Title
Customizing Scoring Functions in Molecular Docking

Author
Pham, Tuan

Publication Date
2007-12-20

Peer reviewed|Thesis/dissertation

Customizing Scoring Functions In Molecular Docking

by

Tuan Anh Pham

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Biological and Medical Informatics

in the

GRADUATE DIVISION

of the

Acknowledgements

First the required content:

The text of Chapter 5 and Chapter 6 of this dissertation is a revised reprint of the material as it appears in the Journal of Medicinal Chemistry and Journal of Computer-Aided Molecular Design, respectively. The coauthor Ajay Jain listed in these publications directed and supervised the research that forms the basis for the dissertation.

Now the good stuff – recognizing those who lent me a hand in this epic journey called graduate school:

First and foremost, I must thank my thesis advisor, Ajay Jain. I feel incredibly fortunate to have found his lab and mentorship. In a confusing time when I was wondering why I was in grad school, he gave me direction and focus. From the outset, our minds were aligned: we both wanted to devote our time to real-world problems. As an engineer with a Computer Science degree, I feel my interest peaks when I can perceive people making use of my work. In that sense, drug discovery and molecular docking were the perfect fit for me in this program. Having that fit between student and advisor has been a pleasure. Thank you, Ajay.

My steady progression from orals to thesis defense was made possible by my thesis committee. Together with Ajay, Andrej Sali and Mark Segal offered excellent insights that helped me forge ahead with my graduate career. Thank you, Andrej and Mark for letting me graduate.

BMI continues to grow and evolve through the hard work and effort of our directors, Patsy Babbitt & Tom Ferrin. Thanks, Tom. Patsy is so incredibly busy; she always seems on the go from one meeting to the next. Yet every time she sees me, she always stops for a second just to say hi. It's the little things like that which make her a heartwarming figure to me and this program. Also, thank you Patsy for letting me squat in your lab in QB3 the last couple of years. I had more pictures up at my squattin' desk in the Babbitt Lab than in my own office at the CRI. Granted, some of those were to fend off other potential squatters, but still. You always made me feel welcome. Thank you, Patsy.

The program runs like a well-oiled machine because of our excellent program coordinators. When I first applied, Barbara Paschke was the face of BMI for me. She was the one who encouraged me to apply to the Ph.D. program. Thanks for the nudge, Barbara. Denise picked up the reins admirably after a year of chaos when Barbara left. Good work, Denise! Then Becca Brown took the job to new heights. So high, she was scooped up by the another program, Biophysics. I'm glad that I get to graduate under her watch. To the Appreciativest Valorwoman, you made my volleyball dreams come true. Our newest coordinator Julia will no doubt continue this legacy of excellence.

Friends are what make the long days of graduate school enjoyable. There's one group in particular that I have to give mad props to: Ben "Sleeveless Polo" Lauffer, Hesper "Sista Assista" Rego, Holly "The Recruiter" Atkinson, Kris "Thunder" Kuchenbecker, Laura "Kickeyball" Lavery, Liz "Smiley" Clarke, and Ranyee "Clutch" Chiang. Thank you, Thunderforce Volleyball for all the Championship Shorts.

My lab mates were my brothers and sisters in science; they made it worthwhile to come into work. As postdocs, Jane Fridlyand and Taku Tokuyasu represented the finish line of where I wanted to be. My office mate, Chris Kingsley introduced me into the sordid world of professional cycling, among other interesting things on the interwebs. Lawrence Hon showed a novice on the tennis court what a sweet stroke looks like. His love of side gadget projects was also highly contagious. When Barbara Novak turned it down, I became Lab Manager by default. We hung out in our office and shared many a lunch. When we weren't shooting the breeze, we encouraged each other to get a move on towards graduation. When James Langham joined the lab a few months before my graduation, I was finally no longer the baby in the lab – now he had the youngest tenure. James, you were a fantastic, fantastic resource for me while I was writing. Thanks to you, I now know more about the currency markets than I ever did. To Ann, Rebecca, and Hannah you made the Jain Lab feel like family. That's the highest compliment I can give. Thank you, everyone in the Jain Lab.

Shout outs to my fellow classmates! Mark Peterson, it's a shame your body broke down in shambles because you're the best tennis player alive (that I know). Mike Kim taught me everything I know about computers. Libusha Kelly, keep it down will ya? I can hear that happy laugh across our entire floor. Juanita Li, what a voice! John Chuang, we sure watched some terrible movies together. Simona Carini, you opened up your beautiful homes to us, but it's the tiramisu I'll always treasure. Christina Chaivorapol runs so much faster than I do it's sorta not fair. Ranyee Chiang, you're not that mean after all. All good people; you made grad school for me. Thanks guys.

I am who I am because of my family. Mom & Dad, thank you for letting me monopolize the dining room table all day and reminding me to eat every time I came over. Sometimes you just need to blow off steam and not think about work anymore – Lee was my outlet for tennis, videogames, and fun, period. Thanks, bro. Since I can remember, education has always been a priority. They can take away everything you have, but they can't take away your education. That's what my Bo taught me. Thank you, Bo. My mother taught me the importance of persevering through hardship to attain my goals. She also taught me to do things with compassion. Thank you, Me. You can't ask for a better support network than this Phamily. Love you guys.

And lastly to my Wifey – She is the bedrock upon which I build this life. This work would never have been completed without her encouragement. Through the highs and lows of the last five years and one quarter, she has stood by me. She's celebrated with me, giving me the Thunder Punch. She's even been to work with me. We have traveled the world together. We have traveled this road together. It is to her with love that I dedicate this dissertation. Thank you, An.

Abstract

Customizing Scoring Functions in Molecular Docking

Tuan A. Pham

In drug discovery, where a model of the protein structure is known, molecular docking is a well-established approach for predictive modeling. Docking algorithms utilize a search strategy for exploring ligand poses within an active site and a scoring function for evaluating the poses. This dissertation explores improvements to both aspects of docking, emphasizing the use of machine learning methods for improving scoring functions. The work is built upon an extensible software platform for modeling molecular interactions, called Surflex.

Performance evaluation has been carried out on benchmarks that have been made publicly available, some of which were constructed in the course of this work. The novel tool pdbgrind, developed as part of the infrastructure for this work, was used to generate the large amount of data necessary to create adequate training and test sets. While the dissertation focuses most strongly on the scoring function problem in docking, some effort was also spent on the tightly coupled problem of search, and modest improvements were shown by enhancing Surflex's representation of protein active sites.

The bulk of the work describes improvements to empirical scoring functions for protein-ligand interactions. This dissertation demonstrates a robust method for tuning scoring function parameters to improve modeling of known binding phenomena. Penalties for inter-atomic overlap and same-charge repulsion were learned using

synthetic negative data. The new function was shown to be equivalent or better than the original function in terms of screening utility on a large and diverse benchmark. This approach was generalized for the entire scoring function to support the use of multiple constraints in refining scoring function parameters. Using the constraint-based optimization procedure, users can exploit multiple types of data to customize functions to suit a particular task or a particular protein target or family of targets. Significant improvement to screening utility was shown using data typical of applications in docking.

The main contributions of this dissertation are generalizable methods for generating and exploiting multiple types of data in refining scoring functions for docking. The approaches can be extended to other areas, including quantitative structure activity prediction or protein folding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Information – storing it, processing it, leveraging it – all to generate something valuable is critical for accelerating biological discoveries. In the last decade, there has been an explosion of biological data. We have seen 633 newly sequenced genomes;[1] an additional 40,566 deposited structures in the Protein Data Bank;[2] and 3,423 combinatorial libraries synthesized.[3] Computers are a necessary and invaluable tool for generating and testing hypotheses with this amount of data. Direct experimental testing is often expensive, so an *in silico* approach can often be an inexpensive means of efficiently reducing the requirements of direct experimentation.

This bioinformatics doctoral dissertation lies at the interstice of computer science and drug development. Here, we apply machine learning techniques to molecular data to improve methods for docking ligands to proteins. One unique aspect of machine learning in the docking arena is that the *precise* relationship of a ligand bound to a protein is not known. This uncertainty manifests as free or only partially constrained variables that define the conformation and alignment of a ligand to a protein. Formally, this represents a problem of multiple instance learning.[4, 5] Though this work has applications within drug

development, the approach described herein may be useful in any multiple instance learning problem.

Refining methods for docking is best done with large datasets. Chapter 3 investigates the complexities of automating the processing of molecular data used throughout this work. In Chapter 4, we will then consider how to search the endless ways in which a ligand might bind into a protein active site. Once bound, a properly tuned scoring function can tell us not only the strength of interaction between the protein and ligand but also the predicted geometry of interaction. Chapter 5 will explore how we can leverage additional data to improve aspects of the scoring function in a robust manner. Chapter 6 will then generalize this idea to customize the entire scoring function to suit a particular task. The key contributions are in defining procedures to combine positive data (known protein-ligand interaction) with negative data (examples of non-ligands or poor geometries).

To introduce this work, we begin with an overview of the drug development cycle in Section 1.1. Given this context, we then focus on one particular stage, high-throughput screening (Section 1.2) and its *in silico* counterpart, virtual screening. To perform these tasks well, we need to understand the underlying principles of protein-ligand binding (Section 1.3). The biophysics field has contemplated this extensively, encapsulating theory into molecular mechanics force fields (Section 1.4). We, however, approach this problem using machine learning – a brief overview of which is discussed in Section 1.5.

1.1. Drug Development Cycle

The drug industry has launched over 165 biopharmaceutical products[6] to improve the quality of life in our society. In doing so, it has garnered sales revenue of $40 billion

in the United States in 2006 alone.[7] This, however, has not come without significant cost and risk to pharmaceutical companies. Table 1.1 reveals the expected capitalized cost of developing a single successful compound as easily exceeding $350 million dollars in R&D expenditures. If we include the thousands of failed candidates, the estimated total cost can range anywhere from $800 million to $1 billion.[8]

Table 1.1. Capitalized cost per biopharmaceutical compound (in Millions of 2005 dollars)[9]

Testing phase	Expected out-of-pocket cost ($)	Phase length (mos.)	Monthly cost ($)	Start of phase to approval (mos.)	End of phase to approval (mos.)	Expected capitalized cost ($)
Preclinical	59.88	52.0	1.15	149.7	97.7	185.62
Phase I	32.28	19.5	1.66	97.7	78.2	71.78
Phase II	31.55	29.3	1.08	78.2	48.9	56.32
Phase III	45.26	32.9	1.38	48.9	16.0	60.98
Total						374.70

Companies must also project resource allocation into the distant future as the average time to market for any single drug is 12+ years. All of this must be borne in an FDA environment where only 8% of candidate drugs that enter the pipeline find approval.[10] Methods that improve the efficiency of the discovery process can have an impact on human health. The reward is worth it; the pharmaceutical market is projected to be $70 billion by the end of the decade.[6]

Figure 1.1. Standard model of drug discovery[11]

Figure 1.1 presents the "standard model" for drug development. New targets are typically identified with knowledge regarding the biological pathway of a particular

disease. Large chemical libraries are then tested for target activity, usually in a high-throughput screen (HTS). Those small molecules that modulate activity above a certain threshold are selected for optimization of their potency, selectivity, and pharmacokinetic properties. These are then tested for efficacy in animal disease models. A successive compound then undergoes further optimization into a candidate drug suitable for humans before entering three phases of clinical trials. Phase I proceeds in a small group (20-100) of healthy individuals to determine the safety profile of a drug. Phase II establishes the efficacy of the drug in promoting positive outcomes in larger groups (100-500) of disease patients. Phase III then conducts randomized trials with groups of 1000-5000 patients to confirm efficacy with statistically significant data. Monitored concurrently throughout this process are adverse side effects. With successful Phase III clinical trials, a drug is approved by the FDA for sale on the US market. Phase IV testing follows for the life of the drug studying long term effects in the patient population.

