 te

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

float calc_segment_rmsd
(
  MOLECULE      *mol_ori,
  MOLECULE      *mol_ref,
  int           segment
)
{
  int atom;
  int atom_id;
  int atom_total = 0;
  float rmsd = 0;

  if (mol_ori->total.atoms != mol_ref->total.atoms)
    exit (fprintf (global.outfile,
      "ERROR calc_segment_rmsd: molecules have different number of atoms\n"));

/*
* Loop through atoms
* 2/97 te
*/
  for (atom_id = 0; atom_id < mol_ori->segment[segment].atom_total; atom_id++)
  {
    atom = mol_ori->segment[segment].atom[atom_id];

    if ((atom < 0) || (atom >= mol_ori->total.atoms))
      exit (fprintf (global.outfile,
        "ERROR calc_segment_rmsd: segment contains bad atom\n"));

    rmsd += square_distance (mol_ori->coord[atom], mol_ref->coord[atom]);
    atom_total++;
  }

  if (atom_total > 0)
    rmsd = sqrt (rmsd / (float) atom_total);

  else
    rmsd = 0;

  return rmsd;
}

##########################################################################
################################ SCORE_DOCK.H ############################
##########################################################################
/*                                                                      */
/*                      Copyright UCSF, 1997                            */
/*                                                                      */

/*
Written by Todd Ewing
9/96
*/
#define NONE            0
#define CONTACT         1
#define CHEMICAL        2
#define ENERGY          3
#define RMSD            4
#define SCORE_TOTAL     5


typedef struct score_type_struct
{
  int           flag;                   /* Flag for this type of scoring */
  STRING20      name;                   /* Name of scoring */
  STRING5       abbrev;                 /* Abbreviated name of scoring */
  float         maximum;                /* Maximum score filter */
  float         size_penalty;           /* Size penalty to compare molecules */
  int           minimize;               /* Flag for if minimization performed */
  float         convergence;            /* Convergence criteria for minimizer */
  float         termination;            /* Termination criteria for minimizer */
  FILE_NAME     file_name;              /* Output file name */
  FILE          *file;                  /* Output file stream */
  int           number_written;         /* Number of orientations written */

} SCORE_TYPE;

typedef struct minimize_struct
{
  int           flag;                   /* Flag for minimization */
  int           torsion_flag;           /* Flag for torsion minimization */

  int           iteration;              /* Maximum simplex iterations */
  int           cycle;                  /* Maximum simplex cycles */
  float         cycle_converge;         /* Simplex cycle convergence */
  float         rotation;               /* Initial rotation step size */
  float         translation;            /* Initial translation step size */
  float         torsion;                /* Initial torsion step size */

  int           call_total;             /* Total calls */
  int           call_sub_total;         /* Total calls per molecule */
  int           call_min;               /* Minimum calls per molecule */
  int           call_max;               /* Maximum calls per molecule */
  int           vertex_total;           /* Total vertices */
  int           vertex_min;             /* Minimum vertices per call */
  int           vertex_max;             /* Maximum vertices per call */
  int           cycle_total;            /* Total cycles */
  int           cycle_min;              /* Minimum cycles per call */
  int           cycle_max;              /* Maximum cycles per call */
  int           iteration_total;        /* Total iterations */
  int           iteration_min;          /* Minimum iterations per call */
  int           iteration_max;          /* Maximum iterations per call */
  float         delta_total;            /* Total score change */
  float         delta_min;              /* Minimum score change per call */
  float         delta_max;              /* Maximum score change per call */

} MINIMIZE;

typedef struct near_struct
{
  int           **flag;                 /* Array of nearby atoms */
  int           total;                  /* Total flags */
  char          *name;                  /* Name of molecule described */

} NEAR;

typedef struct score_struct
{
  int           flag;                   /* Flag to use ANY type of scoring */
  int           intra_flag;             /* Flag to get intra-ligand energy */
  int           inter_flag;             /* Flag to get ligand-rec score */

  SCORE_GRID    grid;                   /* Grid scoring related info */
  SCORE_BUMP    bump;                   /* Bump checking info */
  SCORE_CONTACT contact;                /* Contact scoring info */
  SCORE_CHEMICAL chemical;              /* Chemical scoring info */
  SCORE_ENERGY  energy;                 /* Energy scoring info */
  SCORE_RMSD    rmsd;                   /* RMSD scoring info */

  SCORE_TYPE    type[SCORE_TOTAL];      /* Info for each type of scoring */
  int           key[SCORE_TOTAL];       /* Key to scoring types selected */

  MINIMIZE      minimize;               /* Minimization info */

  NEAR          near;                   /* Atom proximity info */
  float         rmsd_override;          /* RMS cutoff to force write */
  float         time;                   /* Time spent getting energy of pair */

} SCORE;

typedef struct list_struct
{
  int lite_flag;                        /* Flag to not store coordinates */
  int total[SCORE_TOTAL];               /* Length of each list */
  int max[SCORE_TOTAL];                 /* Maximum length of each list */
  MOLECULE **member[SCORE_TOTAL];       /* Sorted lists of molecules */

} LIST;


void score_setup        (SCORE *, LABEL *, MOLECULE *);

int get_anchor_score
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_init,
  MOLECULE      *mol_ori,
  MOLECULE      *mol_min,
  LIST          *best_anchors,
  int           orient_id,
  int           write_flag
);

void get_peripheral_score
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_init,
  MOLECULE      *mol_conf,
  MOLECULE      *mol_ori,
  MOLECULE      *mol_min,
  LIST          *best_anchors,
  LIST          *best_orient,
  int           write_flag
);

void minimize_ligand
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_ori,
  MOLECULE      *mol_score,
  int           rigid_flag,
  int           layer_initial,
  int           layer_final
);

float simplex_score (void *simplex, float *vertex);

float calc_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           layer_final
);

void calc_inter_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
);

void calc_intra_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
);

void sum_score
(
  int           score_type,
  SCORE_PART    *sum,
  SCORE_PART    *increment
);

void output_score_info
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  LIST          *list,
  int           number_docked,
  float         time
);

void  allocate_lists    (SCORE *, LIST *, int, int);
void  allocate_list     (LIST *, int);
void  reset_lists       (SCORE *, LIST *);
void  reset_list        (LIST *, int);
void  free_lists        (SCORE *, LIST *);
```

```
void  free_list          (LIST *, int);
void  reallocate_lists   (SCORE *, LIST *);
void  reallocate_list    (LIST *, int);
void  copy_lists         (SCORE *, LIST *, LIST *);
void  copy_list          (LIST *, LIST *, int);
void  inter_lists        (SCORE *, LIST *, MOLECULE *);
void  merge_lists        (SCORE *, LIST *, LIST *);
void  update_list        (LIST *, MOLECULE *);
int   compare_score      (MOLECULE *, MOLECULE *);
void  sort_list          (LIST *, int type);
void  copy_member        (int, MOLECULE *, MOLECULE *);
int   compare_member     (MOLECULE **, MOLECULE **);
int   print_lists        (SCORE *, LIST *, FILE *);
int   save_lists         (SCORE *, LIST *, FILE *);
int   load_lists         (SCORE *, LIST *, FILE *);

void shrink_list
(
  LIST          *full,
  LIST          *shrunk,
  int           type
);

void update_shrunk_list
(
  LIST          *shrunk,
  MOLECULE      *molecule
);

void initialize_near     (NEAR *near, MOLECULE *molecule);
void allocate_near       (NEAR *near);
void free_near           (NEAR *near);

void free_scores         (LABEL *label, SCORE *score);
################################################################################
################################ SCORE_DOCK.C ##################################
################################################################################
/*                                                                            */
/*                        Copyright UCSF, 1997                                */
/*                                                                            */

/*
Written by Todd Ewing
12/96
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "dock.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "score_dock.h"
#include "flex.h"
#include "simplex.h"
#include "io_ligand.h"
#include "io_grid.h"
#include "transform.h"
#include "rank.h"
#include "vector.h"

typedef struct simplex_struct
{
  DOCK          *dock;
  LABEL         *label;
  SCORE         *score;
  MOLECULE      *mol_ref;
  MOLECULE      *mol_ori;
  MOLECULE      *mol_score;
  MOLECULE      mol_min;
  MOLECULE      mol_best;
  LIST          *list;
  int           rigid_flag;
  int           layer_inner;
  int           layer_outer;
  int           cycle;

} SIMPLEX;


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

Subroutine to calculate and optimize the score for a ligand orientation.
The best scoring orientation is updated.

Return values:
        TRUE    molecule orientation was scored
        FALSE   molecule orientation not scored

1/97 te

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

int get_anchor_score
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_init,
  MOLECULE      *mol_ori,
  MOLECULE      *mol_score,
  LIST          *best_anchors,
  int           orient_id,
  int           write_flag
)
{
  int type;                     /* Score type iterator */

/*
* Read/construct scoring grids
* 11/96 te
*/
  if (!orient_id)
  {
    if (score->inter_flag && !score->grid.init_flag)
    {
      if (score->grid.flag)
        read_grids
        (
          &score->grid,
          &score->bump,
          &score->contact,
          &score->chemical,
          &score->energy,
          &label->chemical
        );

      else
        make_receptor_grid
        (
          &score->grid,
          &score->energy,
          label
        );
    }
  }

  else
    reset_score (&mol_ori->score);

/*
* Evaluate whether anchor orientation bumps with receptor
* 12/96 te
*/
  if
  (
    (score->inter_flag == TRUE) &&
    (score->bump.flag == TRUE) &&
    (check_bump (&score->grid, &score->bump, label, mol_ori) >
      score->bump.maximum)
  )
    return FALSE;

  if (mol_ori->transform.flag)
    mol_ori->transform.rmsd =
      calc_rmsd (mol_ori, mol_init);

  copy_transform (mol_score, mol_ori);
  copy_score (&mol_score->score, &mol_ori->score);
  copy_coords (mol_score, mol_ori);
  copy_torsions (mol_score, mol_ori);
  copy_segments (mol_score, mol_ori);
  copy_layers (mol_score, mol_ori);

/*
* Loop through all scoring types
* 11/96 te
*/
  for (type = 0; type < SCORE_TOTAL; type++)
  {
    if (!score->type[type].flag)
      continue;

    mol_score->score.type = type;

/*
*   Either minimize this orientation or just score it
*   11/96 te
*/
    if (score->type[type].minimize && label->flex.minimize_anchor_flag)
    {
      minimize_ligand
        (dock, label, score, mol_ref, mol_ori, mol_score, TRUE, 0, 0);
      mol_score->transform.rmsd =
        calc_rmsd (mol_score, mol_init);
    }

    else
      calc_score (label, score, mol_score, 0);

/*
*   Update the list of best anchors
*   11/96 te
*/
    update_list (best_anchors, mol_score);

/*
*   Write out the orientation if flagged, and if:
*
*   1. Ligand has moved AND rmsd is within override, OR
*   2. No scoring is performed, OR
*   3. The score is below maximum cutoff
*
*   3/96 te
*/
    if
    (
      write_flag &&
      (
        ((mol_score->transform.flag) &&
          (mol_score->transform.rmsd <=
            score->rmsd_override)) ||
        (mol_score->score.total <= score->type[type].maximum)
      )
    )
    {
      write_ligand
      (
        dock,
        score,
        mol_score,
        score->type[type].file_name,
        score->type[type].file
      );
    }

/*
*   Reset coordinates previous to minimization
*   6/96 te
*/
    if (score->type[type].minimize)
    {
      copy_transform (mol_score, mol_ori);
      copy_torsions (mol_score, mol_ori);
      copy_coords (mol_score, mol_ori);
    }

/*
    print_list (score, best_anchors, global.outfile);
*/
  }

  return TRUE;
}


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

void get_peripheral_score
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_init,
  MOLECULE      *mol_conf,
```

```
  MOLECULE      *mol_ori,
  MOLECULE      *mol_score,
  LIST          *best_anchors,
  LIST          *best_orients,
  int           write_flag
)
{
  int type;                     /* Score type iterator */
  int seed;                     /* Seed conformation iterator */
  int layer;                    /* Segment layer iterator */
  int segment_id;               /* Segment iterator */
  int segment;                  /* Segment id */
  static int allocate_flag = TRUE; /* Flag for whether lists allocated */
  static LIST seed_conform;     /* List of seed conformations */
  static LIST best_conform;     /* List of best conformations */

/*
* Initialize conformation storage lists
* 12/96 te
*/
  if (allocate_flag == TRUE)
  {
    allocate_lists (score, &seed_conform, label->flex.periph_seeds, FALSE);
    allocate_flag = FALSE;
  }

  else
    reset_lists (score, &seed_conform);

  copy_lists (score, &best_conform, best_anchors);

/*
* Loop through all scoring types
* 12/96 te
*/
  for (type = 0; type < SCORE_TOTAL; type++)
  {
    if (!score->type[type].flag)
      continue;

/*
*   Loop through each shell of flexible segments
*   3/97 te
*/
    for
    (
      layer = 1;
      (layer < mol_conf->total.layers) &&
        (best_conform.total[type] > 0);
      layer++
    )
    {
/*
*     Loop through all segments in current layer
*     3/97 te
*/
      for
      (
        segment_id = 0;
        (segment_id < mol_conf->layer[layer].segment_total) &&
          (best_conform.total[type] > 0);
        segment_id++
      )
      {
/*
*       Identify the most different best conforms as seeds
*       3/97 te
*/
        shrink_list
        (
          &best_conform,
          &seed_conform,
          type
        );

        if (global.output_volume == 'v')
        {
          fprintf (global.outfile,
            "Starting layer %d, segment %d, with %d seed%s\n",
            layer + 1, segment_id + 1, seed_conform.total[type],
            seed_conform.total[type] > 1 ? "s" : "");

          print_lists (score, &seed_conform, global.outfile);
          fflush (global.outfile);

/*
          write_topscorers
          (
            dock,
            score,
            &seed_conform,
            mol_ref,
            mol_ori
          );
*/
        }

/*
*       Initialize seed conformations as best scorers from previous level
*       11/96 te
*/
        segment = mol_conf->layer[layer].segment[segment_id];

        best_conform.total[type] =
          seed_conform.total[type] * mol_conf->segment[segment].conform_total;
        reallocate_list (&best_conform, type);
        reset_list (&best_conform, type);

/*
*       Loop through all seed conformations
*       3/97 te
*/
        for (seed = 0; seed < seed_conform.total[type]; seed++)
        {
            copy_member (FALSE, mol_ori, seed_conform.member[type][seed]);
            mol_ori->layer[layer].active_flag = TRUE;
            mol_ori->segment[segment].active_flag = TRUE;
            mol_ori->segment[segment].min_flag = TRUE;
            mol_ori->segment[segment].conform_count = 0;

/*
*           Drive through conformations of this segment
*           3/97 te
*/
            while
            (
              get_segment_conformation
              (
                label,
                score,
                mol_ori,
                layer,
                segment_id
              ) != EOF
            )
            {
              copy_molecule (mol_score, mol_ori);

/*
*             Either minimize this conformation or just score it
*             If minimize, allow rigid+outer layer OR outer two layers
*             2/97 te
*/
              if (score->type[type].minimize)
                minimize_ligand
                  (dock, label, score, mol_ref, mol_ori, mol_score,
                  label->flex.reminimize_anchor_flag,
                  MAX (0, layer - label->flex.reminimize_layers),
                  layer);

              else
                calc_score
                  (label, score, mol_score, layer);

              if (global.output_volume == 'v')
                mol_score->transform.rmsd =
                  calc_rmsd (mol_score, mol_init);

              mol_score->segment[segment].min_flag = FALSE;
              update_list (&best_conform, mol_score);