1.2. High-Throughput Screening

This work addresses two steps in the drug development cycle. Computational molecular docking can replace or augment high-throughput screening (HTS), and it is also frequently employed in lead optimization. The former is a complex and expensive endeavor and will be briefly described here. Lead optimization is also a key bottleneck, but it is a one-compound-at-a-time design and testing process that can be understood as a low-throughput version of screening. Combinatorial chemistry enables the rapid synthesis of chemical compounds by systematically joining molecular building blocks in different combinations. These large chemical libraries of 10^5 – 10^6 small molecules are tested against a protein target in a high-throughput assay. These assays measure the ability of

the molecule to modulate the activity of the protein. The standard method involves a microtitre plate with as many as 3,456 wells containing different test molecules.[12] Activity with the target is typically measured via fluorescence resonance energy transfer (FRET).[13] A fluorescent molecule attached to each compound is excited by energy at a certain wavelength. The act of binding with the target will change the energy emitted by the fluorophore which can then be measured by a charge-coupled device. Figure 1.2 illustrates a simple example of a labeled substrate being phosphorylated, thus allowing it to interact with a labeled acceptor group. The acceptor group in turn absorbs some of the emitted energy of the donor fluorophore. This is a detectable change directly attributable to substrate binding.

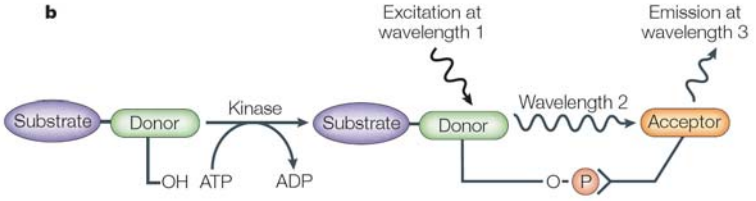


Figure 1.2. FRET-based activity assay[13]

Through continued advances in miniaturization and automation, HTS has been able to generate copious amounts of data. The quality of that data, however, can be fairly low as noted by Lipinski.[11] Given the same chemical library screened against an identical target in three different assays, the concordance in active hits is just 35%. Though much of this result can be attributed to the noise inherent in the methods, the reproducibility within a single assay is much more robust. The coarse filter that is HTS can generate leads for medicinal chemists to further optimize into drug candidates, but there are clearly limitations to the method.

Improvements to this initial filter can save time and money. Virtual screening is a computational method of performing HTS. When the crystallographic structure of a target is known, molecular docking is an inexpensive means of identifying promising molecules to pursue. Docking can provide access to much larger libraries of molecules than can be screened efficiently. This area of research is covered in more detail in Chapter 2.

1.3. Protein-Ligand Binding

Successful screening requires an understanding of the macromolecular interactions that lead to protein-ligand binding. Proteins are linear polymers composed of amino acids joined sequentially to one another via a peptide bond.[14] The distinct sequence of amino acids ultimately gives rise to the three dimensional structure of a protein. This sequence is often referred to as the primary structure of a protein. The secondary structure involves local conformation of the polypeptide chain into recurring structures such as alpha helices, beta sheets, and hairpin loops. The way in which these secondary structure elements combine produces the tertiary structure or global 3D conformation of the protein. Multiple polypeptide chains can further organize into assemblies of larger functional units.

The etymology of the word ligand stems from the Latin verb “ligare” which means to tie or bind.[15] In this work, we refer to a ligand as a molecule of any size that binds or interacts with another through non-covalent forces. The nature of interaction between a ligand and its receptor depends on the delicate balance of physical and chemical forces between them and the forces between each of molecules with the solvent environment. Quantum mechanics describes these forces exactly, but molecular

simulations with full quantum theory remain computationally infeasible for large systems. Another path to understanding can be found via thermodynamics.

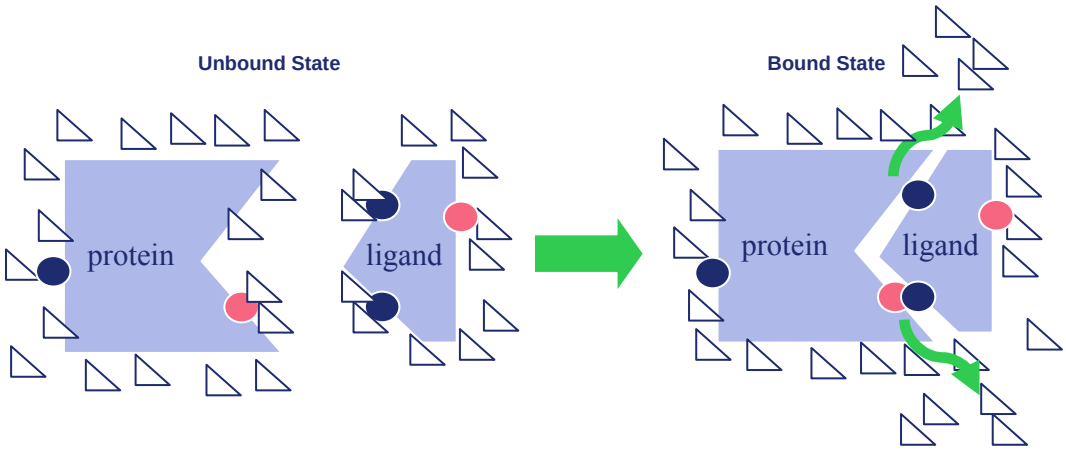


Figure 1.3. Protein-ligand binding process

The binding process begins with the protein and ligand in their unbound state; both roaming free in solvent which is modeled as an aqueous environment (Figure 1.3, left). Polar atoms on the surface of both will have made hydrogen bond interactions with water molecules. Upon binding, van der Waals (vdW) or hydrophobic interactions form at the interface between protein and ligand (Figure 1.3, right). Water molecules within the active site are expelled, their hydrogen bonds replaced by complementary polar interactions with the ligand. The release of ordered water molecules increases the entropy of the system which favors binding (hydrophobic effect). Entropy losses are found in fixing the conformation of the protein and ligand in complex. To determine the strength of binding, thermodynamics attempts to do a detailed accounting of the exchange of hydrogen bonds, addition of van der Waals and polar interactions, entropy loss of the protein and ligand, and entropy gain of the solvent.

Of particular interest is the free energy of binding:[16, 17]

$$\Delta G_{bind} = \Delta G_{complex} - \left(\Delta G_{ligand} + \Delta G_{protein}\right) \quad \text{Eq. 1.1}$$

ΔG_{bind} gives us an indication of the strength of binding. Calculating free energy directly is usually time-intensive, requiring heavy approximation of free energy perturbation or thermodynamic integration methods.[18, 19] The more common experimental technique of calculated binding energy is through the complex's disassociation constant:

$$\Delta G_{bind} = RT \ln K_d \quad \text{Eq. 1.2}$$

where R is the gas constant and T is the absolute temperature. The disassociation constant K_d is an equilibrium constant that measures the propensity of a protein-ligand complex to dissociate back into its component parts, an unbound protein and unbound ligand.

Since direct measurement of K_d is difficult, experimental binding data is often reported using K_i or IC50 as surrogates.[20] These are generated from competitive binding assays which measure the change in uptake of a labeled radioligand in the presence of a competing molecule. IC50 is the concentration of competing ligand that displaces 50% of the specific binding of the radioligand; K_i is the concentration of competing ligand that would occupy 50% of receptors with no radioligand present. K_i may be calculated from IC50 if the K_d of radioligand-receptor binding is known using the Cheng-Prusoff equation:[21]

$$K_i = \frac{IC50}{1 + \frac{[L]}{K_{d.radio}}} \quad \text{Eq. 1.3}$$

Importantly, enzymological measurements of K_i may be related to K_d but are highly dependent on experimental conditions such as temperature, buffer pH, specific substrate

used, and salt conditions. As such, whether a K_i may act as a proxy for K_d must be determined on a case-by-case basis.[20]

Another method by which one arrives at binding energy is by examining the changes in enthalpy (ΔH) and entropy (ΔS) upon complexation:

$$\Delta G = \Delta H - T\Delta S \qquad \text{Eq. 1.4}$$

Here enthalpic changes arise from the van der Waals and electrostatic interactions made between protein and ligand atoms, replacing those lost with solvent. Entropic change encompasses the degrees of freedom (translational, rotational, vibrational) lost to the protein and ligand due to binding. One method of predicting these energies using first principles is described in the following section. Our empirical approach is introduced in Section 2.2.1 of the literature review and described in full in Chapter 6.

1.4. Molecular Mechanics Force Fields

Study of structure-activity relationships using first principles is commonly done using a molecular mechanics force field. This method calculates separately the intramolecular forces (between bonded atoms) and intermolecular forces (between non-bonded atoms) resulting in a prediction of the enthalpic contribution to binding energy. The entropic contribution is omitted. The general form of this potential energy function is given below:[17]

$$E_{MM} = K_b \sum_{bonds}(r - r_0)^2 + K_a \sum_{angles}(\theta - \theta_0)^2 + K_t \sum_{dihedrals} 1 + \cos(n\phi - \delta) + K_i \sum_{impropers}(\psi - \psi_0)^2 + \sum_{nonbonded(i<j)} \left[\left(\frac{A_{ij}}{r^{12}} - \frac{B_{ij}}{r^6} \right) + \frac{q_i q_j}{\varepsilon \cdot r_{ij}^2} \right] \qquad \text{Eq. 1.5}$$

The first four terms of the function operate on bonded atoms. Respectively, they model the deviation from ideal bond length, ideal bond angle, and ideal torsional angle; the fourth term penalizes improper out-of-plane distortions. The last term of the function operates on non-bonded atom pairs measuring the hydrophobic energy via a 6-12 Lennard-Jones potential and the electrostatic energy via Coulomb's Law.

Parameterization of force fields is a daunting task as there are a multitude of parameters to fit. Consider the Lennard-Jones potential which offers two parameters for controlling the depth and width of the energy well that describes the maximum benefit of bringing two atoms together. For a system considering N different atom types, a total of N(N-1) / 2 parameters are required.[17]. To lower the number of atoms modeled, force fields often take a united-atom approach where nonpolar hydrogens are ignored. Note also that most do not include polarizability in their electrostatic models. Instead, this is treated implicitly by choosing partial atomic charges that overweight molecular dipoles. Another complication is the standard procedure of optimizing parameters such that the force field can reproduce QM results in the gas phase – yet, the force field is used in condensed phase applications such as docking.[22] In general, force fields are trained to reproduce the properties of a specific molecular system such as proteins or nucleic acids. Though they are designed to be applied across broad classes of molecular systems, the scope of the original parameterization is important.

Several force fields have gained notoriety in the domain of molecular modeling. They all diverge slightly in their functional form and broadly in their parameters due to differences in training. AMBER[23] and its derivative OPLS[24] have been used to model

proteins, nucleic acids, and carbohydrates. The CHARMM[25] force field models 12 less atom types and has been extensively optimized for use with proteins. The DREIDING[26] force field has been tuned to model small molecules and is used extensively in this work to minimize ligand conformations prior to docking.

1.5. Machine Learning

The computer science field of artificial intelligence (AI) can be defined as the study of algorithms that "make it possible to perceive, reason, and act".[27] The typical AI problem involves capturing complex relationships between relevant descriptors and observed outcomes. The hope is that what is learned from a finite training set can generally be applied to the remaining space. If each example in the training set has a known descriptor or associated target value, this becomes a supervised learning problem.[28] In essence, the goal is to create a rational agent specialized to a domain that is capable of classifying or predicting future outcomes.

This is a simple task when the association between descriptors and outcomes is fairly linear. In drug development, however, the parameters influencing biological activity tend to be numerous and interact in a non-linear fashion (consider the Lennard-Jones potential in Section 1.4). Additional difficulty arises when there is not enough experimental data, also known as the "curse of dimensionality".[29] If m points are necessary to reasonably define a single dimension, then m^n points are required for n dimensions. Thus, if it takes 10 examples to approximate the relationship of a single parameter, then 17 parameters would require 10^{17} examples. Clearly, encoding knowledge in a higher dimensional space requires astute model selection to succeed.

Several such models are embedded in the scoring functions of molecular dockers. These are covered in depth in Section 2.2.1 of the Chapter 2 literature review.

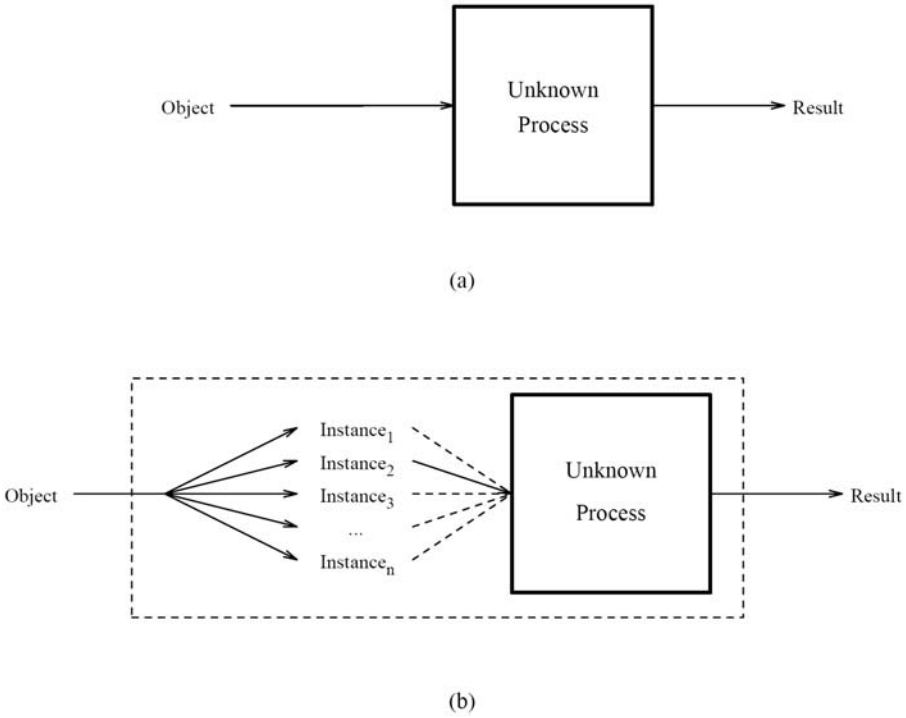


Figure 1.4. Supervised learning: (a) usual situation and (b) multiple instance situation[4]

Consider now the scenario in Figure 1.4 where our learner has only partial or incomplete knowledge about each training example. Instead of each example being represented by a single feature vector, each might be represented by a *set* of potential feature vectors of which only one may be responsible for the observed result. The ambiguous nature of training input arises in the domain of activity prediction for drugs.[4, 5] In this example, the object is a ligand and the observed result is the binding affinity of that ligand with its target. The multiple instances here are the various conformations (rotatable bonds, alignment relative to protein) the ligand can adopt within the binding pocket. At ordinary temperatures, the molecule conformation is constantly changing.

Only a few of these can provide the ideal interactions necessary with the protein to produce the required binding affinity. Each conformation has a certain potential energy that is related to its atomic interactions. The probability that the molecule exists in any particular conformation is exponentially dependent on the potential energy of the conformation according to the Boltzmann distribution. The most probable conformations are lower in energy, and thus more likely to be the correct binding pose.[16] Identifying and retaining low energy poses out of the space of infinite poses as we simultaneously optimize a protein-ligand scoring function is a multiple instance problem. An efficient method of handling this learning complexity is described in Chapter 5 and Chapter 6 as we refine and create new scoring functions.

1.6. Conclusion

This chapter provided context for our work in molecular docking. The need for increased efficiency in the drug development cycle is motivated in Section 1.1. One step in that cycle where computational methods can be applied to great effect is in high-throughput screening (Section 1.2). Here, the game is to detect novel leads for a target of interest from a sea of noise – large chemical libraries full of molecules which do not bind. To understand what constitutes a good binder, we present fundamental concepts in protein-ligand binding in Section 1.3. Molecular mechanics force fields offer a method for estimating binding energy from first principles (Section 1.4). Finally, Section 1.5 outlines the role of artificial intelligence and obstacles to machine learning in virtual screening.

The next chapter will review existing efforts in the field of molecular docking.

Chapter 2
Review of Molecular Docking Literature

2.1. Introduction

The last chapter introduced the topics of drug development with an emphasis on HTS, as the biophysics embedded in that task is amenable to machine learning approaches. In this chapter we will discuss broadly the state of the art in one such approach: molecular docking. Such algorithms all have a search (method of sampling ligand and protein conformation space) and score component (determination of binding strength). Section 2.2 will cover the different methodologies used in search, and Section 2.3 will discuss several types of scoring functions. The last section first describes how algorithms are validated and then provides head-head comparisons of the leading approaches.

2.2. Search

In the search for an optimum pose, a molecular docker must adequately search the conformation and alignment space available to the ligand. The method used to explore

ligand flexibility and alignment within the binding pocket can be broken down into three general approach categories: systematic, stochastic, and simulation. Protein flexibility is discussed in Section 2.2.4.

2.2.1. Systematic methods

This method attempts to explore all the possible conformations of a ligand, a space that is exponential in the number of rotatable bonds.[30] If one has a molecule with 6 rotatable bonds and wants to sample 10 rotations per bond, there are 10^6 conformations to compute. To avoid an expensive exhaustive search, systematic algorithms will incrementally grow the ligand pose into the active site. By fragmenting ligands at rotatable bonds, rigid fragments can be docked piecemeal. The best fragments become seeds for growing the remaining portion of the molecule. Each subsequent step attaches a new fragment after torsional sampling. Obvious crashes with the protein are eliminated. By retaining only the highest ranking solutions during each iteration, the exponential search space can be pruned effectively in a greedy manner.

This is the strategy employed by DOCK[31, 32], and FlexX[33] with variation in the way initial alignments are sought. DOCK places the initial fragment based on shape complementarity whereas FlexX also considers favorable geometries for potential polar interactions. Surflex[30] takes a different, whole molecule approach. Molecules are fragmented with 2-6 rotatable bonds per fragment; all fragments are docked into the binding site. The algorithm also retains along with the docked fragment the initial arbitrary conformation of the remaining molecule (see Figure 2.1). High ranking fragments are then recursively searched for acceptable mutual geometry such that the two can be merged. As each fragment is added to the growing molecule, an energy

minimization step is performed to eliminate the strain energy from poor bond lengths, angles, and torsions.

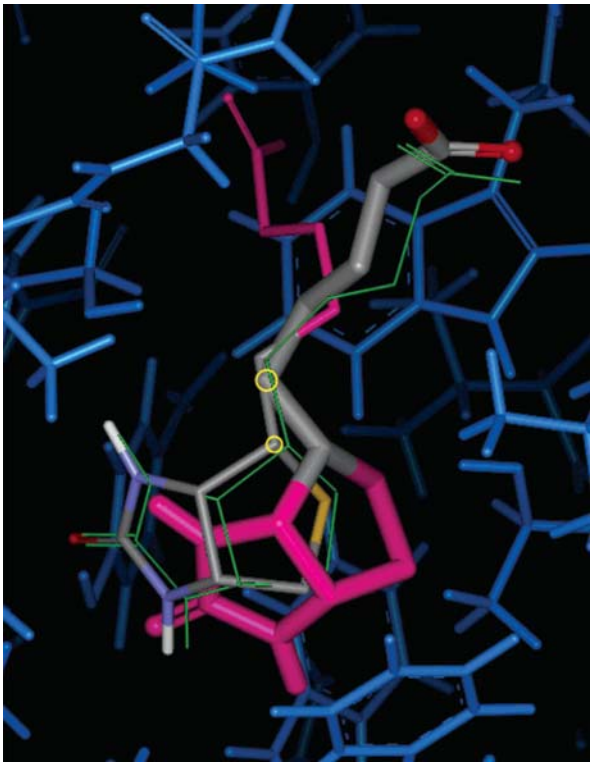


Figure 2.1. Surflex whole molecule search process[30]

In green is the experimental pose for biotin bound to streptavidin pictured in blue (PDB: 1STP). A high-scoring fragment that properly emulates the pose of biotin's ring system is shown in thin, atom color sticks. Note how its magenta tail is crashing into the protein. Conversely, in thick, atom color sticks is a high-scoring tail fragment whose magenta ring system is incorrect. By merging the thin ring fragment to the thick tail fragment at the bond highlighted with the two yellow circles, we can recreate the ideal biotin pose. This merged pose closely follows the parent fragments' original configuration.