            } /* End of conformer loop */
        } /* End of seed loop */

        if (global.output_volume == 'v')
        {
          fprintf (global.outfile,
            "Ending layer %d, segment %d, with %d config%s\n",
            layer + 1, segment_id + 1, best_conform.total[type],
            best_conform.total[type] > 1 ? "s" : "");

          print_lists (score, &best_conform, global.outfile);
        }
      } /* End of segment count loop */
    } /* End of layer loop */

/*
*   Process the top scoring configurations
*   2/97 te
*/
    if (best_conform.total[type] > 0)
    {
      shrink_list
      (
        &best_conform,
        &seed_conform,
        type
      );

/*
*     Re-minimize the top scoring configurations
*     2/97 te
*/
      if
      (
        score->type[type].minimize &&
        label->flex.reminimize_ligand_flag &&
        (mol_ori->total.layers > 2)
      )
      {
        for (seed = 0; seed < seed_conform.total[type]; seed++)
        {
          copy_molecule (mol_score, seed_conform.member[type][seed]);

          for
          (
            segment_id = 0;
            segment_id < mol_score->total.segments;
            segment_id++
          )
            mol_score->segment[segment_id].min_flag = TRUE;

          minimize_ligand
            (dock, label, score, mol_ref, seed_conform.member[type][seed],
            mol_score, TRUE, 0, mol_score->total.layers - 1);
          mol_score->transform.rmsd = calc_rmsd (mol_score, mol_init);

          copy_molecule (seed_conform.member[type][seed], mol_score);
        }

        sort_list (&seed_conform, type);

        if (global.output_volume == 'v')
        {
          fprintf (global.outfile, "Reminimized configurations\n");
          print_lists (score, &seed_conform, global.outfile);
        }
      }

      merge_lists (score, best_orients, &seed_conform);

/*
*     Write out the best configurations to file, if requested
*     2/97 te
*/
      if (write_flag)
      {
        for (seed = 0; seed < seed_conform.total[type]; seed++)
        {
          if
          (
            ((seed_conform.member[type][seed]->transform.flag) &&
              (seed_conform.member[type][seed]->transform.rmsd <=
                score->rmsd_override)) ||
            (seed_conform.member[type][seed]->score.total <=
              score->type[type].maximum)
          )
          {
            write_ligand
            (
              dock,
              score,
              seed_conform.member[type][seed],
              score->type[type].file_name,
              score->type[type].file
            );
          }
        } /* End of seed loop */
```

```
      } /* End of write IF */
    } /* End of process IF */
  } /* End of score type loop */
}


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

void minimize_ligand
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *mol_ref,
  MOLECULE      *mol_ori,
  MOLECULE      *mol_score,
  int           rigid_flag,
  int           layer_inner,
  int           layer_outer
)
{
  int i, j;                     /* Iteration variable */
  int torsion;                  /* Torsion id */
  int segment;                  /* Segment id */
  int layer;                    /* Layer id */
  int neighbor;                 /* Segment neighbor id */
  int flex;                     /* Flex label id */
  SIMPLEX simplex = {0};        /* Simplex data */
  int iteration_total;          /* Number of simplex iterations */
  int vertex_total = 0;         /* Total simplex vertices */
  float *vertex = NULL;         /* Simplex vertices */
  float delta_cycle;            /* Change in score during a cycle */
  float delta_total;            /* Total change in score */
  float distance;               /* Simplex vertex distance */

/*
  if ((score->intra_flag) && (label->flex.minimize_flag == TRUE))
*/
  if (label->flex.minimize_flag == TRUE)
  {
    if
    (
      (layer_inner < 0) ||
      (layer_inner > layer_outer) ||
      (layer_outer >= mol_score->total.layers)
    )
      exit (fprintf (global.outfile,
        "ERROR minimize_ligand: molecule layer selection inappropriate\n"));

/*
*   Identify inner neighbors of minimizing segments
*   4/97 te
*/
    for (layer = layer_outer; layer >= layer_inner; layer--)
    {
      for (i = 0; i < mol_score->layer[layer].segment_total; i++)
      {
        segment = mol_score->layer[layer].segment[i];

        if (mol_score->segment[segment].min_flag)
        {
          if
          (
            ((torsion = mol_score->segment[segment].torsion_id) != NEITHER) &&
            ((flex = mol_score->torsion[torsion].flex_id) != NEITHER) &&
            (label->flex.member[flex].minimize_flag == TRUE)
          )
          {
            vertex_total++;
            mol_score->transform.tors_flag = TRUE;
          }

          if (layer > layer_inner)
          {
            for (j = 0; j < mol_score->segment[segment].neighbor_total; j++)
            {
              neighbor = mol_score->segment[segment].neighbor[j].id;

              if
              (
                (mol_score->segment[neighbor].layer_id < layer) &&
                (mol_score->segment[neighbor].layer_id >= layer_inner)
              )
                mol_score->segment[neighbor].min_flag = TRUE;
            }
          }
        }
      }
    }

/*
*   If no inner bonds can be minimized, then perform rigid-body minimization
*   4/97 te
*/
    if ((vertex_total == 0) && (label->flex.minimize_anchor_flag == TRUE))
    {
      rigid_flag = TRUE;
/*
      fprintf (global.outfile, "Rigid Min\n");
*/
    }

/*
*   Identify outer neighbors of minimizing segments (not already identified)
*   4/97 te
*/
    for (layer = layer_inner + 1; layer <= layer_outer; layer++)
    {
      for (i = 0; i < mol_score->layer[layer].segment_total; i++)
      {
        segment = mol_score->layer[layer].segment[i];

        if (mol_score->segment[segment].min_flag != TRUE)
        {
          for (j = 0; j < mol_score->segment[segment].neighbor_total; j++)
          {
            neighbor = mol_score->segment[segment].neighbor[j].id;

            if
            (
              (mol_score->segment[neighbor].layer_id < layer) &&
              (mol_score->segment[neighbor].layer_id >= layer_inner) &&
              (mol_score->segment[neighbor].min_flag == TRUE)
            )
            {
              if
              (
                ((torsion = mol_score->segment[segment].torsion_id)
                  != NEITHER) &&
                ((flex = mol_score->torsion[torsion].flex_id) != NEITHER) &&
                (label->flex.member[flex].minimize_flag == TRUE)
              )
              {
                vertex_total++;
                mol_score->transform.tors_flag = TRUE;
              }

              mol_score->segment[segment].min_flag = TRUE;
              break;
            }
          }
        }
      }
    }
  }

  if ((score->inter_flag) && (rigid_flag == TRUE))
  {
    simplex.rigid_flag = TRUE;
    vertex_total += 6;
    mol_score->transform.trans_flag =
      mol_score->transform.rot_flag = TRUE;
  }

  if (vertex_total > 0)
  {
    ecalloc
    (
      (void **) &vertex,
      vertex_total,
      sizeof (float),
      "simplex vertex array",
      global.outfile
    );

    simplex.dock = dock;
    simplex.score = score;
    simplex.label = label;
    simplex.mol_ref = mol_ref;
    simplex.mol_ori = mol_ori;
    simplex.mol_score = mol_score;
    simplex.layer_inner = layer_inner;
    simplex.layer_outer = layer_outer;

    copy_molecule (&simplex.mol_min, mol_score);
    copy_molecule (&simplex.mol_best, mol_score);
    simplex.mol_best.score.total = INITIAL_SCORE;
    delta_total = 0;

    for
    (
      simplex.cycle = 0, distance = FLT_MAX;
      (simplex.cycle < score->minimize.cycle) &&
        (distance > score->minimize.cycle_converge) &&
        ((simplex.cycle == 0) ||
          (simplex.mol_best.score.total <
          score->type[mol_score->score.type].termination));
      simplex.cycle++
    )
    {
      simplex_optimize
      (
        (void *) &simplex,
        vertex,
        vertex_total,
        score->type[mol_score->score.type].convergence,
        &iteration_total,
        score->minimize.iteration,
        simplex_score,
        &delta_cycle
      );

      copy_molecule (mol_score, &simplex.mol_best);

/*
*     Compute distance that vertex has moved (and update initial vertex)
*     1/97 te
*/
      for (i = distance = 0; i < vertex_total; i++)
        distance += SQR (vertex[i]);

      distance = sqrt (distance) / (float) (simplex.cycle + 1);
      memset (vertex, 0, vertex_total * sizeof (float));

/*
*     Update minimizer statistics
*     1/97 te
*/
      delta_total += delta_cycle;

      score->minimize.iteration_total += iteration_total;
      score->minimize.iteration_max =
        MAX (score->minimize.iteration_max, iteration_total);
      score->minimize.iteration_min =
        MIN (score->minimize.iteration_min, iteration_total);
    }

    score->minimize.call_sub_total++;

    score->minimize.vertex_total += vertex_total;
    score->minimize.vertex_max =
      MAX (score->minimize.vertex_max, vertex_total);
    score->minimize.vertex_min =
      MIN (score->minimize.vertex_min, vertex_total);

    score->minimize.cycle_total += simplex.cycle;
    score->minimize.cycle_max =
      MAX (score->minimize.cycle_max, simplex.cycle);
    score->minimize.cycle_min =
      MIN (score->minimize.cycle_min, simplex.cycle);

    score->minimize.delta_total += delta_total;
    score->minimize.delta_max =
      MAX (score->minimize.delta_max, delta_total);
    score->minimize.delta_min =
      MIN (score->minimize.delta_min, delta_total);

    free_molecule (&simplex.mol_min);
    free_molecule (&simplex.mol_best);
    efree ((void **) &vertex);
  }

  else
    calc_score
    (
      label,
      score,
      mol_score,
```

```
      layer_outer
    );

/*
* Turn off all minimize flags
* 3/97 te
*/
  if (label->flex.minimize_flag == TRUE)
    for (layer = layer_inner; layer <= layer_outer; layer++)
      for (i = 0; i < mol_score->layer[layer].segment_total; i++)
      {
        segment = mol_score->layer[layer].segment[i];
        mol_score->segment[segment].min_flag = FALSE;
      }
}

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

float simplex_score (void *simplex_input, float *vertex)
{
  int i;
  int flex;                     /* Flex label id */
  int torsion;                  /* Torsion id */
  int segment;                  /* Segment id */
  int layer;                    /* Layer id */
  int intra_current_flag;       /* Flag for whether score up-to-date */
  int vertex_count = 0;
  SIMPLEX *simplex;
  float score;

  simplex = (SIMPLEX *) simplex_input;

/*
* Extract transformation variables from simplex array if zeroth layer
* 12/96 te
*/
  if ((simplex->score->inter_flag) && (simplex->rigid_flag == TRUE))
  {
    for (i = 0; i < 3; i++)
    {
      simplex->mol_min.transform.translate[i] =
        simplex->mol_score->transform.translate[i] +
        vertex[i] * simplex->score->minimize.translation /
        (float) (simplex->cycle + 1);

      simplex->mol_min.transform.rotate[i] =
        simplex->mol_score->transform.rotate[i] +
        vertex[i + 3] * simplex->score->minimize.rotation /
        (float) (simplex->cycle + 1);
    }

    transform_molecule (&simplex->mol_min, simplex->mol_ref);

    vertex_count = 6;

/*
*   Flag all segment intermolecular scores as out-of-date
*   3/97 te
*/
    for (segment = 0; segment < simplex->mol_min.total.segments; segment++)
      simplex->mol_min.segment[segment].score.inter.current_flag = FALSE;
  }

  else
    vertex_count = 0;

/*
* Extract torsion variables from simplex array
* 3/97 te
*/
  if (simplex->label->flex.minimize_flag == TRUE)
  {
    intra_current_flag = TRUE;

    for (layer = simplex->layer_inner; layer <= simplex->layer_outer; layer++)
    {
      for (i = 0; i < simplex->mol_min.layer[layer].segment_total; i++)
      {
        segment = simplex->mol_min.layer[layer].segment[i];

        if ((simplex->mol_min.segment[segment].min_flag == TRUE) &&
          ((torsion = simplex->mol_min.segment[segment].torsion_id) != NEITHER))
        {
          if
          (
            ((flex = simplex->mol_min.torsion[torsion].flex_id) != NEITHER) &&
            (simplex->label->flex.member[flex].minimize_flag == TRUE)
          )
          {
            simplex->mol_min.torsion[torsion].target_angle =
              simplex->mol_score->torsion[torsion].target_angle +
              vertex[vertex_count++] *
                simplex->score->minimize.torsion * PI / 180.0 /
                (float) (simplex->cycle + 1) /
                (float) (simplex->layer_outer - layer + 1);

            torsion_transform (&simplex->mol_min, torsion);
          }

          intra_current_flag = FALSE;
          simplex->mol_min.segment[segment].score.inter.current_flag = FALSE;
        }
      }

      if (intra_current_flag == FALSE)
        for (i = 0; i < simplex->mol_min.layer[layer].segment_total; i++)
        {
          segment = simplex->mol_min.layer[layer].segment[i];
          simplex->mol_min.segment[segment].score.intra.current_flag = FALSE;
        }
    }
  }

/*
* Evaluate the score of this ligand position
* 12/96 te
*/
  score =
    calc_score
    (
      simplex->label,
      simplex->score,
      &simplex->mol_min,
      simplex->layer_outer
    );

  if (score < simplex->mol_best.score.total)
    copy_molecule (&simplex->mol_best, &simplex->mol_min);
```

```
  return score;
}

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

float calc_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           layer_outer
)
{
  int li, lj;                   /* Layer id */
  int sli, slj;                 /* Segment id in layer list */
  int si, sj;                   /* Segment id */
  int asi, asj;                 /* Atom id in segment list */
  int ai, aj;                   /* Atom id */

/*
* Update near flag array which speeds up intramolecular score calculation
* 3/97 te
*/
  if (score->intra_flag)
    initialize_near (&score->near, molecule);

/*
* Initialize score data
* 1/97 te
*/
  reset_score_parts (&molecule->score.intra);
  reset_score_parts (&molecule->score.inter);
  molecule->score.total = 0;

/*
* Loop through all layers of interest
* 1/97 te
  for (li = 0; li <= layer_outer; li++)
*/
  for (li = layer_outer; li >= 0; li--)
  {
/*
*   Loop through layer segments
*   3/97 te
*/
    for (sli = molecule->layer[li].segment_total - 1; sli >= 0; sli--)
    {
      si = molecule->layer[li].segment[sli];
      if (!molecule->segment[si].active_flag) continue;

/*
*     Calculate intermolecular score
*     3/97 te
*/
      if (score->inter_flag)
      {
        if (!molecule->segment[si].score.inter.current_flag ||
          (molecule->segment[si].score.type != molecule->score.type))
        {
          reset_score_parts (&molecule->segment[si].score.inter);
          molecule->segment[si].score.inter.current_flag = TRUE;

/*
*         Loop through segment atoms
*         3/97 te
*/
          for (asi = molecule->segment[si].atom_total - 1; asi >= 0; asi--)
          {
            ai = molecule->segment[si].atom[asi];

            calc_inter_score
            (
              label,
              score,
              molecule,
              ai,
              &molecule->segment[si].score.inter
            );
          }
        }

        sum_score
        (
          molecule->score.type,
          &molecule->score.inter,
          &molecule->segment[si].score.inter
        );
      }

/*
*     Calculate intramolecular score
*     3/97 te
*/
      if (score->intra_flag)
      {
        if (!molecule->segment[si].score.intra.current_flag ||
          (molecule->segment[si].score.type != molecule->score.type))
        {
          reset_score_parts (&molecule->segment[si].score.intra);
          molecule->segment[si].score.intra.current_flag = TRUE;