Another systematic search technique is to precompute a conformational library of ligand poses. Glide[34] selects a set of initial ligand conformations through exhaustive enumeration of minima in ligand torsion-angle space. Filters are then applied over the

entire phase space to locate promising poses. The remaining set is then subject to minimization with an OPLS-AA force field.[24]

2.2.2. Stochastic methods

These methods randomly perturb a single ligand or population of ligands, followed by evaluation with a pre-defined probability function. Early versions of AutoDock[35] uses a simulated annealing Monte Carlo (MC) procedure. At each step, the ligand's six degrees of spatial freedom (translational and rotational) and an arbitrary number of torsional degrees (rotatable bonds) are randomly perturbed and the interaction energy is calculated. The new ligand state is probabilistically accepted or rejected based on the system annealing temperature and a Metropolis criterion: if the new pose scores better, we immediately accept it; otherwise accept the pose only if it passes a Boltzmann-based probability function test.[36] In the early stages, the temperature is high thus allowing the ligand to explore large areas of conformational space. The temperature is gradually lowered to hinder unfavorable moves and investigate energy minima. By repeating this process from several random initial states, consistent low energy binding modes can be found. Glide[37] also makes use of MC to examine nearby torsional minima for its top poses.

GOLD[38] and AutoDock 3.0[39] utilize a genetic algorithm that encodes the principles of biological competition and population dynamics. Trial poses are encoded into "chromosomes" in several subpopulations that are stochastically varied. Only chromosomes which pass a fitness test survive to the next generation. The best intermediate solutions are subject to genetic operations of crossover (exploring search space), mutation (exploring local minima), and migration between subpopulations. The

child produced by these operations replaces the least fit member of its population. The end product of this genetic algorithm is an ensemble of possible docking solutions.

2.2.3. Simulation methods

Molecular dynamics (MD) allows one to see how atoms and molecules interact by directly integrating Newton's equations of motion in a given potential.[40] MD, however, suffers from the inability to cross energy barriers larger than 1-2kT within reasonable simulation timeframes.[16, 41] Encountering a jagged potential energy landscape can easily trap the simulation in local minima. To combat this, different parts of the system may be simulated at different temperatures.[42] Other methods smooth the potential energy surface to allow for greater exploration of ligand space.[43] Currently, however, the large length of time necessary to run a single MD simulation renders it infeasible for virtual screening.

2.2.4. Receptor flexibility

For the sake of complexity, most methods treat the protein as a rigid body. Ideally, the degrees of freedom available to both the ligand and receptor should be explored since many complexes show induced fit.[14] The simplest way of incorporating receptor flexibility is by softening the scoring function penalty for atomic overlap. This makes atomic van der Waals boundaries fuzzy, implicitly capturing structural uncertainties and small side chain movements within the pocket. This is the subject of much discussion in Chapter 5 where we refine the Surflex scoring function parameter governing steric overlap. Leveraging the systematic search technique from Section 2.2.1, side chain rotamer libraries may also be used to sample protein flexibility of select active site residues.[44] Alternatively, a composite receptor structure can be generated by superimposing several different crystallographic models of the same receptor. DOCK[45]

can map the average interaction potential of the several different structures onto a single potential energy grid (see Section 2.3.1). Stochastic methods such as MC and MD simulations are readily extended to treat certain active site side chains flexibly.[43] In conclusion, Kumar et al. have suggested that even a tightly bound complex can be thought of as an ensemble of microstates.[46] Improved capture of these ligand-protein ensemble states would lead docking results to better resemble reality.

2.3. Score

A good search strategy is able to sample adequately a ligand's infinite space of poses such that the experimental binding pose is amongst those generated. A good scoring function is able to *recognize* good poses from bad. Together, they combine to form a molecular docker. Several different scoring avenues have been explored for this purpose: molecular mechanics force-fields, empirical methods, knowledge-based methods, and consensus methods.

2.3.1. Force field based scoring

The first molecular dockers adapted molecular mechanics force fields as scoring functions. For a review of their construction, refer to Section 1.4. Most functions score only the interaction energy between the protein and ligand, neglecting entropy and protein internal energy completely. Ligand strain energy is usually accounted for via in-line minimization during docking. By making use of only the non-bonded force field terms (hydrophobic and electrostatic terms), applications can limit the computational complexity of scoring and be run in a high-throughput manner.

Seminal work done by Kuntz et al.[47] used the non-bonded terms of AMBER[23] to precompute the protein interaction energy at grid points inside the active site. Caching these values enabled rapid calculation of binding energies when DOCK placed a ligand atom at or near the grid point. Screening of polar interactions by solvent was modeled using a simple distance dependent dielectric. Shoichet et al. later extended DOCK to include an implicit solvent model that better predicts experimental binding energies.[48] GOLD[38] and AutoDock[35] differ in that they include an explicit hydrogen-bonding term with directional dependence as well as a torsional term that estimates ligand strain. GOLD also relaxes the interpenetration penalty to a 4-8 Lennard-Jones potential instead of the usual 6-12 (see Section 1.4). By including a tunable hydrogen bonding term, these latter two functions have introduced an empirical-type term into their force field.

2.3.2. Empirical scoring functions

These scoring functions are fit to experimental data to reproduce binding affinities and conformations. They are usually composed of several additive terms thought to be important to binding free energy; many of which also have a counterpart in a molecular mechanics force field. Each term is usually weighted by a scaling factor learned from molecular data. Due to parameterization, the decomposition of forces to individual terms can be difficult to interpret when experimental results are not reproduced, though this is a common complaint of all scoring functions regardless of type. All empirical functions are trained on a finite (and hopefully representative) dataset with the assumption that the scoring function can be generally applied.

The first such function LUDI was introduced by Bohm:[49]

Eq. 2.1

$$\Delta G_{bind} = \Delta G_{hb} \sum_{h-bonds} f(\Delta r, \Delta a) + \Delta G_{ionic} \sum_{ionic} f(\Delta r, \Delta a) + \Delta G_{lipo} \mathrm{A}_{lipo} + \Delta G_{rot} \mathrm{N}_{rot} + \Delta G_0$$

where each ΔG_x are the tunable scale factors; the function *f* encodes preferred interatomic distances and directions for polar interactions; ΔG_{hb} scales the contribution from ideal hydrogen bonds; ΔG_{ionic} scales the contribution from unperturbed ionic bonds; ΔG_{lipo} scales the contribution from the hydrophobic effect; A_{lipo} is the hydrophobic surface area; ΔG_{rot} scales the contribution due to fixing rotatable bonds; N_{rot} is the number of rotatable bonds; and ΔG_0 represents everything not captured by the terms listed (such as translational entropy, solute entropy, etc). This function was calibrated on 45 protein-ligand complexes and was able to predict their binding affinity with an mean error of 1.9 kcal/mol.

FlexX[33] modified Bohm's function[49] by making explicit the distinction between aromatic contacts vs. lipophilic contacts. Aromatic contacts have a form similar to Bohm's h-bond term (Eq. 2.1) with specific distance and angle dependencies. An ideal contact between two lipophilic or nonpolar contacts is a pairwise operation modeled as the sum of their vdW radii. This function was able to reproduce the binding modes of 19 protein-ligand complexes within 0.5 and 1.2Å rmsd. Refer to Section 2.4.2 for an explanation of docking accuracy and rmsd.

ChemScore[50] is also a Bohm-like[49] function:

Eq. 2.2

$$\Delta G_{bind} = \Delta G_{hb} \sum_{h-bonds} f(\Delta r, \Delta a) + \Delta G_{metal} \sum_{metal} f(r) + \Delta G_{lipo} \sum_{lipo} f(r) + \Delta G_{rot} H_{rot} + \Delta G_0$$

This scoring function treats polar interactions in a unified way. Its contact-based hydrophobic term is similar to that of FlexX[33]. It also adds a term for contributions from metal chelation. ChemScore introduced a novel way in which to model the entropic cost of fixing rotatable bonds by folding into the calculation the local bond environment. Calibrated with 82 complexes, this function predicted binding affinities with a cross-validation error of 2.07 kcal/mol.

The scoring function used in Glide[34] is a modified ChemScore descendant. Electrostatic interactions between neutral-neutral, neutral-charged, and charged-charged atoms are each distinguished with their own polar term. Additionally, GlideScore attempts to model solvation phenomena. First, it includes a term that counts polar moieties that have been buried against a hydrophobic surface. Second, by docking explicit waters along with the ligand into the active site, it can measure the degree to which polar atoms have been desolvated.

Details of the Surflex[51] scoring function is in Section 6.3.1 as it is highly relevant to the customization work described in Chapter 6. Briefly, this 17 parameter function is a linear combination of Gaussian and sigmoidal functions. There are five major terms that model hydrophobic interactions, polar interactions, mismatched polar contacts, entropy, and desolvation. It was trained on 34 protein-ligand structures with an affinity prediction error of 1.4 kcal/mol.

2.3.3. Knowledge-based scoring

Knowledge-based functions are similar to their empirical brethren in that they both are generated from molecular data. Knowledge-based potentials are based on the observed frequencies of pairwise atom distances in protein-ligand structures. When these frequencies are converted to free energies using the Boltzmann distribution, they create a potential of mean force (PMF) that in theory captures implicitly all aspects inherent to binding.[16, 36] They also do not rely on binding affinity data, greatly simplifying the task of creating large training datasets. The general form of a PMF is given by the Helmholtz free energy of interaction:[52]

$$\Delta W = \sum_{i}^{ligand} \sum_{j}^{protein} \Delta\omega_{ij}\left(r_{ij}\right) \qquad \text{Eq. 2.3}$$

where the protein-ligand conformation score ΔW is the sum of all pairwise potentials $\Delta\omega$ between ligand atom i and protein atom j. The statistical potential has the form:

$$\Delta\omega_{ij}(r) = -kT \ln\left(\frac{g_{ij}(r)}{g(r)}\right) \qquad \text{Eq. 2.4}$$

where $g_{ij}(r)$ is the probability that group i and group j are proximal to one another at a distance r; $g(r)$ is the normalization reference probability selected such that $g_{ij}(r)$ approaches zero as r approaches infinity (where the protein and ligand are unbound); k is the Boltzmann constant; and T is the absolute temperature. Care must be taken in the choice of reference state. Approximations often use either a specific empirical value or a volume-corrected average potential over all training data.[41] Pairwise distances can be measured in either a continuous or coarse manner with binned thresholds.

PMF[53] is a smooth potential that tracks solvent effects within 12Å, non-carbon pairs within 9Å, and carbon pairs within 6Å. Only protein-ligand atom pairs of the same

type are considered as the reference state. After training on 687 structures, the best affinity prediction error of 1.4 kcal/mol was reported for 16 serine proteases. The work of Gohlke saw the development of another smooth potential, DrugScore[54], which is derived from 1,376 complexes. Only short distances less than 6Å are considered in order to emphasize specific protein-ligand interactions. Solvent effects are calculated using a separate potential based on atomic surface area. On a test set of 91 structures, DrugScore was able to improve the ability of FlexX[33] to recognize correctly docked poses by 35%.

2.3.4. Consensus scoring

By using a composite of several different functions, consensus schemes hope to minimize the imperfections of individual scoring methods and maximize the probability of finding active ligands. However, consensus scoring also magnifies any error shared between functions. Wang et al.[55] tested combinations of all three different types of functions discussed: force fields (FlexX,[33] X-Score[56]); empirical (LUDI,[49] LigScore[55]); and PMFs (PLP[57], DrugScore[54]). By averaging the scoring ranks of conformational ensembles from the different functions, the consensus method was able to improve identification of the correct binding mode by 80%. Charifson et al.[58] also found that a consensus approach of 13 different scoring functions was useful in reducing the number of false positives returned in virtual screening since different functions tended to produce different sets of false positives. Though, it should be noted that virtual screening may potentially lose hits using this method due to the smaller intersection enforced by multiple algorithms.

2.4. Algorithm Assessment

Molecular dockers are typically assessed by three metrics: scoring accuracy, docking accuracy, and screening utility. Most published benchmarking studies, however, focus on the latter two in order to gauge success between several competing algorithms.

2.4.1. Scoring Accuracy

Throughout the literature review of scoring functions in Section 2.3, we have reported where available a scoring function's affinity prediction error in kcal/mol. This attribute is interesting only in regard to the specific testing set for which it was reported. Moreover, there are important caveats regarding experimental affinity data, as discussed in Section 1.3. To sum, the large variance in reported K_d due to differences in experimental setup can render the problem of fitting binding affinity data incredibly difficult. Many methods rely on the crystallographic structure of a receptor for docking, yet rarely is that structure determined in the identical conditions used to measure binding.

Addressing these problems remains challenging. The solution pH can be taken into account in part by assigning different partial charges to ionizable groups on amino acids and ligands. The temperature dependence of enthalpy and entropy are impossible to model without time-intensive simulation methods. Generally, the precise environmental conditions under which the binding affinity was obtained are ignored in the calculations of binding free energy.[17] Successful virtual screening requires only the correct rank ordering of compounds to work. As such, a scoring function needs only predict accurately the *relative* affinity of one ligand to another. This allows for cancellation of errors.[52] Consider two ligands that are similar or bind similarly; their entropy terms are probably also very similar. Regardless of whether a scoring function has a poor or

excellent model of entropy, this term is ultimately subtracted out when we consider only the differences in affinity between two ligands.

Nevertheless, Perola et al.[59] considered the scoring accuracy of 9 functions: ChemScore,[50] GlideScore,[34] PLP,[57] PMF,[53] PMF612,[59] MMFF_VDW,[60] MMFF_TOT,[60] OPLS_VDW,[24] OPLS_TOT.[24] Using a set of 111 diverse, pharmaceutically relevant complexes, poor correlation between predicted and experimental binding affinities was observed. A genetic algorithm-based subset selection of the data further showed that given the appropriate dataset, any function can appear to be predictive. Thus, for methods that were trained and tested using the same dataset, the reported scoring accuracy must be interpreted as valid only for those and similar complexes. Thus, we arrive at this need for model testing on datasets that were not a part of the model building process. This important theme is stressed throughout our machine learning work in Chapter 5 and Chapter 6.

2.4.2. Docking Accuracy

How well does a docker recapitulate the geometry of a protein-ligand complex? This is usually measured by the root-mean-square deviation (rmsd) of non-hydrogen atom positions in the predicted ligand pose versus that in the crystallographic structure:

$$RMSD\left(pose_i, pose_j\right) = \sqrt{\frac{\sum_n \left(atom_i - atom_j\right)^2}{n}} \qquad \text{Eq. 2.5}$$

Prediction within 2.0Å rmsd of the correct pose is widely held in the field as the passing standard.[41] The best rmsd over all docked poses and the rmsd of the top ranked pose by score are generally reported. The latter is of particular importance since recognition of the best pose with the most favorable energy can be highly relevant in lead optimization.[20]

Examination of poses and their scores also allows classification of docking error into two types.[61] A "soft failure" occurs when the best ligand pose by rmsd has a higher score (less favorable) than the experimental binding affinity. This indicates a failure in the search strategy to locate the optimal, lower scoring native pose. Conversely, a "hard failure" occurs when the score of any pose is lower in energy than that of the experimental pose. Assuming the crystallographic pose exists in the global energy minimum, the scoring function here has overestimated the binding affinity of the alternate pose. Remedying soft failures requires increased sampling of conformational space; hard failures are fixed only through improvement of the scoring function. To see a systematic method for addressing hard failures, consult Chapter 6.

The most reliable and impartial benchmarking study was performed by Kellenberger et al.[62], a research group with no investment in any of the methods reviewed. Eight docking programs were included in the study: DOCK,[31] FlexX,[33] Fred,[63] Glide,[34] GOLD,[64] Slide,[65] Surflex,[30] and QXP.[66] Docking accuracy was assessed on 100 protein-ligand complexes.

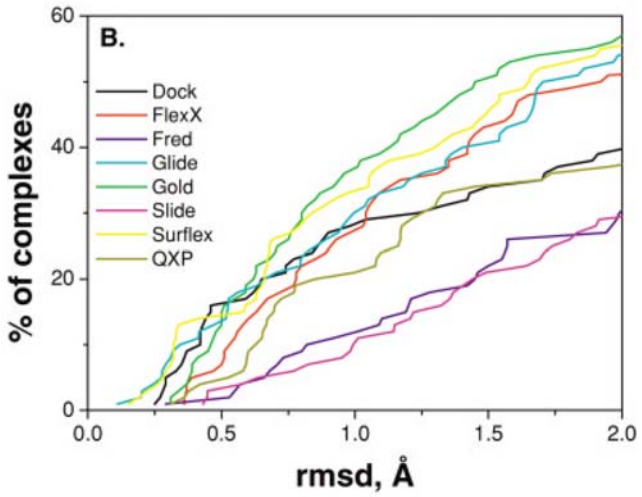


Figure 2.2. Docking accuracy of 8 different methods[62]

Pictured is a plot of the percentage of successful dockings < 2.0 Å rmsd of each method for 100 protein-ligand complexes. Only the rmsd of the top ranked pose returned by score is reported.

QXP exhibited an extreme sensitivity to the starting conformation that rendered it unable to produce good dockings unless the crystallographic pose was used as input. Of the remaining methods, GOLD, Surflex, Glide, and FlexX (in order of decreasing performance) behaved similarly with a 50 to 55% success rate in returning a top ranked pose within 2.0Å rmsd of the native pose, whereas DOCK, Fred, and Slide (in order of decreasing performance) do not exceed 40%. Despite very different implementations, it is notable that the best methods have relatively equivalent performance in generating and recognizing the correct binding mode of a ligand.

2.4.3. Screening Utility

Screening utility measures the relative improvement of using a molecular docker over random selection in identifying active (true positive) ligands for a protein amongst a sea of decoy (false positive) ligands thought or known not to bind. A screening library is

thus composed of both true positives (TPs) and false positives (FPs). Several performance metrics are in use. This work uses full receiver operating characteristic (ROC) curves to plot the rate of recovery of TPs relative to FPs. Others report enrichment factors:[52]

$$EF = \frac{a/n}{A/N} \qquad \text{Eq. 2.6}$$

where a is the number of active compounds in the top n ranked compounds returned by a docker from a screening library of N compounds of which A are active. Typically, enrichment factors are given after a specific proportion (1, 3, 5, or 10%) of the library has been screened. Still others only report the maximal enrichment factor over all proportions. All of these metrics reflect similar characteristics, and many are computable from each other.

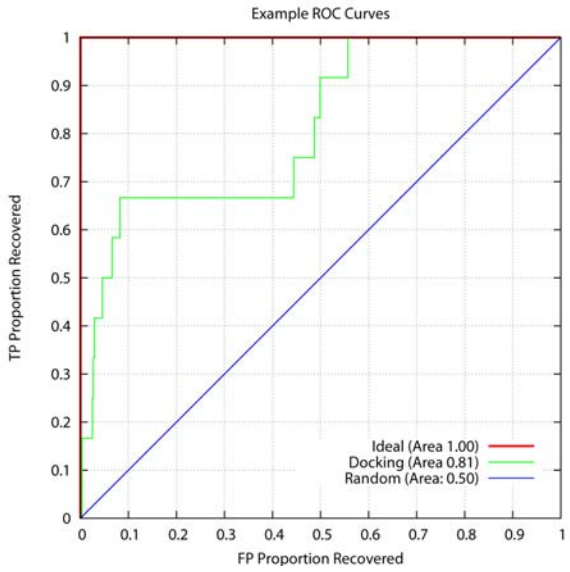


Figure 2.3. ROC curves: ideal vs docking vs random

For example, enrichment factors at any screening level can be read directly from an ROC curve. Additionally, screening improvement in ROC plots are easy to discern by the

change in area under the curve (AUC). Consider Figure 2.3: an ideal function (red) that could perfectly identify actives from decoys would have an area of 1.0 where 100% of the TPs are recovered before a single FP is screened. Conversely, random screening (blue) would have an AUC of 0.5. Docking methods (green) fall somewhere in between these two extremes. Improvements in ROC will see a leftward shift (and increased AUC) such that the curve begins to approach the ideal.

Rognan et al.[67] introduced a screening benchmark consisting of two proteins, thymidine kinase (TK) and estrogen receptor (ER) that was used to evaluate the screening performance of GOLD,[38] DOCK,[47] and FlexX.[33]. Each protein had 10 active ligands which were screened against a decoy background of 990 random molecules from the Available Chemicals Directory (ACD). This dataset was used unmodified by Jain to validate the screening utility of Surflex.[30] Table 2.1 shows the combined reports for all four programs.

Table 2.1. Comparison of screening utility for thymidine kinase and estrogen receptor[20]

TP rate (%)	FP rate from 990 random ligands (%)							
	Thymidine kinase				Estrogen receptor			
	DOCK	FlexX	GOLD	Surflex	DOCK	FlexX	GOLD	Surflex
80	23.4	8.8	8.3	0.9	13.3	57.8	5.3	0.2
90	25.5	13.3	9.1	2.8	17.4	70.9	8.3	0.7

This result indicates that for these two proteins Surflex has a much lower FP rate upon recovery of 80 to 90% of the active ligands than competing algorithms. Several other docking programs were independently analyzed with the TK screening set in a subsequent work.[62] Those results are summarized in Table 2.2 and the enrichment plot in Figure 2.4. The table provides the recovery rate for actives at fixed decoy rates. Here one

can think of the FP rate as the percentage of library screened; this is true when the number of decoys (990) is much greater than the number of actives (10).