/*
*         Loop from current layer to all inner layers
*         3/97 te
*/
          for (lj = li; lj >= 0; lj--)
          {
/*
*           Loop through layer segments
*             For current layer, only consider previous segments
*             For inner layers, consider all segments
*           3/97 te
*/
            for
            (
              slj =
                (li == lj
                  ? sli - 1
                  : molecule->layer[lj].segment_total - 1);
              slj >= 0;
              slj--
            )
            {
              sj = molecule->layer[lj].segment[slj];

/*
*             Loop through segment atoms
*             3/97 te
```

```
*/
              for (asi = molecule->segment[si].atom_total - 1; asi >= 0; asi--)
              {
                ai = molecule->segment[si].atom[asi];

                for
                  (asj = molecule->segment[sj].atom_total - 1; asj >= 0; asj--)
                {
                  aj = molecule->segment[sj].atom[asj];

                  if (!score->near.flag[ai][aj])
                    calc_intra_score
                    (
                      label,
                      score,
                      molecule,
                      ai,
                      aj,
                      &molecule->segment[si].score.intra
                    );

                } /* End of asj loop */
              } /* End of asi loop */
            } /* End of slj loop */
          } /* End of lj loop */
        } /* End of current_flag IF block */

        sum_score
        (
          molecule->score.type,
          &molecule->score.intra,
          &molecule->segment[si].score.intra
        );

      } /* End of intra-scoring IF block */

      molecule->segment[si].score.type = molecule->score.type;
      molecule->segment[si].score.total =
        molecule->segment[si].score.intra.total +
        molecule->segment[si].score.inter.total;

    } /* End of sli loop */
  } /* End of li loop */

  molecule->score.total =
    molecule->score.intra.total +
    molecule->score.inter.total;

  return molecule->score.total;
}


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

void calc_intra_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           atomi,
  int           atomj,
  SCORE_PART    *intra
)
{
  if (molecule->score.type == CONTACT)
    calc_intra_contact
    (
      &score->contact,
      label,
      molecule,
      atomi,
      atomj,
      intra
    );

  else if (molecule->score.type == ENERGY)
    calc_intra_energy
    (
      &score->energy,
      label,
      molecule,
      atomi,
      atomj,
      intra
    );

  else if (molecule->score.type == CHEMICAL)
    calc_intra_energy
    (
      &score->energy,
      label,
      molecule,
      atomi,
      atomj,
      intra
    );

  else
    exit (fprintf (global.outfile,
      "ERROR update_score_pair: illegal score type requested\n"));
}


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

void calc_inter_score
(
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  int           atom,
  SCORE_PART    *inter
)
{
  if (molecule->score.type == CONTACT)
    calc_inter_contact
    (
      &score->grid,
      &score->bump,
      &score->contact,
      label,
      molecule,
      atom,
      inter
    );

  else if (molecule->score.type == CHEMICAL)
    calc_inter_chemical
    (
      &score->grid,
      &score->energy,
      &score->chemical,
      label,
      molecule,
      atom,
      inter
    );

  else if (molecule->score.type == ENERGY)
    calc_inter_energy
    (
      &score->grid,
      &score->energy,
      label,
      molecule,
      atom,
      inter
    );

  else if (molecule->score.type == RMSD)
    calc_inter_rmsd
    (
      &score->grid,
      molecule,
      atom,
      inter
    );
}


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

void sum_score
(
  int           score_type,
  SCORE_PART    *sum,
  SCORE_PART    *increment
)
{
  if (score_type == CONTACT)
    sum_contact (sum, increment);

  else if (score_type == ENERGY)
    sum_energy (sum, increment);

  else if (score_type == CHEMICAL)
    sum_chemical (sum, increment);

  else if (score_type == RMSD)
    sum_rmsd (sum, increment);

  else
    exit (fprintf (global.outfile, "ERROR sum_score: unknown score type\n"));
}

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

void output_score_info
(
  DOCK          *dock,
  LABEL         *label,
  SCORE         *score,
  MOLECULE      *molecule,
  LIST          *list,
  int           ligand_id,
  float         time
)
{
  int i;
  STRING80 line;

  if (global.output_volume == 't')
  {
    if (ligand_id == 0)
    {
      fprintf (global.outfile, "%s", "Result_header");
      fprintf (global.outfile, " %s", "name");

      if (label->flex.flag)
      {
        if (label->flex.multiple_anchors)
          fprintf (global.outfile, " %s", "anchors");

        if (label->flex.drive_flag && !label->flex.anchor_flag)
          fprintf (global.outfile, " %s", "conforms");
      }

      if (dock->multiple_orients)
      {
        fprintf (global.outfile, " %s", "matches");

        if (score->bump.flag)
          fprintf (global.outfile, " %s", "orients");
      }

      for (i = 0; i < SCORE_TOTAL; i++)
        if (score->type[i].flag)
        {
          fprintf (global.outfile, " %s", score->type[i].name);

          if (score->intra_flag && score->intra_flag)
            fprintf (global.outfile, " %s %s", "intra", "inter");

          if (dock->multiple_orients || score->type[i].minimize)
            fprintf (global.outfile, " %s", "rmsd");
        }

      fprintf (global.outfile, " %s\n", "time");
    }

    fprintf (global.outfile, "%s", "Results");
    fprintf (global.outfile, " %s", molecule->info.name);

    if (label->flex.flag)
    {
      if (label->flex.multiple_anchors)
        fprintf (global.outfile, " %d", dock->anchor_total);

      if (label->flex.drive_flag && !label->flex.anchor_flag)
        fprintf (global.outfile, " %d", dock->conform_total);
    }

    if (dock->multiple_orients)
    {
      fprintf (global.outfile, " %d", dock->orient_total);

      if (score->bump.flag)
        fprintf (global.outfile, " %d", dock->score_total);
```

```
    }

    for (i = 1; i < SCORE_TOTAL; i++)
      if (score->type[i].flag)
      {
        fprintf (global.outfile, " %.2f",
          list->member[i][0]->score.total);

        if (score->intra_flag && score->intra_flag)
          fprintf (global.outfile, " %.2f %.2f",
            list->member[i][0]->score.intra.total,
            list->member[i][0]->score.inter.total);

        if (dock->multiple_orients || score->type[i].minimize)
          fprintf (global.outfile, " %.2f", list->member[i][0]->transform.rmsd);
      }

    fprintf (global.outfile, " %.2f\n", time);
  }

  else
  {
    fprintf (global.outfile,
      "\n______________________Docking_Results______________________\n");

    fprintf (global.outfile, "%-12s : %s\n", "Name", molecule->info.name);
    fprintf (global.outfile, "%-12s : %s\n", "Description",
      molecule->info.comment);

    if (label->flex.flag)
    {
      if (label->flex.multiple_anchors)
        fprintf (global.outfile, "%-50s : %10d\n", "Total anchors",
          dock->anchor_total);

      if (label->flex.drive_flag && !label->flex.anchor_flag)
        fprintf (global.outfile, "%-50s : %10d\n", "Total conformations",
          dock->conform_total);
    }

    if (dock->multiple_orients)
    {
      fprintf (global.outfile, "%-50s : %10d\n", "Orientations tried",
        dock->orient_total);

      if (score->bump.flag)
        fprintf (global.outfile, "%-50s : %10d\n", "Orientations scored",
          dock->score_total);
    }

    fprintf (global.outfile, "\n");

    for (i = 1; i < SCORE_TOTAL; i++)
      if (score->type[i].flag)
      {
        if (score->intra_flag || score->inter_flag)
        {
          sprintf (line, "Best %s%s score",
            score->intra_flag
              ? (score->inter_flag ? "" : "intramolecular ")
              : (score->inter_flag ? "intermolecular " : ""),
            score->type[i].name);

          fprintf (global.outfile, "%-50s : %10.2f\n", line,
            list->member[i][0]->score.total);

          if (score->intra_flag && score->inter_flag)
          {
            sprintf (line, "  Intramolecular %s score", score->type[i].name);
            fprintf (global.outfile, "%-50s : %10.2f\n", line,
              list->member[i][0]->score.intra.total);

            sprintf (line, "  Intermolecular %s score", score->type[i].name);
            fprintf (global.outfile, "%-50s : %10.2f\n", line,
              list->member[i][0]->score.inter.total);
          }
        }

        if (dock->multiple_orients|| score->type[i].minimize)
        {
          sprintf (line, "RMSD of best %s scorer (A)",
            score->type[i].name);
          fprintf (global.outfile, "%-50s : %10.2f\n", line,
            list->member[i][0]->transform.rmsd);
        }

        if
        (
          !dock->rank_ligands &&
          dock->write_orients &&
          !dock->rank_orients
        )
          fprintf (global.outfile, "%-50s : %10d\n", "Orientations written",
            score->type[i].number_written);

        fprintf (global.outfile, "\n");
      }

    fprintf (global.outfile,
      "%-50s : %10.2f\n", "Elapsed cpu time (sec)", time);

    fprintf (global.outfile, "\n\n");
  }

  fflush (global.outfile);
}


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

Top score list processing routines

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

void allocate_lists
(
  SCORE         *score,
  LIST          *list,
  int           length,
  int           lite_flag
)
{
  int i;

  list->lite_flag = lite_flag;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
    {
      list->total[i] = length;
      allocate_list (list, i);
      list->total[i] = 0;
    }
}

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

void allocate_list
(
  LIST          *list,
  int           type
)
{
  int i;

  ecalloc
  (
    (void **) &list->member[type],
    list->total[type],
    sizeof (MOLECULE *),
    "molecule list",
    global.outfile
  );

  for (i = 0; i < list->total[type]; i++)
  {
    ecalloc
    (
      (void **) &list->member[type][i],
      1,
      sizeof (MOLECULE),
      "molecule list",
      global.outfile
    );

    reset_molecule (list->member[type][i]);
  }

  list->max[type] = list->total[type];
}

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

void reset_lists (SCORE *score, LIST *list)
{
  int i;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
      reset_list (list, i);
}

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

void reset_list (LIST *list, int type)
{
  int i;

  for (i = 0; i < list->max[type]; i++)
    reset_molecule (list->member[type][i]);

  list->total[type] = 0;
}

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

void free_lists (SCORE *score, LIST *list)
{
  int i;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
      free_list (list, i);
}


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

void free_list (LIST *list, int type)
{
  int i;

  for (i = 0; i < list->max[type]; i++)
  {
    free_molecule (list->member[type][i]);
    efree ((void **) &list->member[type][i]);
  }

  efree ((void **) &list->member[type]);
  list->max[type] = 0;
}


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

void reallocate_lists (SCORE *score, LIST *list)
{
  int i;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
      reallocate_list (list, i);
}


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

void reallocate_list (LIST *list, int type)
{
  if (list->max[type] < list->total[type])
  {
    free_list (list, type);
    list->max[type] = list->total[type];
    allocate_list (list, type);
  }
}

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

void copy_lists (SCORE *score, LIST *copy, LIST *original)
{
  int i;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
      copy_list (copy, original, i);
}
```

```
/* ////////////////////////////////////////////////////////////////////// */

void copy_list (LIST *copy, LIST *original, int type)
{
  int i;

  copy->total[type] = original->total[type];
  reallocate_list (copy, type);

  for (i = 0; i < original->total[type]; i++)
    copy_member
      (original->lite_flag, copy->member[type][i], original->member[type][i]);

  copy->total[type] = original->total[type];
}

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

void inter_lists (SCORE *score, LIST *list, MOLECULE *reference)
{
  int i, j;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
    {
      for (j = 0; j < list->total[i]; j++)
        list->member[i][j]->score.total =
          list->member[i][j]->score.inter.total +
          list->member[i][j]->transform.heavy_total *
            score->type[i].size_penalty;

      sort_list (list, i);

      if (i != RMSD)
      {
        for (j = 0; j < list->total[i]; j++)
          if (list->member[i][j]->score.total > 0)
            list->total[i] = j;

        if (list->total[i] == 0)
        {
          reference->score.type = i;
          reference->score.total = 0;
          update_list (list, reference);
        }
      }
    }
}

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

void merge_lists (SCORE *score, LIST *list, LIST *add_list)
{
  int i, j;

  for (i = 0; i < SCORE_TOTAL; i++)
    if (score->type[i].flag == TRUE)
      for (j = 0; (j < add_list->total[i]) && (j < list->max[i]); j++)
        update_list (list, add_list->member[i][j]);
}

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

void update_list
(
  LIST *list,
  MOLECULE *molecule
)
{
  int type;

  type = molecule->score.type;

  if ((type < 0) || (type >= SCORE_TOTAL))
  {
    fprintf (global.outfile, "ERROR update_list: Accessing illegal list.\n");
    exit (EXIT_FAILURE);
  }

  if (list->total[type] < list->max[type])
    copy_member
    (
      list->lite_flag,
      list->member[type][list->total[type]++],
      molecule
    );

  else if (type == 0)
    copy_member
    (
      list->lite_flag,
      list->member[type][list->max[type] - 1],
      molecule
    );

  else if (molecule->score.total <
    list->member[type][list->max[type] - 1]->score.total)
    copy_member
    (
      list->lite_flag,
      list->member[type][list->max[type] - 1],
      molecule
    );

  else
    return;

  sort_list (list, type);
}

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

int compare_score (MOLECULE *reference, MOLECULE *candidate)
{
  if
  (
    (reference->info.name != NULL) &&
    (candidate->info.name != NULL) &&
    !strcmp (reference->info.name, candidate->info.name)
  )
  {
    if (candidate->score.total < reference->score.total)
      return TRUE;
  }

  else
  {
    if (candidate->score.inter.total < reference->score.inter.total)
      return TRUE;
  }

  return FALSE;
}

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

void sort_list (LIST *list, int type)
{
  if (list->total[type] > 1)
    qsort
    (
      list->member[type],
      list->total[type],
      sizeof (MOLECULE *),
      (int (*)()) compare_member
    );
}

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

void copy_member (int lite_flag, MOLECULE *old, MOLECULE *new)
{
  if (lite_flag == TRUE)
  {
    copy_info (old, new);
    copy_transform (old, new);
    copy_score (&old->score, &new->score);
    copy_torsions (old, new);
    copy_keys (old, new);
  }

  else
    copy_molecule (old, new);
}

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

int compare_member (MOLECULE **mol1, MOLECULE **mol2)
{
  return ((*mol1)->score.total == (*mol2)->score.total) ? 0 :
    (((*mol1)->score.total > (*mol2)->score.total) ? 1 : -1);
}

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

int print_lists (SCORE *score, LIST *list, FILE *file)
{
  int i, j;

  if (file == NULL)
    return FALSE;

  for (i = 0; i < SCORE_TOTAL; i++)
  {
    if (score->type[i].flag == TRUE)
    {
      for (j = 0; j < list->total[i]; j++)
        fprintf (file, "%d %d %f %f\n",
          i, j + 1, list->member[i][j]->score.total,
          list->member[i][j]->transform.rmsd);
    }
  }

  return TRUE;
}

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

int save_lists (SCORE *score, LIST *list, FILE *file)
{
  int i, j;

  for (i = 0; i < SCORE_TOTAL; i++)
  {
    if (score->type[i].flag == TRUE)
    {
      efwrite (&list->total[i], sizeof (int), 1, file);

      for (j = 0; j < list->total[i]; j++)
        save_molecule (list->member[i][j], file);
    }
  }

  return TRUE;
}

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

int load_lists (SCORE *score, LIST *list, FILE *file)
{
  int i, j;

  for (i = 0; i < SCORE_TOTAL; i++)
  {
    if (score->type[i].flag == TRUE)
    {
      efread (&list->total[i], sizeof (int), 1, file);

      if (list->total[i] > list->max[i])
        return FALSE;

      for (j = 0; j < list->total[i]; j++)
        load_molecule (list->member[i][j], file);
    }
  }

  return TRUE;
}


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

void shrink_list
(
  LIST          *full,
  LIST          *shrunk,
  int           type
)
{
  int i, j;
  int remainder;
  static int *discard = NULL;
  static float **rmsd = NULL;
  static int size = 0;
  int discard_member;
  float rmsd_min;

  if (full->total[type] <= 0)
```

```
  {
    fprintf (global.outfile, "WARNING shrink_list: No members provided\n");
    return;
  }