Table 2.2. TP rates for fixed FP rates[20]

FP (%)	Thymidine kinase TP rate (%)							
	DOCK	FlexX	Fred	Glide	GOLD	QXP	Slide	Surflex
2.5	0.0	20.0	0.0	20.0	10.0	0.0	0.0	40.0
5.0	10.0	40.0	0.0	50.0	40.0	20.0	0.0	80.0

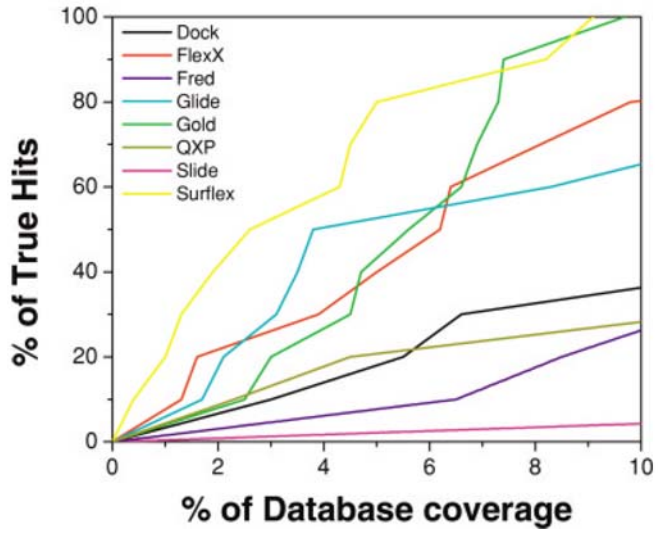


Figure 2.4. Enrichment in TK inhibitors for 8 methods[62]

Thus, it appears that Surflex is significantly better at predicting the correct rank order of this particular screening library. Clearly, additional data is necessary to draw more substantial conclusions. Chapter 3 provides an opportunity for accomplishing this task in an open and automated fashion. At the very least, this result offers a starting point for discussing the adoption of uniform benchmarking in the field. The work in Chapter 5 introduced a public dataset of 29 screening targets with the hope of facilitating this standard of cross-method validation. Finally, Chapter 6 illustrates a novel method for optimizing functions in screening against specific targets.

2.5. Conclusion

There are myriad ways to search ligand conformational space and score the resulting pose. Many of the leading molecular dockers were reviewed in this chapter with an emphasis on the design choices made with respect to search and score. In an ideal world, all correct approaches would converge. This is especially true of scoring since all functions are trying to capture the nuances of binding given a structural snapshot of a protein-ligand complex. Remarkably, such convergence was observed for Surflex,[30] a PMF method,[68] and ChemScore[50] in the directional dependence of hydrogen bonding.[69]

Which approach is right? This is difficult to determine given the limited public benchmarking of all programs. Our empirical approach has a small number of parameters making it easy to train. It uses a systematic search strategy that rapidly culls the flexible search space. Its ability to reproduce the correct ligand binding mode is on par with the best algorithms in the field. And in the application arena of virtual screening, it has been shown independently to be the most efficient in recovering actives with low false positive rates. The rest of this dissertation will detail how we go about systematically improving this already successful method.

This work requires a large amount of data. In Chapter 3, we introduce pdbgrind, an open source utility that can generate molecular data suitable for docking in a rapid and automated fashion.

Chapter 3
pdbgrind

3.1. Abstract

Pdbgrind is a freely available, open source program designed to automate the process of data generation from the PDB for use in large-scale molecular docking projects. Given the minimal atomic coordinates and element information available in a standard PDB file, pdbgrind will accurately predict with a rule-based engine the bond order and protonation state of the extracted protein and associated ligands. This approach was validated on a test set of 800 structures from the PDBbind database, achieving ligand protonation accuracy rates of over 91%. Challenges to conversion include poor resolution on solved structures, inconsistent naming within PDB files, and ambiguous or unmodeled chemical moiety geometries.

3.2. Introduction

This chapter describes a tool that was developed out of necessity due to the nature of the molecular docking field. Currently, there is no single uniform method for preparing molecular datasets for docking, though this is changing albeit slowly. This is often due to the specialized preparation necessary for individual docking programs. The exact details

of how data is massaged a priori are often overlooked and left unpublished, which can lead to skewed validation results. For a more detailed discussion, please refer to other work.[70] Moreover, learning algorithms require large amounts of data to provide clean separation between training and testing sets. We sought to create a standard, automated, open source platform for creating directly from the PDB large-scale datasets that are suitable for docking method development and validation. The culmination of this work is the tool pdbgrind.

The explosion in the number of crystallographically determined structures of protein-ligand complexes has provided a wealth of data for use in the development and validation of docking algorithms. The PDB acts as the primary repository for this data where structures are deposited as PDB files. These contain primarily atomic spatial coordinates coupled with annotation regarding the crystallization experiment. Docking algorithms, however, require a richer representation of the complex in question. Specifically, a need arises for accurate bond connectivity, bond order, and most importantly, correct protonation states of all interacting atoms within the protein binding pocket and the bound ligand. It is also imperative to infer this knowledge in an automated way with minimal manual intervention in order to take advantage of the large amount of data available for docking studies.

Several methods have been developed attempting to address the general problem of ligand data extraction from the PDB, yet the number of standalone programs is few. PDB2PQR[71] attempts the protonation of biomolecules for use in continuum electrostatics calculations, but neglects complexed ligands as it does not process the HETATM entries in a PDB file. Similarly, Word, et al wrote the program REDUCE[72] to protonate PDB

files with a particular focus on optimizing protein side chain orientations, but did not extend this process to a bound ligand. OpenBabel[73] is an open source chemical file format converter but does not do ligand extraction or optimization of protein-ligand contacts. The PRODRG[74] web server can infer molecular topologies from coordinate data. However, it is limited in scope to only a small number of modeled heavy atoms. Other solutions[75, 76] are not available as standalone programs, but are components of larger molecular modeling packages, which are either not amenable to high-throughput data generation or require the purchase of usage licenses.

We developed pdbgrind to provide an automatic method of rapid refinement of quality protein-ligand complexes from PDB files. Similar to BALI,[75] pdbgrind deduces bond orders and protonation states for bound ligand atoms by recognizing recurring functional groups and aromatic ring systems. Moreover, pdbgrind can extract small molecules from PDB complexes using inferred connectivity information. It further refines the protein-ligand structure for docking studies by optimizing proton interactions within the binding pocket. The pdbgrind algorithm was validated on the PDBbind database,[77] a curated set of protein-ligand complexes of known binding affinity.

3.3. Methods

3.3.1. Implementation Details

Pdbgrind saves all inferred connectivity, bond order, atom type, and protonation information into the more descriptive Tripos .MOL2 file format.[78] MOL2 is the widely distributed standard in small molecule data exchange that can be viewed in several packages such as RASMOL[79] and Chimera.[80] This concise file specification is commonly

used by molecular modeling and docking algorithms. This program was developed specifically to generate docking data for use in optimizing and validating the flexible molecular docker Surflex.[30] All pdbgrind code was written in ANSI C and implemented on a Dell Workstation PWS530 (Dual Pentium 1.50GHz, 512Mb RAM). Extensive usage and code documentation is given in Appendix A. Pdbgrind is freely available at www.jainlab.org.

3.3.2. Atomic Connectivity

Similar to the method of Baber and Hodgkin,[81] an interatomic distance metric is used to determine connectivity between pairs of atoms. Two atoms, whose interatomic distance is less than the sum of their van der Waals radii (Table 3.1) minus a tolerance of 1.25Å, are considered bonded. Whereas others have used covalent radii to infer bonds between atom pairs, the assumption that unexpected proximity given the repulsive force from van der Waals shells implies covalent bonding has proven accurate.

Table 3.1. Sample van der Waals radii

Element	Radius (Å)
C	1.60
H	1.20
O	1.40
N	1.50
S	1.95
P	1.90
F	1.35
Zn	1.20
Fe	1.20

Given the connectivity graph of all atoms in the PDB file, pdbgrind can now recognize non-covalently bonded molecules by finding all unconnected subgraphs. This allows automatic extraction of any ligands, water molecules, and metallic ions in complex with the protein. The protein, ligands, and water are saved into separate files.

Metallic cofactors are saved in complex with the protein. For files that contain multi-unit biomolecules, all protein subunits are placed into a single protein file. The largest molecule found in the PDB file is considered the protein. Ligands are differentiated from other protein subunits by a size threshold; ligands are assumed to be small molecules with less than 15% the number of atoms of the largest molecule found in the PDB file.

Inferred connectivity allows this method to extract the ligand deterministically from the geometric data embedded in the file. This obviates the need to rely on inconsistent annotation such as HETATOM records, or in the case of peptidic ligands, chain identifiers for ligand extraction from the PDB file. This also removes errors due to inconsistent CONECT records deposited for ligand HETATOMS. Other methods[75] required manual inspection following ligand extraction due to the error-prone nature of these descriptor fields.

Finally, standard atom connectivity for the 20 amino acids is derived from the atom name field.

3.3.3. Assignment of Bond Order

Inferring bond order using a generalized methodology is complicated by the varying resolution of crystallized structures in the PDB. Crystallographers often describe biomolecules without hydrogens, leaving atoms with unknown valencies. For the standard amino acids, this is not a significant problem as atom and residue naming follow a standard convention thus producing logical bond orders and protonation. For a bound ligand, however, it is left to the discerning user of the PDB file to determine appropriate protonation based on expert knowledge of ligand chemistry.

Pdbgrind encodes this knowledge in a rule-based methodology. Bond order is assigned in a sequential process to individual molecules parsed from the PDB file. All bonds within a molecule begin as single covalent bonds. The algorithm then processes each bond and upgrades the bond order to a double or triple bond based on its chemical environment. Ideal bond length thresholds (Table 3.2) govern the initial round of bond order assignments. First, bonds of appropriate length connecting atoms of linear geometry are assigned triple bonds. Then, bonds of resonant length between double and single bonds are marked as potentially aromatic, feeding an aromatic ring assignment algorithm.

Table 3.2. Ideal bond length thresholds

Bond	Length (Å)
C≡C	1.20
C=C	1.34
C–C	1.54
C≡N	1.17
C=N	1.29
C–N	1.47
C=O	1.20
C–O	1.43
N=O	1.21
N–O	1.40

Cycle detection within the molecular connectivity graph marks all ring atoms. Those atoms marked as ring and connected by an aromatic bond move on to further testing. Aromatic ring systems must also have a planar geometry. To detect this, pdbgrind enumerates all possible sets of five connected ring atoms; if each atom is trigonal planar and the entire five atom set is planar, they are all marked as aromatic. Furthermore, each ring system must satisfy the constraint that half its bonds are of resonant length. The algorithm then propagates alternating single and double bond assignments to those connected aromatic atoms until all sp2 aromatic carbons have received a double bond.