/*
* Eliminate members with repulsive scores
* (but only if the best one is attractive).
* 3/97 te
  if (full->member[type][0]->score.total < 0)
    for (i = 0; i < full->total[type]; i++)
      if (full->member[type][i]->score.total > 0)
        full->total[type] = i;
*/
  for (i = 0; i < full->total[type]; i++)
    if (full->member[type][i]->score.total >= INITIAL_SCORE)
      full->total[type] = i;

/*
* Allocate arrays
* 3/97 te
*/
  if (size < full->total[type])
  {
    for (i = 0; i < size; i++)
      efree ((void **) &rmsd[i]);
    efree ((void **) &rmsd);
    efree ((void **) &discard);

    size = full->total[type];

    ecalloc
    (
      (void **) &discard,
      size,
      sizeof (int),
      "shrink discard array",
      global.outfile
    );

    ecalloc
    (
      (void **) &rmsd,
      size,
      sizeof (float *),
      "shrink rmsd array",
      global.outfile
    );

    for (i = 0; i < size; i++)
      ecalloc
      (
        (void **) &rmsd[i],
        size,
        sizeof (float),
        "shrink rmsd array",
        global.outfile
      );
  }

  else
  {
    for (i = 0; i < size; i++)
      memset (rmsd[i], 0, size * sizeof (float));
    memset (discard, 0, size * sizeof (int));
  }

/*
* Compute the RMSD between all members
* 3/97 te
*/
  for (i = 0; i < full->total[type]; i++)
    for (j = i + 1; j < full->total[type]; j++)
    {
      rmsd[i][j] =
        calc_layer_rmsd
          (full->member[type][i], full->member[type][j]) / (float) j;
    }

/*
* Discard members until the desired number remains
* 3/97 te
*/
  for
  (
    remainder = full->total[type];
    remainder > shrunk->max[type];
    remainder--
  )
  {
    rmsd_min = FLT_MAX;
    discard_member = NEITHER;

    for (i = 0; i < full->total[type]; i++)
      if (!discard[i])
        for (j = i + 1; j < full->total[type]; j++)
          if (!discard[j] && (rmsd[i][j] < rmsd_min))
          {
            rmsd_min = rmsd[i][j];
            discard_member = j;
          }

    if (discard_member == NEITHER)
      exit (fprintf (global.outfile, "ERROR shrink_list: No member found\n"));

    else
      discard[discard_member] = TRUE;
  }

/*
* Transfer remainder over to shrunk list
* 3/97 te
*/
  reset_list (shrunk, type);

  for (i = 0; i < full->total[type]; i++)
    if (!discard[i])
      update_list (shrunk, full->member[type][i]);
}


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

void update_shrunk_list (LIST *shrunk, MOLECULE *molecule)
{
  int type;
  static LIST full;

  type = molecule->score.type;

  full.total[type] = shrunk->max[type] + 1;
  reallocate_list (&full, type);
  copy_list (&full, shrunk, type);
  update_list (&full, molecule);
  shrink_list (&full, shrunk, type);
}


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

void initialize_near
(
  NEAR          *near,
  MOLECULE      *molecule
)
{
  int i, j, k;
  int ni, nni;

/*
* Check if near data is up-to-date
* 3/97 te
*/
  if
  (
    (near->total == molecule->total.atoms) &&
    (near->name != NULL) &&
    (molecule->info.name != NULL) &&
    !strcmp (near->name, molecule->info.name)
  )
    return;

/*
* Either reallocate space for the flags or reset them
* 3/97 te
*/
  if (near->total < molecule->total.atoms)
  {
    free_near (near);
    near->total = molecule->total.atoms;
    allocate_near (near);
  }

  else
    for (i = 0; i < near->total; i++)
      memset (near->flag[i], 0, near->total * sizeof (int));

  vstrcpy (&near->name, molecule->info.name);

  for (i = 0; i < molecule->total.atoms; i++)
    for (j = 0; j < molecule->atom[i].neighbor_total; j++)
    {
      ni = molecule->atom[i].neighbor[j].id;
      near->flag[i][ni] = near->flag[ni][i] = TRUE;

      for (k = 0; k < molecule->atom[ni].neighbor_total; k++)
      {
        nni = molecule->atom[ni].neighbor[k].id;
        near->flag[i][nni] = near->flag[nni][i] = TRUE;
      }
    }
}


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

void allocate_near (NEAR *near)
{
  int i;

  ecalloc
  (
    (void **) &near->flag,
    near->total,
    sizeof (int *),
    "near flag array",
    global.outfile
  );

  for (i = 0; i < near->total; i++)
    ecalloc
    (
      (void **) &near->flag[i],
      near->total,
      sizeof (int *),
      "near flag array",
      global.outfile
    );
}


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

void free_near (NEAR *near)
{
  int i;

  for (i = 0; i < near->total; i++)
    efree ((void **) &near->flag[i]);

  efree ((void **) &near->flag);
  efree ((void **) &near->name);
}


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

void free_scores (LABEL *label, SCORE *score)
{
  free_near (&score->near);

  if (score->grid.flag)
    free_grids
    (
      &score->grid,
      &score->bump,
      &score->contact,
      &score->chemical,
      &score->energy,
      &label->chemical
    );

  if (score->energy.flag || score->chemical.flag)
    free_vdw_energy (&score->energy);

  if (score->chemical.flag)
    free_table (&label->chemical, &label->chemical.score_table);
}
*****************************************************************************
```

```
*********************************** SCORE_GRID.H ***********************************
************************************************************************************
/*                                                                       */
/*                       Copyright UCSF, 1997                            */
/*                                                                       */

/*
Written by Todd Ewing
9/96 te
*/

void make_grids
(
  SCORE_GRID *,
  SCORE_BUMP *,
  SCORE_CONTACT *,
  SCORE_CHEMICAL *,
  SCORE_ENERGY *,
  LABEL *,
  MOLECULE *
);

************************************************************************************
*********************************** SCORE_GRID.C ***********************************
************************************************************************************
/*                                                                       */
/*                       Copyright UCSF, 1997                            */
/*                                                                       */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "mol.h"
#include "global.h"
#include "search.h"
#include "label.h"
#include "score.h"
#include "grid.h"
#include "score_grid.h"

void make_grids
(
  SCORE_GRID *grid,
  SCORE_BUMP *bump,
  SCORE_CONTACT *contact,
  SCORE_CHEMICAL *chemical,
  SCORE_ENERGY *energy,
  LABEL *label,
  MOLECULE *receptor
)
{
  int i, j, k, l, atomi, index;
  int ilo, ihi, jlo, jhi, klo, khi;
  int contact_minimum;
  int chem_id;
  int inside_grid;
  float dist, dist_sq;
  float dist_inv;
  float dist_min;
  float dist_sq_min;
  float dist_power;
  int report_increment;

  XYZ origin_coord, gridpt_coord;
  int grid_cutoff, grid_coord[3];
  float square_distance (XYZ, XYZ);

  dist_min = DISTANCE_MIN;
  dist_sq_min = SQR (DISTANCE_MIN);

  if (!energy->vdw_init_flag)
    initialize_vdw_energy (energy, &label->vdw);

/*
  if (energy->flag)
    energy->dielectric_factor = 322.0 / energy->dielectric_factor;
*/

/*
* Determine the longest interaction distance to consider.
* Take the square of each radii in preparation for grid calculation.
*/
  grid->distance = 0.0;

  if (bump->flag)
  {
    for (i = 0, bump->distance = 0.0; i < label->vdw.total; i++)
      bump->distance = MAX
        (bump->distance,
        2.0 * bump->clash_overlap * label->vdw.member[i].radius);

    grid->distance = MAX (grid->distance, bump->distance);
  }

  if (contact->flag)
    grid->distance = MAX (grid->distance, contact->distance);

  if (energy->flag)
    grid->distance = MAX (grid->distance, energy->distance);

  grid_cutoff = (int) (grid->distance / grid->spacing + 1.0);
  report_increment = receptor->total.atoms / 10;

  for (atomi = 0; atomi < receptor->total.atoms; atomi++)
  {
    inside_grid = TRUE;
    for (i = 0; i < 3; i++)
    {
      origin_coord[i] = receptor->coord[atomi][i] - grid->origin[i];
      grid_coord[i] = NINT (origin_coord[i] / grid->spacing);

      if (grid_coord[i] >= (grid->span[i] + grid_cutoff))
        inside_grid = FALSE;
      if (grid_coord[i] < (0 - grid_cutoff))
        inside_grid = FALSE;
    }

    if (inside_grid)
    {
      ilo = MAX (0, (grid_coord[0] - grid_cutoff));
      ihi = MIN (grid->span[0], (grid_coord[0] + grid_cutoff + 1));
      jlo = MAX (0, (grid_coord[1] - grid_cutoff));
      jhi = MIN (grid->span[1], (grid_coord[1] + grid_cutoff + 1));
      klo = MAX (0, (grid_coord[2] - grid_cutoff));
      khi = MIN (grid->span[2], (grid_coord[2] + grid_cutoff + 1));

      chem_id = receptor->atom[atomi].chem_id;

      for (i = ilo; i < ihi; i++)
      {
        gridpt_coord[0] =
          ((float) (i)) *
          grid->spacing + grid->origin[0];

        for (j = jlo; j < jhi; j++)
        {
          gridpt_coord[1] =
            ((float) (j)) *
            grid->spacing + grid->origin[1];

          for (k = klo; k < khi; k++)
          {
            gridpt_coord[2] =
              ((float) (k)) *
              grid->spacing + grid->origin[2];

            index =
              grid->span[0] * grid->span[1] * k +
              grid->span[0] * j + i;

            dist_sq = square_distance (receptor->coord[atomi], gridpt_coord);

            if (dist_sq > dist_sq_min)
              dist = sqrt (dist_sq);

            else
            {
              dist = dist_min;
              dist_sq = dist_sq_min;
            }

            if (dist <= grid->distance)
            {
/*
*             Increment bump grids
*/
              if
              (
                bump->flag &&
                (receptor->atom[atomi].heavy_flag == TRUE) &&
                (dist <= bump->distance)
              )
                for (l = 0; l < label->vdw.total; l++)
                  if
                  (
                    (label->vdw.member[l].heavy_flag == TRUE) &&
                    (label->vdw.member[l].bump_id < bump->grid[index]) &&
                    (dist < bump->clash_overlap *
                      (label->vdw.member[receptor->atom[atomi].vdw_id].radius +
                      label->vdw.member[l].radius))
                  )
                    bump->grid[index] = label->vdw.member[l].bump_id;

/*
*             Increment contact grids
*/
              if
              (
                contact->flag &&
                (dist <= contact->distance) &&
                (receptor->atom[atomi].heavy_flag == TRUE)
              )
                contact->grid[index] -= 1;

/*
*             Increment energy and chemical grids
*/
              if
              (
                energy->flag &&
                (label->vdw.member[receptor->atom[atomi].vdw_id].well_depth !=
                  0.0) &&
                (dist <= energy->distance)
              )
              {
                dist_inv = 1.0 / dist;

                POWER (dist_inv, energy->repulsive_exponent, dist_power);
                energy->avdw[index] +=
                  energy->vdwA[receptor->atom[atomi].vdw_id] *
                  dist_power;

                POWER (dist_inv, energy->attractive_exponent, dist_power);
                energy->bvdw[index] +=
                  energy->vdwB[receptor->atom[atomi].vdw_id] *
                  dist_power;

                if (chemical->flag)
                  chemical->grid[chem_id][index] +=
                    energy->vdwB[receptor->atom[atomi].vdw_id] *
                    dist_power;

                if (energy->distance_dielectric)
                  energy->es[index] += energy->dielectric_factor *
                    receptor->atom[atomi].charge * SQR (dist_inv);

                else
                  energy->es[index] += energy->dielectric_factor *
                    receptor->atom[atomi].charge * dist_inv;
              }
            }
          }
        }
      }
    }

    if (!(atomi % report_increment))
    {
      fprintf (global.outfile, "%-40s: %8d\n",
        "Percent of protein atoms processed",
        atomi / report_increment * 10);
      fflush (global.outfile);
    }
  }

/*
* Find value of contact grid minimum
*/
  if (contact->flag)
  {
    for (i = 0, contact_minimum = 0; i < grid->size; i++)
      if (contact->grid[i] < contact_minimum)
        contact_minimum = contact->grid[i];

    fprintf (global.outfile, "\n%-40s: %8d\n", "Global min of contact grid",
      contact_minimum);
```

```
  }
}


********************************************************************************
*********************************** SCREEN.H ***********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
10/95
*/

int  update_keys               (LABEL_CHEMICAL *, MOLECULE *, int);
int  update_folded_keys        (LABEL_CHEMICAL *, MOLECULE *, int);
int  update_unfolded_keys      (LABEL_CHEMICAL *, MOLECULE *, int);
MASK get_mask                  (CHEMICAL_SCREEN *, float);

void fold_keys
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *copy,
  MOLECULE                *original
);

int check_pharmacophore
(
  LABEL_CHEMICAL          *label_chemical,
  float                   uncertainty,
  MOLECULE                *target,
  MOLECULE                *candidate
);

void update_equivalency
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *molecule
);

void update_uncertainty
(
  LABEL_CHEMICAL          *label_chemical,
  float                   uncertainty,
  MOLECULE                *molecule
);

int mask_keys
(
  MOLECULE       *target,
  MOLECULE       *candidate
);

float check_dissimilarity
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *target,
  MOLECULE                *candidate
);

float compare_keys
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *target,
  MOLECULE                *candidate
);

int bit_count (MASK mask);
********************************************************************************
*********************************** SCREEN.C ***********************************
********************************************************************************
/*                                                                            */
/*                          Copyright UCSF, 1997                              */
/*                                                                            */

/*
Written by Todd Ewing
1/97
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "label.h"
#include "screen.h"
#include "vector.h"

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

Routine to update the distance keys for the current conformation.

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

int update_keys
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *molecule,
  int                     conform_id
)
{
  if (label_chemical->screen.fold_flag)
    return
      update_folded_keys (label_chemical, molecule, conform_id);

  else
    return
      update_unfolded_keys (label_chemical, molecule, conform_id);
}


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

Routine to update the distance keys between labeled atoms
for the current conformation.  The distance keys for all atoms with
the same label are folded on top of each other.