Those remaining bonds that have not been processed as triple or aromatic then undergo a stage of double bond thresholding coupled with functional group recognition. Again, due to the low resolution of certain structures, it may not be apparent from bond length alone that a double bond should exist between two atoms. For these cases, pdbgrind investigates the chemical neighborhood around an atom in an attempt to identify common chemical moieties (Figure 3.1) that will elucidate the correct bond order assignment.

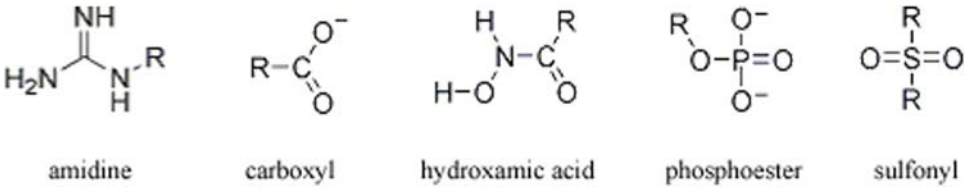


Figure 3.1. Sampling of modeled functional groups

Finally, all leftover carbons with trigonal planar geometry and without double bonds are assumed sp2 and are assigned a double bond to its nearest neighbor.

3.3.4. Protonation

Accurate bond order information is essential for correct protonation of the molecule. Knowledge of the location and orientation of hydrogens is critical for the steric and electrostatic computations necessary in molecular docking. Assuming neutral pH, hydrogens are added according to appropriate hybridization geometries to those atoms with unfilled valencies after bond order assignment. Certain functional groups with non-standard valency (e.g. amidine) are recognized and protonated appropriately.

Since structures are rarely resolved to the precision of individual protons, pdbgrind is also instrumented with the ability to optimize proton placement such that they maximize hydrogen bonding. Figure 3.2 illustrates this feature with the RHA-thermolysin

complex (PDB: 1TLP). Rotamers such as hydroxyls and thiols are sampled for nearby hydrogen bond acceptors within 4Å. In our case, the sampling of the hydroxyl proton (pink spheres) of TYR-157 residue reveals a good hydrogen bond (dashed green line) with a ligand acceptor oxygen. The proton conformation is sampled around the rotatable bond every 15° in search of the strongest hydrogen bond based on the Surflex scoring function.[51] Likewise, proton optimization has been extended to sample imidazole tautomers, deciding between which of the two nitrogen ring atoms (Nδ or Nε) should be protonated to provide the greatest polar interaction. Figure 3.2 contains three such cases where pink spheres show the possible proton placements. HIS-142 and HIS-146 have been protonated such that an N acceptor retains its ability to chelate a metal ion Fe (yellow sphere). The proton assignment in HIS-231 allows it to form a hydrogen bond with an acceptor O on the ligand's sulfonyl group. All flexible proton donors on the ligand are optimized. Only protein protons in the binding pocket are optimized.

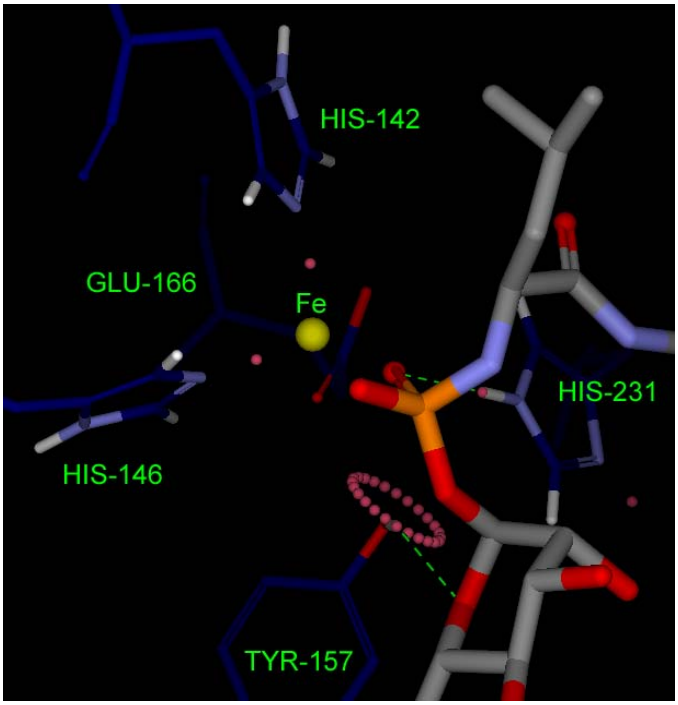


Figure 3.2. Proton optimization of the RHA-thermolysin complex (PDB: 1TLP)

Aromatic ring systems often have multiple solutions to their bond order assignment problem. Nitrogen atoms in particular offer two viable options for aromatic bond assignment: 1) receive a double bond, leaving the N atom a potential proton acceptor, or 2) do not receive a double bond, leaving the protonated N atom a potential proton donor. The initial protonation of the ring system can lend additional evidence towards an optimal assignment. To this end, pdbgrind performs a second pass in its optimization round by analyzing protein-ligand interactions specific to aromatic moieties. Those protons that incur a steric clash suggest their removal. Likewise, atoms that might participate in a significant hydrogen bond if protonated are not assigned double bonds. Bond order assignment is subsequently redone with the inclusion of the additional constraints collected from this optimization round.

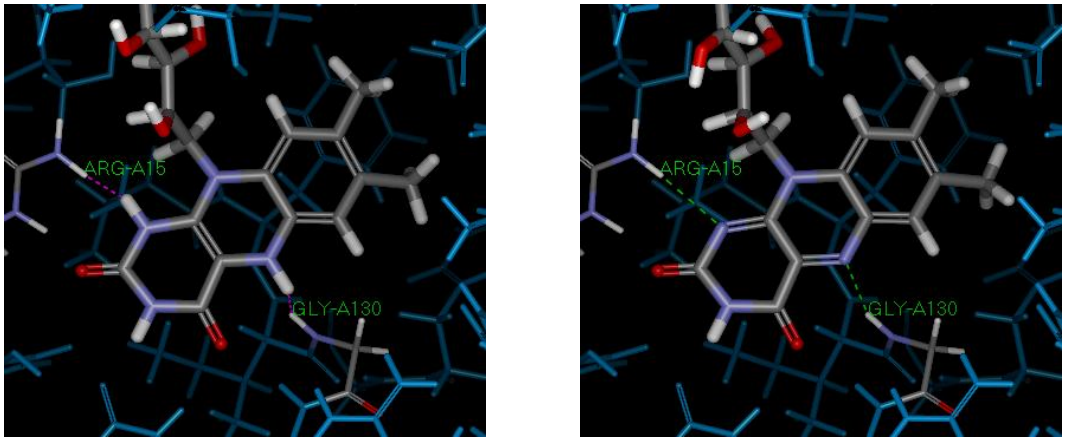


Figure 3.3. Two alternative representations of FMN docked into its receptor

Figure 3.3 illustrates one such example for FMN bound to flavin reductase P (PDB: 1BKJ). On the left depicts a solution to the aromatic bond assignment problem that results in significant steric crashing (denoted in dashed pink lines) between the

ligand and protein at the amidine side chain of ARG15 and the main chain amide of GLY130. Upon detection of this obvious steric hindrance, pdbgrind generates an alternative solution (Figure 3.3, right) absent the offending protons, establishing two favorable hydrogen bonding pairs (denoted in green dashed lines).

3.3.5. Useful Extensions

To avoid redundancy, often only the monomer of a biological unit is saved within the PDB file. However, it is important in docking studies to generate the entire biological unit when the binding pocket is comprised of adjacent monomers in order to depict faithfully the entire set of protein-ligand interactions. To this aim, pdbgrind provides a utility for reconstructing the full biomolecule from the single monomer given the appropriate matrix transformations.

Moreover, molecular docking typically focuses on the binding pocket of a complex to optimize computation time. Hence this method is able to trim the protein saving only those protein atoms within a certain radius of the bound ligand, lowering the memory requirements necessary for evaluation. The method is flexible enough that any sphere of interest defined by a point and radius can be specified to trim the protein appropriately.

In cases where the user disagrees with the bond inference of pdbgrind, the program also provides the facility for assigning bonds a specific bond order using a '`coerce`' command. The molecule is then re-protonated taking into account the the user's changes in bond order.

Detailed usage information is provided in Appendix A.1.1.

3.4. Results and Discussion

Pdbgrind was validated on 800 refined structures from the PDBbind database.[77] This database is a collection of complexes from the PDB (Release No. 103, January 2003) along with their published binding affinity. Each PDB complex in the refined 800 set has been split into protein and ligand mol2 files via the Sybyl software. Protonation of each molecule was assigned but not optimized; the ligand structures have been left unminimized. This set of molecules and binding affinities is an example of a dataset that could be used for developing a scoring function for use in molecular docking. The PDB files used for pdbgrind validation were downloaded directly from the Protein Data Bank (ftp.rcsb.org) since the full biological unit was not consistently represented in the 800 PDBbind dataset.

3.4.1. Conversion accuracy

Accuracy was judged in an automatic fashion. For each complex, the PDBbind ligand was taken as the reference ligand against which all generated ligands from the PDB file were compared. Similar connectivity and atom composition were evaluated using an algorithm that detects graph isomorphs, or a one-one mapping between two sets of vertices. If the generated ligand is an isomorph of the true ligand under bond order constraints, the output ligand of pdbgrind was considered correct. Structures that incurred mismatches in atom number and connectivity were inspected manually and parsed into failure groups. Protein generation was considered robust due to the small variation in named residues, thus simplifying the inference steps necessary to assigning correct bond order and protonation to the molecule.

Pdbgrind was able to generate the correct ligand from the PDB file in 735 of the 800 PDBbind complexes for a 91.8% conversion rate. Of the 735 structures, there were 43 cases where the generated ligand differed from the reference ligand. After manual inspection and literature search, it was determined that the PDBbind reference ligand was in error. Figure 3.2 illustrates such an example. This clearly highlights the difficulty in inferring ligand structure from PDB files and the need for an accurate conversion tool. Similarly, in 41/800 (5%) structures, pdbgrind indicated failure not from the inference engine, but from problems within the PDB file itself. Naming conflicts, bad crystal contacts, and incomplete biological units are just an example of the inconsistencies inherent within the PDB. Pdbgrind produced an incorrect ligand in 24/800 (3%) structures. In these cases, the bond length and atom geometry information were not enough to disambiguate a correct structure, thus forcing the user to inspect the molecule and modify manually errant bonds using the `coerce` command. The space of ligand chemistry is vast; it is unrealistic to expect an algorithm to recognize and process successfully all possible structures. However, it has been shown here that pdbgrind performs adequately on the majority of the ligand chemistry represented in the PDBbind database.