Return values:
        TRUE:   no problems
        FALSE:  big problem (unable to assign chemical labels)

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

int update_folded_keys
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *molecule,
  int                     conform_id
)
{
  int i, j;
  int li, lj;

  if (molecule->info.assign_chem == FALSE)
    if (assign_chemical_labels (label_chemical, molecule) != TRUE)
      return FALSE;

  if (conform_id == 0)
  {
    molecule->transform.fold_flag = TRUE;

    reset_keys (molecule);
    molecule->total.keys = label_chemical->total;
    reallocate_keys (molecule);

    for (i = 0; i < molecule->total.keys; i++)
      for (j = 0; j < molecule->total keys; j++)
        molecule->key[i][j].count = 0;

    for (i = 0; i < molecule->total.atoms; i++)
    {
      if (molecule->atom[i].heavy_flag != TRUE)
        continue;

      li = molecule->atom[i].chem_id;

      for (j = i + 1; j < molecule->total.atoms; j++)
      {
        if (molecule->atom[j].heavy_flag != TRUE)
          continue;

        lj = molecule->atom[j].chem_id;

        if (li < lj)
          molecule->key[li][lj].count++;

        else
          molecule->key[lj][li].count++;
      }
    }
  }

  for (i = 0; i < molecule->total.atoms; i++)
  {
    if (molecule->atom[i].heavy_flag != TRUE)
      continue;

    li = molecule->atom[i].chem_id;

    for (j = i + 1; j < molecule->total.atoms; j++)
    {
      if (molecule->atom[j].heavy_flag != TRUE)
        continue;

      lj = molecule->atom[j].chem_id;

/*
      if ((li == 2) && (lj == 2))
        fprintf (global.outfile, "atoms %s %s: dist %g\n",
          molecule->atom[i].name,
          molecule->atom[j].name,
          molecule->distance[i][j]);
*/

      if (li < lj)
        molecule->key[li][lj].distance |=
          get_mask
          (
            &label_chemical->screen,
            dist3 (molecule->coord[i], molecule->coord[j])
          );

      else
        molecule->key[lj][li].distance |=
          get_mask
          (
            &label_chemical->screen,
            dist3 (molecule->coord[i], molecule->coord[j])
          );
    }
  }

  return TRUE;
}


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

Routine to update the distance keys between ATOMS for the current conformation.

Return values:
        TRUE:   no problems
        FALSE:  big problem (unable to assign chemical labels)

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

int update_unfolded_keys
(
  LABEL_CHEMICAL          *label_chemical,
  MOLECULE                *molecule,
  int                     conform_id
)
{
  int i, j;               /* Atom iterators */

  if (molecule->info.assign_chem == FALSE)
    if (assign_chemical_labels (label_chemical, molecule) != TRUE)
      return FALSE;

  if (conform_id == 0)
  {
    molecule->transform.fold_flag = FALSE;

    reset_keys (molecule);
    molecule->total.keys = molecule->total.atoms;
    reallocate_keys (molecule);

    for (i = 0; i < molecule->total.atoms; i++)
      if (molecule->atom[i].heavy_flag == TRUE)
        for (j = i + 1; j < molecule->total.atoms; j++)
          if (molecule->atom[j].heavy_flag == TRUE)
            molecule->key[i][j].count = 1;
  }

  for (i = 0; i < molecule->total.atoms; i++)
    if (molecule->atom[i].heavy_flag == TRUE)
      for (j = i + 1; j < molecule->total.atoms; j++)
```

```
        if (molecule->atom[j].heavy_flag == TRUE)
          molecule->key[i][j].distance |=
            get_mask
            (
              &label_chemical->screen,
              dist3 (molecule->coord[i], molecule->coord[j])
            );

  return TRUE;
}


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

Routine to construct a bit mask encoding the current distance.
11/96 te

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

MASK get_mask
(
  CHEMICAL_SCREEN       *screen,
  float                 distance
)
{
  int key;

  key = NINT ((distance - screen->distance_minimum) /
    screen->distance_interval) + 1;

  key = MAX (key, 0);
  key = MIN (key, screen->interval_total - 1);

  return (MASK) 1 << key;
}


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

Routine to read unfolded keys from one molecule and record
them as folded keys in another molecule.

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

void fold_keys
(
  LABEL_CHEMICAL        *label_chemical,
  MOLECULE              *copy,
  MOLECULE              *original
)
{
  int i, j;
  int li, lj;

  copy->transform.fold_flag = TRUE;

  if (original->transform.fold_flag == TRUE)
  {
    copy_keys (copy, original);
    return;
  }

  reset_keys (copy);
  copy->total.keys = label_chemical->total;
  reallocate_keys (copy);

  for (i = 0; i < original->total.atoms; i++)
  {
    if (original->atom[i].heavy_flag != TRUE)
      continue;

    li = original->atom[i].chem_id;

    for (j = i + 1; j < original->total.atoms; j++)
    {
      if (original->atom[j].heavy_flag != TRUE)
        continue;

      lj = original->atom[j].chem_id;

      if (li <= lj)
      {
        copy->key[li][lj].count++;

        copy->key[li][lj].distance |=
          original->key[i][j].distance;
      }

      else
      {
        copy->key[lj][li].count++;

        copy->key[lj][li].distance |=
          original->key[i][j].distance;
      }
    }
  }
}


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

Routine to check if a candidate molecule might include a target.

Return values:
        TRUE:   candidate MAY include target
        FALSE:  candidate CANNOT include target
11/96 te
////////////////////////////////////////////////////////////////////// */

int check_pharmacophore
(
  LABEL_CHEMICAL        *label_chemical,
  float                 uncertainty,
  MOLECULE              *target,
  MOLECULE              *candidate
)
{
  static MOLECULE tmp_targ = {0};
  static MOLECULE tmp_cand = {0};

  reset_keys (&tmp_cand);
  reset_keys (&tmp_targ);

  fold_keys (label_chemical, &tmp_targ, target);
  fold_keys (label_chemical, &tmp_cand, candidate);

  update_equivalency (label_chemical, &tmp_cand);
  update_uncertainty (label_chemical, uncertainty, &tmp_cand);

  return
    mask_keys (&tmp_targ, &tmp_cand);
}


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

Routine to supplement keys to include off-diagonal equivalencies
in the chemical match table.
1/97 te

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

void update_equivalency
(
  LABEL_CHEMICAL        *label_chemical,
  MOLECULE              *molecule
)
{
  int i, j, k;
  static MOLECULE temporary = {0};

  if (molecule->transform.fold_flag != TRUE)
    exit (fprintf (global.outfile, "update_equivalency: keys not folded\n"));

  copy_keys (&temporary, molecule);

  for (i = 0; i < molecule->total.keys; i++)
    for (j = i + 1; j < molecule->total.keys; j++)
      if (label_chemical->match_table[i][j])
      {
/*
*       Update keys involving only equivalent labels
*       1/97 te
*/
        molecule->key[i][i].count +=
          temporary.key[i][j].count + temporary.key[j][j].count;

        molecule->key[i][j].count +=
          temporary.key[i][i].count + temporary.key[j][j].count;

        molecule->key[j][j].count +=
          temporary.key[i][i].count + temporary.key[i][j].count;

        molecule->key[i][i].distance |=
          temporary.key[i][j].distance | temporary.key[j][j].distance;

        molecule->key[i][j].distance |=
          temporary.key[i][i].distance | temporary.key[j][j].distance;

        molecule->key[j][j].distance |=
          temporary.key[i][i].distance | temporary.key[i][j].distance;

/*
*       Update keys involving equivalent labels and another label
*       1/97 te
*/
        for (k = 0; k < molecule->total.keys; k++)
          if ((k != i) && (k != j))
          {
            if (j < k)
            {
              molecule->key[i][k].count += temporary.key[j][k].count;
              molecule->key[i][k].distance |= temporary.key[j][k].distance;

              molecule->key[j][k].count += temporary.key[i][k].count;
              molecule->key[j][k].distance |= temporary.key[i][k].distance;
            }

            else if (i < k)
            {
              molecule->key[i][k].count += temporary.key[k][j].count;
              molecule->key[i][k].distance |= temporary.key[k][j].distance;

              molecule->key[k][j].count += temporary.key[i][k].count;
              molecule->key[k][j].distance |= temporary.key[i][k].distance;
            }

            else
            {
              molecule->key[k][i].count += temporary.key[k][j].count;
              molecule->key[k][i].distance |= temporary.key[k][j].distance;

              molecule->key[k][j].count += temporary.key[k][i].count;
              molecule->key[k][j].distance |= temporary.key[k][i].distance;
            }
          }
      }
}


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

Routine to smear keys according to the uncertainty in distance.
11/96 te

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

void update_uncertainty
(
  LABEL_CHEMICAL        *label_chemical,
  float                 uncertainty,
  MOLECULE              *molecule
)
{
  int i, j, k;
  int smear;

  smear = (int) (uncertainty /
    label_chemical->screen.distance_interval + .9999);

  for (i = 0; i < molecule->total.keys; i++)
    for (j = i; j < molecule->total.keys; j++)
      for (k = 0; k < smear; k++)
        molecule->key[i][j].distance |=
          molecule->key[i][j].distance << 1 |
          molecule->key[i][j].distance >> 1;
}


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

Routine to check if candidate molecule contains all keys of target molecule.

Return values:
        TRUE:   candidate molecule contains ALL keys
        FALSE:  candidate molecule doesn't contain all keys
11/96 te
```

```c
//////////////////////////////////////////////////////////////////// */

int mask_keys
(
  MOLECULE      *target,
  MOLECULE      *candidate
)
{
  int i, j;

/*
* Verify that a comparison can be made
* 11/96 te
*/
  if ((target->transform.fold_flag != TRUE) ||
    (candidate->transform.fold_flag != TRUE))
    exit (fprintf (global.outfile, "ERROR mask_keys: keys not folded.\n"));

/*
* Check chemical label compositions
* 11/96 te
*/
  for (i = 0; i < target->total.keys; i++)
    for (j = i; j < target->total.keys; j++)
    {
      if (candidate->key[i][j].count < target->key[i][j].count)
        return FALSE;

      if (target->key[i][j].distance !=
        (target->key[i][j].distance & candidate->key[i][j].distance))
        return FALSE;
    }

  return TRUE;
}


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

Routine to evaluate the similarity of a candidate molecule with
a target molecule.

Return value: FLOAT dissimilarity
11/96 te
//////////////////////////////////////////////////////////////////////// */

float check_dissimilarity
(
  LABEL_CHEMICAL        *label_chemical,
  MOLECULE              *target,
  MOLECULE              *candidate
)
{
  static MOLECULE tmp_targ = {0};
  static MOLECULE tmp_cand = {0};

  if (label_chemical->screen_table == NULL)
    get_table
    (
      label_chemical,
      label_chemical->screen_file_name,
      &label_chemical->screen_table
    );

  reset_keys (&tmp_cand);
  reset_keys (&tmp_targ);

  fold_keys (label_chemical, &tmp_targ, target);
  fold_keys (label_chemical, &tmp_cand, candidate);

  return
    1.0 - compare_keys (label_chemical, &tmp_targ, &tmp_cand);
}


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

Routine to compare the similarity of two molecules.

Return values:
        TRUE:   candidate molecule contains ALL keys
        FALSE:  candidate molecule doesn't contain all keys
11/96 te

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

float compare_keys
(
  LABEL_CHEMICAL        *label_chemical,
  MOLECULE              *target,
  MOLECULE              *candidate
)
{
  int i, j;
  float numerator = 0;
  float denominator = 0;

/*
* Verify that a comparison can be made
* 11/96 te
*/
  if ((target->transform.fold_flag != TRUE) ||
    (candidate->transform.fold_flag != TRUE))
    exit (fprintf (global.outfile, "ERROR compare_keys: keys not folded.\n"));

/*
* Check chemical label compositions
* 1/97 te
*/
  for (i = 0; i < target->total.keys; i++)
    for (j = i; j < target->total.keys; j++)
    {
      numerator +=
        label_chemical->screen_table[i][j] * (float)
        (MIN (candidate->key[i][j].count, target->key[i][j].count) *
        bit_count (target->key[i][j].distance & candidate->key[i][j].distance));

      denominator +=
        label_chemical->screen_table[i][j] * (float)
        (MAX (candidate->key[i][j].count, target->key[i][j].count) *
        bit_count (target->key[i][j].distance | candidate->key[i][j].distance));
    }

  if (denominator > 0.0)
    return numerator / denominator;

  else
    return 1;
}


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

Routine to count the number of bits turned ON in a mask.

Return values:  INT     the number of ON bits
11/96 te

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

int bit_count (MASK mask)
{
  int i;

  for (i = 0; mask; i++)
    mask >>= 1;

  return i;
}
*****************************************************************************
*********************************** SEARCH.H ********************************
*****************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

/*
Structures and routines to perform searches of linked elements
(eg. bonded atoms) in either depth-first or breadth-first manner.

Written by Todd Ewing
11/96
*/

typedef struct search_struct
{
  int complete_flag;
  int total_size;
  int max_size;
  int thread_total;
  int radius;
  int **target;
  int **origin;
  int **thread;
  int *total;
  int *count;
  int *neighbor_id;
  int *log;
} SEARCH;


/*
Routines to perform searches
11/96 te
*/

void allocate_search    (SEARCH *search);
void reset_search       (SEARCH *search);
void free_search        (SEARCH *search);
void reallocate_search  (SEARCH *search);

int breadth_search
(
  SEARCH        *search,
  void          *array,
  int           total,
  int           get_neighbor (void *, int, int),
  void          flag_neighbor (void *, int, int, int),
  int           *seed,
  int           seed_total,
  int           avoid,
  int           iteration
);

int depth_search
(
  SEARCH        *search,
  void          *array,
  int           total,
  int           get_neighbor (void *, int, int),
  void          flag_neighbor (void *, int, int, int),
  int           *seed,
  int           seed_total,
  int           avoid,
  int           iteration
);

int get_search_radius
(
  SEARCH        *search,
  int           target,
  int           origin
);

int get_search_origin
(
  SEARCH        *search,
  int           target,
  int           radius,
  int           count
);

int get_search_target
(
  SEARCH        *search,
  int           origin,
  int           radius,
  int           count
);

*****************************************************************************
*********************************** SEARCH.C ********************************
*****************************************************************************
/*                                                                          */
/*                          Copyright UCSF, 1997                            */
/*                                                                          */

#include "define.h"
#include "utility.h"
#include "global.h"
#include "search.h"


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

void allocate_search (SEARCH *search)
{
```

```
  int i;

  if (search->max_size > 0)
  {
    ecalloc
    (
      (void **) &search->count,
      search->max_size + 1,
      sizeof (int),
      "search count",
      global.outfile
    );

    ecalloc
    (
      (void **) &search->total,
      search->max_size + 1,
      sizeof (int),
      "search total",
      global.outfile
    );

    ecalloc
    (
      (void **) &search->target,
      search->max_size + 1,
      sizeof (int *),
      "search target",
      global.outfile
    );

    ecalloc
    (
      (void **) &search->origin,
      search->max_size + 1,
      sizeof (int *),
      "search origin",
      global.outfile
    );

    for (i = 0; i < search->max_size + 1; i++)
    {
      ecalloc
      (
        (void **) &search->target[i],
        search->max_size,
        sizeof (int),
        "search target",
        global.outfile
      );

      ecalloc
      (
        (void **) &search->origin[i],
        search->max_size,
        sizeof (int),
        "search origin",
        global.outfile
      );
    }

    ecalloc
    (
      (void **) &search->neighbor_id,
      search->max_size + 1,
      sizeof (int),
      "search neighbor id",
      global.outfile
    );

    ecalloc
    (
      (void **) &search->log,
      search->max_size + 1,
      sizeof (int),
      "search log",
      global.outfile
    );
  }
}

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

void reset_search (SEARCH *search)
{
  int i;

  if (search->max_size > 0)
  {
    memset (search->count, 0, (search->max_size + 1) * sizeof (int));
    memset (search->total, 0, (search->max_size + 1) * sizeof (int));

    for (i = 0; i < search->max_size + 1; i++)
    {
      memset (search->target[i], 0, search->max_size * sizeof (int));
      memset (search->origin[i], 0, search->max_size * sizeof (int));
    }

    memset (search->neighbor_id, 0, (search->max_size + 1) * sizeof (int));
    memset (search->log, 0, (search->max_size + 1) * sizeof (int));
  }

  search->complete_flag = FALSE;
  search->radius = 0;
  search->total_size = 0;
}

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

void free_search (SEARCH *search)
{
  int i;

  efree ((void **) &search->count);
  efree ((void **) &search->total);

  if (search->max_size > 0)
  {
    for (i = 0; i < search->max_size + 1; i++)
    {
      efree ((void **) &search->target[i]);
      efree ((void **) &search->origin[i]);
    }
  }

  efree ((void **) &search->target);
  efree ((void **) &search->origin);
  efree ((void **) &search->neighbor_id);
  efree ((void **) &search->log);

  search->max_size = 0;
}

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

void reallocate_search (SEARCH *search)
{
  if (search->total_size > search->max_size)
  {
    free_search (search);
    search->max_size = search->total_size;
    allocate_search (search);
  }
}


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

Subroutine to traverse a list of linked nodes non-recursively, breadth-first.

11/96 te

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

int breadth_search
(
  SEARCH        *search,
  void          *nodes,
  int           total,
  int           get_neighbor (void *, int, int),
  void          flag_neighbor (void *, int, int, int),
  int           *seed,
  int           seed_total,
  int           avoid,
  int           iteration
)
{
  int i;
  int node;
  int neighbor;

  if (iteration == 0)
  {
    reset_search (search);
    search->total_size = total;
    reallocate_search (search);

    if ((avoid >= 0) && (avoid < total))
      search->log[avoid] = TRUE;

    for (i = 0; i < seed_total; i++)
    {
      if ((seed[i] >= 0) && (seed[i] < total))
      {
        search->log[seed[i]] = TRUE;
        search->target[search->radius][search->total[search->radius]] = seed[i];
        search->origin[search->radius][search->total[search->radius]] = NEITHER;
        search->total[search->radius]++;
      }

      else
        exit (fprintf (global.outfile,
          "ERROR breadth_search: seed value inappropriate (%d).\n", seed[i]));
    }
  }

  else
    search->count[search->radius]++;

  if (search->count[search->radius] < search->total[search->radius])
    return search->target[search->radius][search->count[search->radius]];

  else
  {
    for
    (
      search->count[search->radius] = 0;
      search->count[search->radius] < search->total[search->radius];
      search->count[search->radius]++
    )
    {
      node = search->target[search->radius][search->count[search->radius]];

      for
      (
        search->neighbor_id[search->radius] = 0;
        (neighbor = get_neighbor
          (nodes, node, search->neighbor_id[search->radius])) != EOF;
        search->neighbor_id[search->radius]++
      )
      {
        if (flag_neighbor)
          flag_neighbor
          (
            nodes,
            node,
            search->neighbor_id[search->radius],
            !search->log[neighbor]
          );

        if (!search->log[neighbor])
        {
          search->target
            [search->radius + 1]
            [search->total[search->radius + 1]] = neighbor;
          search->origin
            [search->radius + 1]
            [search->total[search->radius + 1]] = node;

          search->total[search->radius + 1]++;
          search->log[neighbor] = TRUE;
        }
      }
    }

    if (search->total[search->radius + 1] > 0)
    {
      search->radius++;
      return search->target[search->radius][search->count[search->radius]];
    }

    else
    {
      search->complete_flag = TRUE;
      return EOF;
    }
  }
}
```

```
/* ///////////////////////////////////////////////////////////////////////

Subroutine to traverse a list of linked nodes non-recursively, depth-first.

11/96 te

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

int depth_search
(
  SEARCH        *search,
  void          *nodes,
  int           total,
  int           get_neighbor (void *, int, int),
  void          flag_neighbor (void *, int, int, int),
  int           *seed,
  int           seed_total,
  int           avoid,
  int           iteration
)
{
  int i;
  int node;
  int neighbor;
  int terminus_found;

  if (iteration == 0)
  {
    reset_search (search);
    search->total_size = total;
    reallocate_search (search);

    if ((avoid >= 0) && (avoid < total))
      search->log[avoid] = TRUE;

    for (i = 0; i < seed_total; i++)
    {
      if ((seed[i] >= 0) && (seed[i] < total))
      {
        search->log[seed[i]] = TRUE;
        search->target[search->radius][search->total[search->radius]] = seed[i];
        search->origin[search->radius][search->total[search->radius]] = NEITHER;
        search->total[search->radius]++;
      }

      else
        exit (fprintf (global.outfile,
          "ERROR depth_search: seed value inappropriate (%d).\n", seed[i]));
    }
  }

  else
    search->radius--;

  if (search->radius >= 0)
  {
    for (terminus_found = FALSE; !terminus_found;)
    {
      node = search->target[search->radius][search->count[search->radius]];
      search->target[search->radius + 1][search->total[search->radius + 1]] =
        NEITHER;

      for
      (
        search->neighbor_id[search->radius] = 0;
        (neighbor = get_neighbor
          (nodes, node, search->neighbor_id[search->radius])) != EOF;
        search->neighbor_id[search->radius]++
      )
      {
        if (flag_neighbor)
          flag_neighbor
          (
            nodes,
            node,
            search->neighbor_id[search->radius],
            !search->log[neighbor]
          );

        if (!search->log[neighbor])
        {
          search->target
            [search->radius + 1]
            [search->total[search->radius + 1]] = neighbor;
          search->origin
            [search->radius + 1]
            [search->total[search->radius + 1]] = node;
          search->log[neighbor] = TRUE;

          break;
        }
      }

      if (search->target
        [search->radius + 1]
        [search->total[search->radius + 1]] == NEITHER)
        terminus_found = TRUE;

      else
      {
        search->count[search->radius + 1] =
          search->total[search->radius + 1]++;
        search->radius++;
      }
    }

    return search->target[search->radius][search->count[search->radius]];
  }

  else
  {
    search->complete_flag = TRUE;
    return EOF;
  }
}


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

int get_search_origin
(
  SEARCH        *search,
  int           target,
  int           radius,
  int           count
)
{
  if ((target == NEITHER) && (radius == NEITHER) && (count == NEITHER))
    return search->origin[search->radius][search->count[search->radius]];

  else if (search->complete_flag)
  {
    if (target != NEITHER)
    {
      for (radius = 0; radius < search->total_size; radius++)
        for (count = 0; count < search->total[radius]; count++)
          if (search->target[radius][count] == target)
            return search->origin[radius][count];

      return NEITHER;
    }

    else if ((radius >= 0) && (radius < search->total_size) &&
      (count >= 0) && (count < search->total[radius]))
      return search->origin[radius][count];

    else
      return NEITHER;
  }

  else
    exit (fprintf (global.outfile,
      "ERROR get_search_target: search not complete.\n"));

  return TRUE;
}

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

int get_search_target
(
  SEARCH        *search,
  int           origin,
  int           radius,
  int           count
)
{
  if ((origin == NEITHER) && (radius == NEITHER) && (count == NEITHER))
    return search->target[search->radius][search->count[search->radius]];

  else if (search->complete_flag)
  {
    if ((origin >= 0) && (origin < search->total_size))
    {
      for (radius = 0; radius < search->total_size; radius++)
        for (count = 0; count < search->total[radius]; count++)
          if (search->origin[radius][count] == origin)
            return search->target[radius][count];

      return NEITHER;
    }

    else if ((radius >= 0) && (radius < search->total_size) &&
      (count >= 0) && (count < search->total[radius]))
      return search->target[radius][count];

    else if (count != NEITHER)
      return NEITHER;
  }

  else
    exit (fprintf (global.outfile,
      "ERROR get_search_target: search not complete.\n"));

  return TRUE;
}


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

int get_search_radius
(
  SEARCH        *search,
  int           target,
  int           origin
)
{
  int radius;
  int count;

  if ((origin == NEITHER) && (target == NEITHER))
    return search->radius;

  else if (search->complete_flag)
  {
    if ((target >= 0) && (target < search->total_size))
    {
      for (radius = 0; radius < search->total_size; radius++)
        for (count = 0; count < search->total[radius]; count++)
          if (search->target[radius][count] == target)
            return radius;

      return NEITHER;
    }

    if ((origin >= 0) && (origin < search->total_size))
    {
      for (radius = 0; radius < search->total_size; radius++)
        for (count = 0; count < search->total[radius]; count++)
          if (search->origin[radius][count] == origin)
            return radius;

      return NEITHER;
    }

    return NEITHER;
  }

  else
    exit (fprintf (global.outfile,
      "ERROR get_search_radius: search not complete.\n"));

  return TRUE;
}

###########################################################################
################################## SIMPLEX.H ##############################
###########################################################################
/*                                                                       */
/*                        Copyright UCSF, 1997                           */
/*                                                                       */

/*

General-purpose simplex optimizer

Define simplex_struct and simplex_score() in file containing calling routine.
```

```c
11/96 te

*/

float simplex_optimize
(
  void  *simplex,               /* Structure of info passed to scoring fn */
  float *solution,              /* Array of variables to optimize */
  int   size,                   /* Number of variables */
  float score_converge,         /* Convergence criteria */
  int   *iteration,             /* Number of simplex iterations */
  int   iteration_max,          /* Maximum number of simplex iterations */
  float (*simplex_score)        /* Scoring function */
    (void *, float *),
  float *delta                  /* Minimizer score improvement (init - fin) */
);
*************************************************************************
*********************************** SIMPLEX.C ***************************
*************************************************************************
/*
Originally transcribed by Dan Gschwend.
Converted to C, and generalized for any optimization problem by Todd Ewing.
3/96

Multidimensional minimization of the function funk(x) where x is an
n-dimensional vector, by the downhill simplex method of Nelder
and Mead.  In this case, funk is the score function.
Input is a matrix p whose n+1 rows are n-dimensional
vectors which are the vertices of the starting simplex.  Also input
is the vector y of length n+1, whose  components must be
pre-initialized to the values of funk evaluated at the n+1
vertices (rows) of p.
On output, p and y will have been reset to n+1 new points all
within score_converge of a minimum function value; iteration gives the number
of iterations taken.

Taken from Numerical Recipes:  The Art of Scientific
Computing, by Press, Flannery, Teukolsky, and Vetterling.
1986 by Cambridge University Press, C edition
pp. 292-293.

References and further reading:
  Nelder, J.A., and Mead, R. 1965, Computer Journal, vol. 7,
      p. 308.
  Yarbro, L.A., and Deming, S.N. 1974, Analytica Chim. Acta,
      vol. 73, p. 391.
  Jacoby, S.L.S., Kowalik, J.S., and Pizzo, J.T. 1972,
      Iterative Methods for Nonlinear Optimization Problems
      (Englewood Cliffs, N.J.:  Prentice-Hall).
*/

#include "define.h"
#include "utility.h"
#include "global.h"

float simplex_optimize
(
  void *simplex,                /* Structure of info passed to scoring fn */
  float *solution,              /* Array of variables to optimize */
  int size,                     /* Number of variables */
  float score_converge,         /* Convergence criteria */
  int *iteration,               /* Number of simplex iterations */
  int iteration_max,            /* Maximum number of simplex iterations */
  float (*simplex_score)        /* Scoring function */
    (void *, float *),
  float *delta                  /* Minimizer score improvement (init - fin) */
)
{
  int i, j;
  int ilo, ihi, inhi;
  float **p = NULL;
  float *pr = NULL;
  float *prr = NULL;
  float *pbar = NULL;
  float *y = NULL;
  float ypr = NULL;
  float yprr = NULL;
  float alpha = 1.0;    /* range: 0=no extrap, 1=unit step extrap, higher OK */
  float beta = 0.5;     /* range: 0=no contraction, 1=full contraction */
  float optimum;
  int replace_flag;     /* flag for whether bad vertex replaced */

/*
* Allocate space for arrays
*/
  ecalloc
  (
    (void **) &p,
    (size + 1),
    sizeof (float *),
    "simplex array",
    global.outfile
  );

  for (i = 0; i < size + 1; i++)
    ecalloc
    (
      (void **) &p[i],
      size,
      sizeof (float),
      "simplex array",
      global.outfile
    );

  ecalloc
  (
    (void **) &y,
    size + 1,
    sizeof (float),
    "simplex array",
    global.outfile
  );

  ecalloc
  (
    (void **) &pr,
    size,
    sizeof (float),
    "simplex array",
    global.outfile
  );

  ecalloc
  (
    (void **) &prr,
    size,
    sizeof (float),
    "simplex array",
    global.outfile
```

```c
  );

  ecalloc
  (
    (void **) &pbar,
    size,
    sizeof (float),
    "simplex array",
    global.outfile
  );

/*
* Begin the simplex optimization
*/
  *iteration = 0;

  do
  {
    if (*iteration == 0)
    {
/*
*     Initialize simplex array.
*       The first row contains values initialized externally.
*       All other rows contain perturbations from the first row,
*         by amounts between -1.0 and 1.0.
*     1/97 te
*/
      for (j = 0; j < size; j++)
        p[0][j] = solution[j];

      for (i = 1; i < size + 1; i++)
        for (j = 0; j < size; j++)
          p[i][j] =
            solution[j] +
            2.0 * ((float) rand () / (float) RAND_MAX - 0.5);

      for (i = 0; i < size + 1; i++)
      {
        for (j = 0; j < size; j++)
          solution[j] = p[i][j];

        y[i] = simplex_score (simplex, solution);
      }

      *delta = y[0];
    }

    else
    {
/*
*     Begin a new iteration.  Compute the vector average of all points
*     except the highest, i.e. the center of the "face" of the simplex
*     across from the high point.  We will subsequently explore the
*     ray from the high point through that center.
*/
      for (j = 0; j < size; j++)
        pbar[j] = 0.0;

      for (i = 0; i < size + 1; i++)
        if (i != ihi)
          for (j = 0; j < size; j++)
            pbar[j] += p[i][j];

/*
*     Extrapolate by a factor alpha through the face, i.e. reflect the
*     simplex from the high point.
*/
      for (j = 0; j < size; j++)
      {
        pbar[j] /= (float) size;
        solution[j] = pr[j] = (1.0 + alpha) * pbar[j] - alpha * p[ihi][j];
      }

/*
*     Evaluate the function at the reflected point.
*/
      ypr = simplex_score (simplex, solution);

      if (ypr <= y[ilo])
      {
/*
*       Gives a result better than the best point, so try an
*       additional extrapolation by a factor alpha.
*/
        for (j = 0; j < size; j++)
          solution[j] = prr[j] = (1.0 + alpha) * pr[j] - alpha * pbar[j];

/*
*       Check out the function there.
*/
        yprr = simplex_score (simplex, solution);

        if (yprr < y[ilo])
        {

/*
*         The additional extrapolation succeeded, and replaces
*         the high point.
*/
          for (j = 0; j < size; j++)
            p[ihi][j] = prr[j];

          y[ihi] = yprr;
        }

        else
        {

/*
*         The additional extrapolation failed, but we can still use
*         the reflected point
*/
          for (j = 0; j < size; j++)
            p[ihi][j] = pr[j];

          y[ihi] = ypr;
        }
      }

      else if (ypr >= y[inhi])
      {
/*
*       The reflected point is worse than the second-highest.
*       If it's better than the highest, then replace the highest.
*/
        replace_flag = FALSE;

        if (ypr < y[ihi])
        {
```

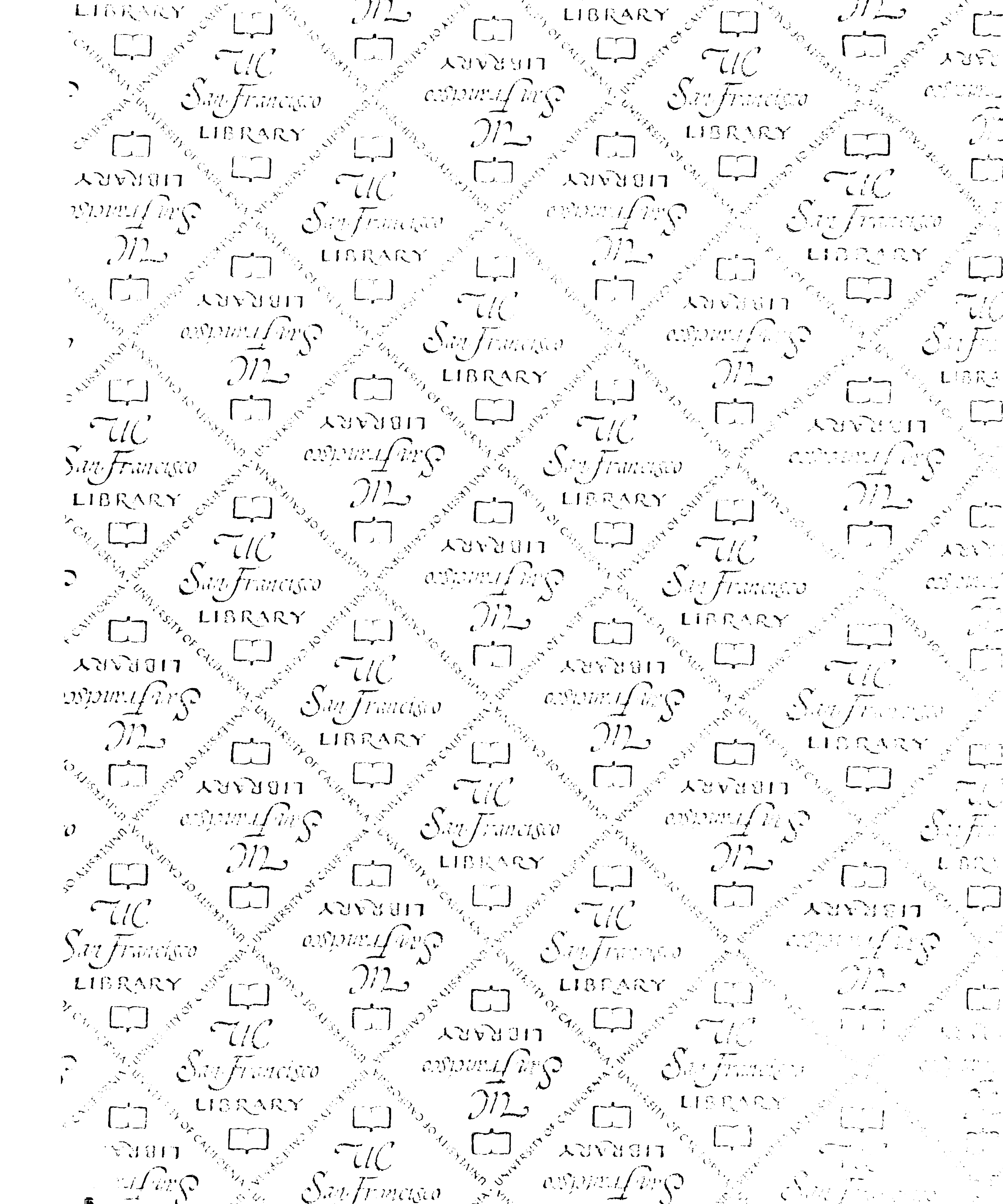
```
          for (j = 0; j < size; j++)
            p[ihi][j] = pr[j];

          y[ihi] = ypr;
          replace_flag = TRUE;
        }

/*
*       But look for an intermediate lower point, in other words,
*       perform a contraction of the simplex along one dimension, then
*       evaluate the function.
*/
        for (j = 0; j < size; j++)
          solution[j] = prr[j] = beta * p[ihi][j] + (1.0 - beta) * pbar[j];

        yprr = simplex_score (simplex, solution);

        if (yprr < y[ihi])
        {

/*
*         Contraction gives an improvement, so accept it.
*/
          for (j = 0; j < size; j++)
            p[ihi][j] = prr[j];

          y[ihi] = yprr;
          replace_flag = TRUE;
        }

        if (replace_flag == FALSE)
        {
/*
*         Can't seem to get rid of that high point.  Better contract
*         around the lowest (best) point.
*/
          for (i = 0; i < size + 1; i++)
            if (i != ilo)
          {
            for (j = 0; j < size; j++)
              solution[j] = p[i][j] = pr[j] = 0.5 * (p[i][j] + p[ilo][j]);

            y[i] = simplex_score (simplex, solution);
          }
        }
      }

      else
      {
/*
*       We arrive here if the original reflection gives a middling point.
*       Replace the old high point and continue.
*/
        for (j = 0; j < size; j++)
          p[ihi][j] = pr[j];

        y[ihi] = ypr;
      }
    }
/*
*   Identify the best and worst vertices in the current simplex.
*   We must determine which point is the highest (worst), next-
*   highest, and lowest (best).
*/
    ilo = 0;

    if (y[0] > y[1])
    {
      ihi = 0;
      inhi = 1;
    }

    else
    {
      ihi = 1;
      inhi = 0;
    }

/*
*   Loop over points in the simplex
*/
/*
    if (size == 6)
    fprintf (global.outfile, "  simplex scores:");
*/

    for (i = 0; i < size + 1; i++)
    {
/*
    if (size == 6)
      fprintf (global.outfile, " %g", y[i]);
*/

      if (y[i] < y[ilo])
        ilo = i;

      if (y[i] > y[ihi])
      {
        inhi = ihi;
        ihi = i;
      }

      else if (y[i] > y[inhi])
      {
        if (i != ihi)
          inhi = i;
      }
    }
/*
    if (size == 6)
    fprintf (global.outfile, "\n");
*/
  } while
      ((++(*iteration) < iteration_max) &&
      (ABS (y[ihi] - y[ilo]) > score_converge));

/*
* Store the best scoring vertex of the simplex
*/
  for (i = 0; i < size; i++)
    solution[i] = p[ilo][i];

  optimum = y[ilo];

  *delta -= optimum;

/*
* Free up space allocated for simplex arrays
*/
  for (i = 0; i < size + 1; i++)
    efree ((void **) &p[i]);

  efree ((void **) &p);
  efree ((void **) &y);
  efree ((void **) &pr);
  efree ((void **) &prr);
  efree ((void **) &pbar);

/*
* Return the best scoring vertex
  fprintf (global.outfile, "Finished simplex, best score %g\n", optimum);
*/
  return optimum;
}

*********************************************************************************
********************************** TRANSFORM.H **********************************
*********************************************************************************
/*                                                                      */
/*                        Copyright UCSF, 1997                          */
/*                                                                      */

/*
Written by Todd Ewing
10/95
*/
/*
Routines defined in transform.c, that are called by outside functions
*/

int orient_molecule
(
  MOLECULE *target,
  MOLECULE *current,
  MOLECULE *mol_ref,
  MOLECULE *mol_conf,
  MOLECULE *mol_ori
);

void transform_molecule
(
  MOLECULE *mol_ori,
  MOLECULE *mol_ref
);

void rigid_transform
(
  MOLECULE *mol_ori,
  MOLECULE *mol_ref
);

void torsion_transform
(
  MOLECULE *molecule,
  int torsion_id
);

void rotate_atoms
(
  MOLECULE *molecule,
  XYZ matrix[3],
  int origin,
  int atom
);

int get_matrix_from_quaternion (XYZ [3], XYZ, int);
int get_quaternion_from_matrix (XYZ, int *, XYZ [3]);
float compute_torsion (MOLECULE *, int);
void overall_rotation (MOLECULE *, MOLECULE *);
void overall_translation (MOLECULE *, MOLECULE *);

/*
Fortran routines in transformf.f
*/
int orient_gk_(int *, XYZ *, XYZ *, XYZ, XYZ *, XYZ, int *);
void transform_(int *, XYZ *, XYZ, XYZ *, XYZ, XYZ *);
void transform_atom_ (XYZ, XYZ *, XYZ);

*********************************************************************************
********************************** TRANSFORM.C **********************************
*********************************************************************************
/*                                                                      */
/*                        Copyright UCSF, 1997                          */
/*                                                                      */

/*
Written by Todd Ewing
10/95
*/
#include "define.h"
#include "utility.h"
#include "mol.h"
#include "global.h"
#include "vector.h"
#include "transform.h"


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

int orient_molecule
(
  MOLECULE *target,
  MOLECULE *current,
  MOLECULE *mol_ref,
  MOLECULE *mol_conf,
  MOLECULE *mol_ori
)
{
  static XYZ rotation_matrix[3];

/*
* Calculate rotation matrix and translation vector.
* 6/95 te
*/
  if (!orient_gk_
  (
    &target->total.atoms,
    target->coord,
    current->coord,
    mol_ori->transform.com,
    rotation_matrix,
    mol_ori->transform.translate,
    &mol_ori->transform.refl_flag
  ))
    return FALSE;

/*
* Update transform records
```

```
* 10/96 te
*/
  get_quaternion_from_matrix
   (
    mol_ori->transform.rotate,
    &mol_ori->transform refl_flag,
    rotation_matrix
   );

  mol_ori->transform.trans_flag = TRUE;
  mol_ori->transform.rot_flag = TRUE;

/*
* If a previous transformation is present, then update this info
* 10/96 te
*/
  if (mol_conf->transform.rot_flag)
    overall_rotation (mol_ori, mol_conf);

  if (mol_conf->transform.trans_flag)
    overall_translation (mol_ori, mol_conf);

/*
* Transform the molecule coordinates
* 10/96 te
*/
  transform_molecule (mol_ori, mol_ref);

  return TRUE;
}


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

void transform_molecule
(
  MOLECULE *mol_trans,
  MOLECULE *mol_ref
)
{
  int i;

  mol_trans->transform.flag = TRUE;

/*
* Perform a rigid-body rotation/translation
* 12/96 te
*/
  if ((mol_trans->transform.trans_flag == TRUE) ||
    (mol_trans->transform.rot_flag == TRUE))
    rigid_transform (mol_trans, mol_ref);

/*
* Perform a rotation about each rotatable bond vector
* 12/96 te
*/
  if (mol_trans->transform.tors_flag == TRUE)
    for (i = 0; i < mol_trans->total.torsions; i++)
      torsion_transform (mol_trans, i);
}


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

void rigid_transform
(
  MOLECULE *mol_trans,
  MOLECULE *mol_ref
)
{
  static XYZ rotation[3];

  mol_trans->transform.flag = TRUE;

/*
* Calculate a rotation matrix from the quaternion
* 12/96 te
*/
  get_matrix_from_quaternion
   (
    rotation,
    mol_trans->transform.rotate,
    mol_trans->transform.refl_flag
   );

  transform_
   (
    &mol_trans->total.atoms,
    mol_ref->coord,
    mol_trans->transform.com,
    rotation,
    mol_trans->transform.translate,
    mol_trans->coord
   );
}


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

void torsion_transform
(
  MOLECULE      *molecule,
  int           torsion_id
)
{
  int i;
  static XYZ bond_matrix[3];
  static XYZ bond_vector;
  static float difference;

  void rotateaxis (float, XYZ, XYZ *);

  molecule->transform.flag = TRUE;

/*
* Compute current torsion angle and difference
* 12/96 te
*/
  molecule->torsion[torsion_id].current_angle =
    compute_torsion (molecule, torsion_id);

  difference =
    molecule->torsion[torsion_id].target_angle -
    molecule->torsion[torsion_id].current_angle;

/*
* If difference is significant, then rotate bond
* 12/96 te
*/
  if (NON_ZERO (difference))
  {
    for (i = 0; i < 3; i++)
      bond_vector[i] =
        molecule->coord[molecule->torsion[torsion_id].target][i] -
        molecule->coord[molecule->torsion[torsion_id].origin][i];

    rotateaxis (difference, bond_vector, bond_matrix);

    rotate_atoms
    (
      molecule,
      bond_matrix,
      molecule->torsion[torsion_id].target,
      molecule->torsion[torsion_id].target
    );

    molecule->torsion[torsion_id].current_angle =
      molecule->torsion[torsion_id].target_angle;
  }
}


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

Subroutine that calculate a rotation matrix, given a vector axis
and an angle of rotation.

10/95 ys

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

void rotateaxis (float phi, XYZ vect, XYZ rot[3])
{
  float del1,del2,del3;
  float vect_length;
  float vcosx,vcosy,vcosz;

  del1=1-cos(phi);
  del2=sin(phi);
  del3=cos(phi);

  vect_length=sqrt(vect[0]*vect[0]+
                   vect[1]*vect[1]+
                   vect[2]*vect[2]);
  vcosx= vect[0]/vect_length;
  vcosy= vect[1]/vect_length;
  vcosz= vect[2]/vect_length;

  rot[0][0]=vcosx*vcosx*del1 + del3;
  rot[0][1]=vcosx*vcosy*del1 + vcosz*del2;
  rot[0][2]=vcosx*vcosz*del1 - vcosy*del2;
  rot[1][0]=vcosy*vcosx*del1 - vcosz*del2;
  rot[1][1]=vcosy*vcosy*del1 + del3;
  rot[1][2]=vcosy*vcosz*del1 + vcosx*del2;
  rot[2][0]=vcosz*vcosx*del1 + vcosy*del2;
  rot[2][1]=vcosz*vcosy*del1 - vcosx*del2;
  rot[2][2]=vcosz*vcosz*del1 + del3;

  return;
}

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

void rotate_atoms
(
  MOLECULE      *molecule,
  XYZ           bond_matrix[3],
  int           origin,
  int           atom
)
{
  int i;

/*
  fprintf (global.outfile, "orig %s atom %s\n",
    molecule->atom[origin].name,
    molecule->atom[atom].name);
*/

  transform_atom_
  (
    molecule->coord[atom],
    bond_matrix,
    molecule->coord[origin]
  );

  for (i = 0; i < molecule->atom[atom].neighbor_total; i++)
    if (molecule->atom[atom].neighbor[i].out_flag == TRUE)
      rotate_atoms
        (molecule, bond_matrix, origin, molecule->atom[atom].neighbor[i].id);
}


/* //////////////////////////////////////////////////////////////////// */
/*
Function to compute a rotation matrix from quaternion values.
For discussion of this technique, see "Computer Simulation of Liquids"
by M.P. Allen and D.J. Tildesley from Oxford Science Publishers, 1987
10/96 te
*/

int get_matrix_from_quaternion
(
  XYZ   m[3],           /* rotation matrix */
  XYZ   qin,            /* input independent quaternion elements */
  int   refl            /* reflection flag */
)
{
  int   i;              /* iterater */
  float qn;             /* dependent quaternion element, q-naught */
  float qn2;            /* square of q-naught */
  XYZ   q;              /* independent quaternion elements */
  XYZ   q2;             /* square of independent quaternion elements */
  float sum2;           /* sum of squares of independent q values */
  float sum;            /* sum of independent q values */

/*
* Check that each q-value is between -1.0 and 1.0.
* If not, remap into this range using wrap-around.
* Compute q-squared values and their sum
* 10/96 te
*/
  for (i = 0, sum2 = 0.0; i < 3; i++)
  {
    q[i] = qin[i];

    if (q[i] > 1.0)
      q[i] = fmod (q[i] + 1.0, 2.0) - 1.0;
```

```
    else if (q[i] < -1.0)
      q[i] = fmod (q[i] - 1.0, 2.0) + 1.0;

    sum2 += q2[i] = SQR (q[i]);
  }

/*
* If the sum-of-squares is less than 1.0, compute q-naught
* 10/96 te
*/
  if (sum2 < 1.0)
  {
    qn2 = 1.0 - sum2;
    qn = sqrt (qn2);
  }

/*
* If the sum is 1.0, set q-naught to zero
* 10/96 te
*/
  else if (sum2 == 1.0)
    qn = qn2 = 0.0;

/*
* Otherwise, renormalize q-values and set q-naught to zero
* 10/96 te
*/
  else
  {
    for (i = 0; i < 3; i++)
      q2[i] /= sum2;

    for (i = 0, sum = sqrt (sum2); i < 3; i++)
      q[i] /= sum;

    qn = qn2 = 0.0;
  }

/*
* Compute rotation matrix elements
* 10/96 te
*/
  m[0][0] = qn2 + q2[0] - q2[1] - q2[2];
  m[0][1] = 2.0 * (q[0] * q[1] + qn * q[2]);
  m[0][2] = 2.0 * (q[0] * q[2] - qn * q[1]);

  m[1][0] = 2.0 * (q[0] * q[1] - qn * q[2]);
  m[1][1] = qn2 - q2[0] + q2[1] - q2[2];
  m[1][2] = 2.0 * (q[1] * q[2] + qn * q[0]);

  m[2][0] = 2.0 * (q[0] * q[2] + qn * q[1]);
  m[2][1] = 2.0 * (q[1] * q[2] - qn * q[0]);
  m[2][2] = qn2 - q2[0] - q2[1] + q2[2];

/*
* If a reflection is required, then update the matrix
* 10/96 te
*/
  if (refl == TRUE)
    for (i = 0; i < 3; i++)
      m[i][2] *= -1.0;

  return TRUE;
}

/* ///////////////////////////////////////////////////////////////////////// */
/*
Function to compute the quaternion values from a rotation matrix
For discussion of this technique, see 'Computer Simulation of Liquids'
by M.P. Allen and D.J. Tildesley from Oxford Science Publishers, 1987
10/96 te
*/

int get_quaternion_from_matrix
(
  XYZ   q,              /* independent quaternion elements */
  int   *refl,          /* reflection flag */
  XYZ   m[3]            /* rotation matrix */
)
{
  int   i;              /* iterater */
  int   max;            /* quaternion element with greatest magnitude */
  float max_val;        /* value of greatest quaternion element */
  float qn;             /* dependent quaternion element, q-naught */
  float qn2;            /* square of q-naught */
  XYZ   q2;             /* square of independent quaternion elements */
  float det;            /* determinant of input matrix */

/*
* Compute determinant of matrix
* 10/96 te
*/
  det =
    m[0][0] * (m[1][1] * m[2][2] - m[2][1] * m[1][2]) -
    m[0][1] * (m[1][0] * m[2][2] - m[2][0] * m[1][2]) +
    m[0][2] * (m[1][0] * m[2][1] - m[2][0] * m[1][1]);

/*
* If the determinant is negative, then set the reflection flag and
* invert matrix
* 10/96 te
*/
  if (det < 0.0)
  {
    *refl = TRUE;

    for (i = 0; i < 3; i++)
      m[i][2] *= -1.0;
  }

  else if (*refl != NEITHER)
    *refl = FALSE;

/*
* Calculate q-squared values (times 4)
* 10/96 te
*/
  qn2   = 1.0 + m[0][0] + m[1][1] + m[2][2];
  q2[0] = 1.0 + m[0][0] - m[1][1] - m[2][2];
  q2[1] = 1.0 - m[0][0] + m[1][1] - m[2][2];
  q2[2] = 1.0 - m[0][0] - m[1][1] + m[2][2];

/*
* Identify the largest q-squared value
* 10/96 te
*/
  for (i = max = 0, max_val = qn2; i < 3; i++)
    if (q2[i] > max_val)
    {
      max_val = q2[i];
      max = i + 1;
    }

/*
* Compute the other q-values based on the positive root
* of the largest q-squared value
* 10/96 te
*/
  switch (max)
  {
    case 0:
    {
      qn   = 0.5 * sqrt (qn2);

      q[0] = 0.25 * (m[1][2] - m[2][1]) / qn;
      q[1] = 0.25 * (m[2][0] - m[0][2]) / qn;
      q[2] = 0.25 * (m[0][1] - m[1][0]) / qn;

      break;
    }

    case 1:
    {
      q[0] = 0.5 * sqrt (q2[0]);

      qn   = 0.25 * (m[1][2] - m[2][1]) / q[0];
      q[1] = 0.25 * (m[0][1] + m[1][0]) / q[0];
      q[2] = 0.25 * (m[0][2] + m[2][0]) / q[0];

      break;
    }

    case 2:
    {
      q[1] = 0.5 * sqrt (q2[1]);

      qn   = 0.25 * (m[2][0] - m[0][2]) / q[1];
      q[0] = 0.25 * (m[1][0] + m[0][1]) / q[1];
      q[2] = 0.25 * (m[2][1] + m[1][2]) / q[1];

      break;
    }

    case 3:
    {
      q[2] = 0.5 * sqrt (q2[2]);

      qn   = 0.25 * (m[0][1] - m[1][0]) / q[2];
      q[0] = 0.25 * (m[2][0] + m[0][2]) / q[2];
      q[1] = 0.25 * (m[2][1] + m[1][2]) / q[2];

      break;
    }

    default:
    {
      printf ("Unable to find q2 value > 0\n");
      return FALSE;
    }
  }

/*
* Reset q-values so that q-naught is always positive
* 10/96 te
*/
  if (qn < 0.0)
    for (i = 0; i < 3; i++)
      q[i] *= -1.0;

  return TRUE;
}


/* ///////////////////////////////////////////////////////////////////////
Compute current angle of specified bond
10/96 te
/////////////////////////////////////////////////////////////////////// */

float compute_torsion (MOLECULE *molecule, int torsion_id)
{
  if
  (
    (molecule->torsion[torsion_id].origin_neighbor == NEITHER) ||
    (molecule->torsion[torsion_id].origin == NEITHER) ||
    (molecule->torsion[torsion_id].target == NEITHER) ||
    (molecule->torsion[torsion_id].target_neighbor == NEITHER)
  )
    exit (fprintf (global.outfile,
      "ERROR compute_torsion: Torsion atoms undefined in %s\n",
      molecule->info.name));

  return
    dihed
    (
      molecule->coord[molecule->torsion[torsion_id].origin_neighbor],
      molecule->coord[molecule->torsion[torsion_id].origin],
      molecule->coord[molecule->torsion[torsion_id].target],
      molecule->coord[molecule->torsion[torsion_id].target_neighbor]
    );
}


/* ///////////////////////////////////////////////////////////////////////
Compute a new overall rotation
10/96 te
/////////////////////////////////////////////////////////////////////// */

void overall_rotation
(
  MOLECULE *mol_new,
  MOLECULE *mol_orig
)
{
  static XYZ rot_orig[3], rot_new[3], rot_temp[3];

  get_matrix_from_quaternion
    (rot_orig, mol_orig->transform.rotate, mol_orig->transform.refl_flag);
  get_matrix_from_quaternion
    (rot_new, mol_new->transform.rotate, mol_new->transform.refl_flag);
  mmult3 (rot_orig, rot_new, rot_temp);
  get_quaternion_from_matrix
    (mol_new->transform.rotate, &mol_new->transform.refl_flag, rot_temp);
}


/* ///////////////////////////////////////////////////////////////////////
Compute a new overall translation
```

```
10/96 te
///////////////////////////////////////////////////////////////////// */

void overall_translation
(
  MOLECULE *mol_new,
  MOLECULE *mol_orig
)
{
  int i;

  for (i = 0; i < 3; i++)
    mol_new->transform.translate[i] += mol_orig->transform.translate[i];
}

*******************************************************************************
*********************************** UTILITY.H *********************************
*******************************************************************************
/*                                                                   */
/*                        Copyright UCSF, 1997                       */
/*                                                                   */

/*
Written by Todd Ewing
10/95
*/
FILE *efopen (const char *, const char *, FILE *);
FILE *rfopen (const char *, const char *, FILE *);
void efclose (FILE **);
void rremove (const char *, FILE *);

void efread (void *, size_t , size_t , FILE *);
void efwrite (void *, size_t , size_t , FILE *);

void emalloc (void **, const int, const char *, FILE *);
void ecalloc (void **, const int, const int, const char *, FILE *);
void erealloc (void **, const int, const char *, FILE *);
void efree (void **);

char *subst_char (char *line, char old, char new);
char *strip_char (char *line, char character);
char *strip_newline (char *line);
char *white_line (char *line);

char *vstrcpy (char **, const char *);
char *vstrcat (char **, const char *);
char *vchrcat (char **, const char);
char *vfgets (char **, FILE *);

char *find_record (char **, const char *, FILE *);
*******************************************************************************
*********************************** UTILITY.C *********************************
*******************************************************************************
/*                                                                   */
/*                        Copyright UCSF, 1997                       */
/*                                                                   */

/*
Written by Todd Ewing
10/95
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "define.h"
#include "global.h"

FILE *efopen
(
  const char *file_name,
  const char *mode,
  FILE *message_stream
)
{
  FILE  *file;

  file = fopen (file_name, mode);

  if (file == NULL)
    exit (fprintf (message_stream,
      "ERROR efopen: Unable to open %s.  Terminating execution.\n",
      file_name));

  return file;
}

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

FILE *rfopen
(
  const char *file_name,
  const char *mode,
  FILE *message_stream
)
{
  FILE  *file;

  file = fopen (file_name, mode);

  if (file == NULL)
  {
    fprintf (message_stream,
      "WARNING rfopen: Unable to open %s, but will continue trying.\n",
      file_name);
    fflush (message_stream);
    while (file == NULL)
      file = fopen (file_name, mode);
  }

  return file;
}

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

void efclose (FILE **file)
{
  if (fclose (*file) != 0)
    exit (fprintf (global.outfile,
      "ERROR efclose: Unable to close file.\n"));

  *file = NULL;
}

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

void efread (void *ptr, size_t size, size_t nobj, FILE *stream)
{
  if (fread (ptr, size, nobj, stream) != nobj)
    exit (fprintf (stderr,
      "ERROR efread: Unable to read proper number of objects.\n"));
}

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

void rremove
(
  const char *file_name,
  FILE *message_stream
)
{
  if (remove (file_name))
  {
    fprintf (message_stream,
      "WARNING rremove: Unable to remove %s, but will continue trying.\n",
      file_name);
    fflush (message_stream);
    while (remove (file_name));
  }
}

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

void efwrite (void *ptr, size_t size, size_t nobj, FILE *stream)
{
  if (fwrite (ptr, size, nobj, stream) != nobj)
    exit (fprintf (stderr,
      "ERROR efwrite: Unable to write proper number of objects.\n"));
}

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

void emalloc
(
  void **ptr,
  const int size,
  const char *name,
  FILE *message_stream
)
{
  if (*ptr != NULL)
    exit (fprintf (message_stream,
      "ERROR emalloc: %s not freed prior to allocation.\n", name));

  if (size > 0)
  {
    *ptr = (void *) malloc (size);

    if (*ptr == NULL)
      exit (fprintf (message_stream,
        "ERROR emalloc: Insufficient memory for %s.\n", name));
  }
}

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

void ecalloc
(
  void **ptr,
  const int objectnum,
  const int objectsize,
  const char *name,
  FILE *message_stream
)
{
  if (*ptr != NULL)
    exit (fprintf (message_stream,
      "ERROR ecalloc: %s not freed prior to allocation.\n", name));

  if (objectnum * objectsize > 0)
  {
    *ptr = (void *) calloc (objectnum, objectsize);

    if (*ptr == NULL)
      exit (fprintf (message_stream,
        "ERROR ecalloc: Insufficient memory for %s.\n", name));
  }
}

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

void erealloc
(
  void **ptr,
  const int size,
  const char *name,
  FILE *message_stream
)
{
  void *new_ptr;

  new_ptr = (void *) realloc (*ptr, size);

  if ((size > 0) && (new_ptr == NULL))
    exit (fprintf (message_stream,
      "ERROR erealloc: Insufficient memory for %s.\n", name));

  else
    *ptr = new_ptr;
}

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

void efree (void **ptr)
{
  if (ptr == NULL)
    exit (fprintf (global.outfile,
      "ERROR efree: NULL address passed\n"));

  else if (*ptr != NULL)
  {
    free (*ptr);
    *ptr = NULL;
  }
}

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

char *subst_char (char *line, char old, char new)
```

```
{
  char *ptr;

  if (line)
    for (ptr = line; *ptr != '\0'; ptr++)
      if (*ptr == old)
        *ptr = new;

  return line;
}

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

char *strip_char (char *line, char character)
{
  if ((line) && (strchr (line, character)))
    strcpy (strchr (line, character), "");

  return line;
}

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

char *strip_newline (char *line)
{
  if (line[strlen (line) - 1] == '\n')
    line[strlen (line) - 1] = 0;

  return line;
}

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

char *white_line (char *line)
{
  int i;

  for (i = 0; i < strlen (line); i++)
    if (isspace (line[i]))
      line[i] = ' ';

  return line;
}

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

char *vstrcpy (char **copy_ptr, const char *original)
{
  if (original)
  {
    if (*copy_ptr)
    {
      if (strlen (original) > strlen (*copy_ptr))
      {
        efree ((void **) copy_ptr);
        ecalloc
        (
          (void **) copy_ptr,
          strlen (original) + 1,
          sizeof (char),
          original,
          stdout
        );
      }

      else
        memset (*copy_ptr, 0, strlen (*copy_ptr));
    }

    else
      ecalloc
      (
        (void **) copy_ptr,
        strlen (original) + 1,
        sizeof (char),
        original,
        stdout
      );

    strcpy (*copy_ptr, original);
  }

  else if (*copy_ptr)
  {
    efree ((void **) copy_ptr);
    *copy_ptr = NULL;
  }

  return *copy_ptr;
}

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

char *vstrcat (char **base_ptr, const char *addition)
{
  char *temp = NULL;

  if (addition)
  {
    if (*base_ptr)
    {
      temp = *base_ptr;
      *base_ptr = NULL;

      ecalloc
      (
        (void **) base_ptr,
        strlen (temp) + strlen (addition) + 1,
        sizeof (char),
        addition,
        stdout
      );

      strcpy (*base_ptr, temp);
      efree ((void **) &temp);
    }

    else
      ecalloc
      (
        (void **) base_ptr,
        strlen (addition) + 1,
        sizeof (char),
        addition,
        stdout
      );

    strcat (*base_ptr, addition);
  }

  return *base_ptr;
}

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

char *vchrcat (char **base_ptr, const char addition)
{
  char *temp = NULL;

  if (*base_ptr)
  {
    temp = *base_ptr;
    *base_ptr = NULL;

    ecalloc
    (
      (void **) base_ptr,
      strlen (temp) + 2,
      sizeof (char),
      "vchrcat intermediate",
      stdout
    );

    strcpy (*base_ptr, temp);
    efree ((void **) &temp);
  }

  else
    ecalloc
    (
      (void **) base_ptr,
      2,
      sizeof (char),
      "vchrcat intermediate",
      stdout
    );

  (*base_ptr)[strlen (*base_ptr)] = addition;

  return *base_ptr;
}

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

char *vfgets (char **line_ptr, FILE *file)
{
  STRING100 line = "";
  char *return_value = NULL;

  efree ((void **) line_ptr);

  while (return_value = fgets (line, 100, file))
  {
    vstrcat (line_ptr, line);

    if (line[strlen (line) - 1] == '\n')
      break;
  }

  if (return_value == NULL)
    return NULL;

  else
    return *line_ptr;
}

/* //////////////////////////////////////////////////////////////////
Function to advance into a file until a particular record is found.

Return value:
  NULL: if end of file is reached before finding record
  char *: if the record is found, then return the record
////////////////////////////////////////////////////////////////// */

char *find_record
(
  char **line,
  const char *record,
  FILE *file
)
{
  char *return_value = NULL;

  while (((return_value = vfgets (line, file)) != NULL) &&
    strncmp (*line, record, strlen (record)));

  return return_value;
}
```



UCSF LIBRARY

For reference

Not to be taken from the room.

UNIVERSITY OF CALIFORNIA

UC San Francisco LIBRARY