3.4.2. Error sources

Low-resolution structures adversely affect automatic conversion as they often include bond angles and bond lengths that deviate from those expected. Often this will lead to atom-atom crashes of van der Waals shells or atoms incorrectly assigned as covalently bonded. Abnormal bond angles will also disturb the planar geometries expected of sp2 carbons, possibly disrupting aromatic bond formation as in the bent rings

of Figure 3.4. As the number of molecules in the PDB grows, continuing pdbgrind development will attempt to capture emerging patterns of ligand chemistry through the addition of rules to its inference engine.

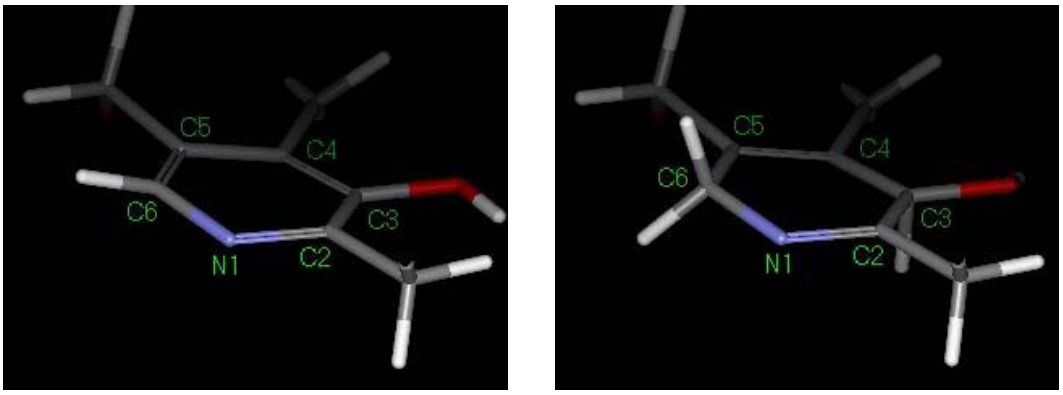


Figure 3.4. A difficult case: holoenzyme cofactor pryidoxal-5'-phosphate (PDB: 9AAT)

A nitrogen atom is oriented out of plane from the six member aromatic cycle. The left figure depicts the reference ligand with a correctly assigned aromatic system with three double bonds: C6=C5, N1=C2, C4=C3. In the right figure, pdbgrind has not recognized the ring as aromatic since the encoded rule that all atoms of an aromatic system must be coplanar has been violated. Instead, it has detected planar sp2 carbons within the ring C5, C4, C2) and assigned them double bonds (C5=C4, N1=C2).

Poorly resolved models can also lead to unpredictable naming conventions by crystallographers. If several ligand poses are equally likely in a binding pocket, a depositor may choose to list the atom coordinates for all possible poses but fail to give each ligand a separate chain identifier. Inconsistent use of the standard vocabulary provided by the PDB makes automatic extraction of the ligand increasingly difficult.

3.5. Conclusion

This chapter described the method by which we generate and uniformly prepare large-scale datasets for our learning algorithm development and validation. Pdbgrind is an open source method that is able to parse protein-ligand complexes from the Brookhaven Protein Databank (PDB)[2] suitable for docking. Using only the geometric information encoded within the standard PDB file, pdbgrind infers proper bond connectivity, bond order, and protonation of the protein and ligand, converting them into the more descriptive Sybyl MOL2 file format. This work includes recognition of recurrent chemical groups and a method of detection and assignment of aromatic ring systems. Furthermore, pdbgrind provides tools for refining the dataset past the stage of simple conversion to include protein trimming and optimization of polar interactions between the receptor and ligand. The freely available pdbgrind was designed to provide the molecular docking field a high-throughput method for uniformly producing large, high-quality datasets for use in algorithm development, validation, and benchmarking.

This chapter highlighted how we handled some of the data demands of this research. The next chapter will use this data to validate modifications to the search strategy used in the Surflex molecular docker.

Chapter 4
Enhanced Protomols

4.1. Abstract

The search strategy employed by Surflex-Dock aligns ligand fragments to a "protomol" to generate good starting points for whole molecule docking. The protomol acts as a negative image of the binding pocket. By employing protomols with more ligand-like features, we seek to improve docking accuracy. Validation of this hypothesis was done on a benchmark of 81 complexes. Ligands were assigned random starting conformations and then docked using both default and enhanced protomols. Results indicate that protomol augmentation can lead to better search of ligand pose space.

4.2. Introduction

This chapter will explore work conducted in the search space of molecular docking. Here we attempt to discover the correct binding mode of a ligand within a protein pocket when starting from a random ligand conformation. How well one searches ligand pose space intimately affects one's ability to score that ligand. If the scoring function never sees poses that are close to correct, it will never see the ideal interactions possible between the protein and ligand. Attributing a binding energy prediction to such a

complex becomes very difficult. Conversely, a good scoring function will guide a docker to seek out correct binding modes. In this field, search and scoring are the backbone of algorithm development – for if a molecular docker is to be successful, it must do both well.

As described in Section 2.2 of the Literature Review, there are many strategies for exploring the degrees of freedom available to a ligand within a binding pocket. Surflex uses a molecular similarity technique[82] to align putative ligand poses to a protomol.[30] Morphological similarity between two molecules can be defined as a function of surface distances from molecular features as measured from a set of observer points placed on a uniform grid. Consider Figure 4.1 which depicts amidst a set of observer points the nicotine molecule on the right and a competitive nicotine agonist on the left. Note the observed distances from the hydrophobic surface (grey), a hydrogen bond acceptor (red), and a hydrogen bond donor (blue). An intuitive superposition of the two molecules is pictured with blue surfaces. Both molecules exhibit very similar hydrophobic and polar features.

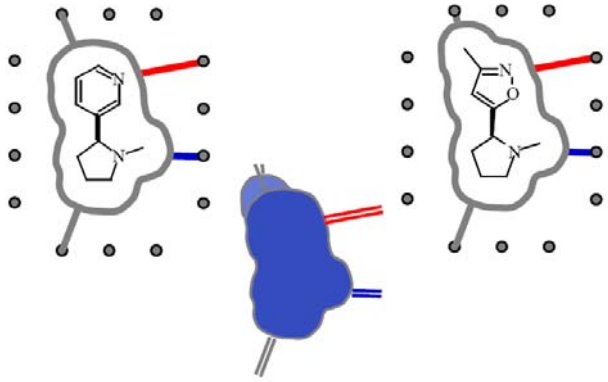


Figure 4.1. Surface features as seen from a set of observer points for two molecules

Nicotine is on the left whereas a competitive nicotine agonist is shown on the right. A good superposition of the two molecular surfaces and their features is shown in blue.

Now consider Figure 4.2 where our synthetic agonist (top) and nicotine molecule (bottom) are placed in a random conformation. If two molecules are at all similar, there must exist a set of corresponding observer points that are seeing the same features at similar distances. To arrive at a proper superposition of the two, we must find corresponding triplets of observer points that "see" the same things at similar distances. Additionally, observer points must take into consideration the directional component of a polar surface moiety. The transformation that superimposes the two triangles offers an efficient means of generating a good alignment.

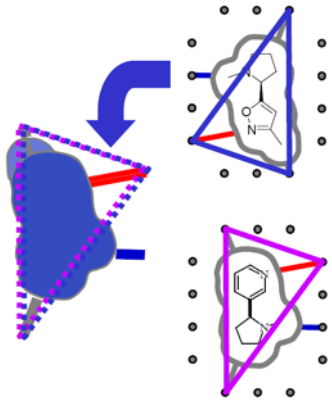


Figure 4.2. Alignments generated by finding triangles of similar size and composition

Surflex uses a protomol to efficiently cull the infinite search space available to a ligand. Within the protein pocket, this protomol represents an idealized ligand – it fills the pocket completely and participates in perfect hydrogen-bonds with every available polar protein atom. Aligning ligand fragments to such a model provides the docker with good starting points from which it can incrementally build the entire molecule. The work in this chapter will investigate the possibility of improved search when the protomol becomes more ligand-like in appearance.

4.3. Methods

A platform for experimenting with enhanced protomols was developed. We will describe a general overview of its implementation; detailed code documentation for all functions and data structures can be found in Appendix B.1.2. Recall our central motivating idea that alignment might benefit from a protomol which displays more ligand-like features.

Protomols are composed of small molecular fragments that are placed in the binding pocket. Interesting protein residues are first marked using a distance threshold

from the bound conformation of a co-crystallized ligand. Unoccupied space in the pocket is identified by connecting all pairs of marked residues with lines. As the lines are traversed, voxel (3D point within the binding pocket) scores are incremented. High scoring voxels are kept as starting points for placing our molecular fragments. These fragments "probe" the protein space around the voxel, looking for interesting interactions as measured by our scoring function. Multiple orientations are sampled and the probes are optimized using the Surflex scoring function.

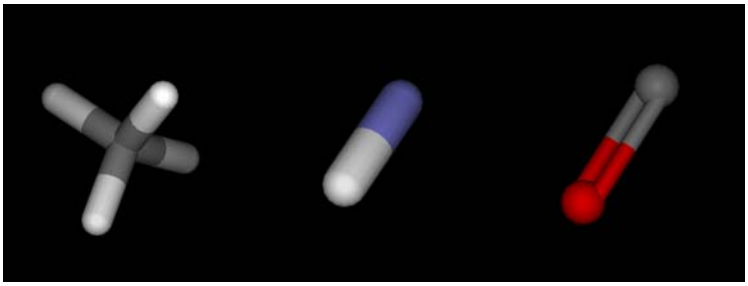


Figure 4.3. Small probes: CH_3, NH, C=O.

Originally, protomols were comprised of three simple probe types: CH_4, N-H, C=O (Figure 4.3). Together, these sampled respectively the hydrophobic, hydrogen bond accepting, and hydrogen bond donating regions of the binding pocket. The high scoring probes are kept and collectively form the protomol. An example using our small probes is shown for streptavidin on the left in Figure 4.4 (hydrogens not shown for CH_4 probes). The native ligand biotin is shown in green, while its 2D structure is in yellow. Note that the protomol was generated using only the protein structure; the native ligand served only to identify proximal residues within the binding pocket. Not only do the probes faithfully capture all of the streptavidin-biotin polar interactions, they also find additional interactions within the pocket not made by biotin.

Since all probes scoring higher than a given threshold are saved, the original protomol (left, Figure 4.4) has several probes acting as exemplars for the same hydrogen bond. When we think of our alignment protocol, recall that we are looking for matching surface regions as seen from external observer points. Having a cluster of probes of the same polar type creates a general polar area on the surface of the protomol. This in turn leads to a larger number of possible alignment matches of the ligand to that region of space.

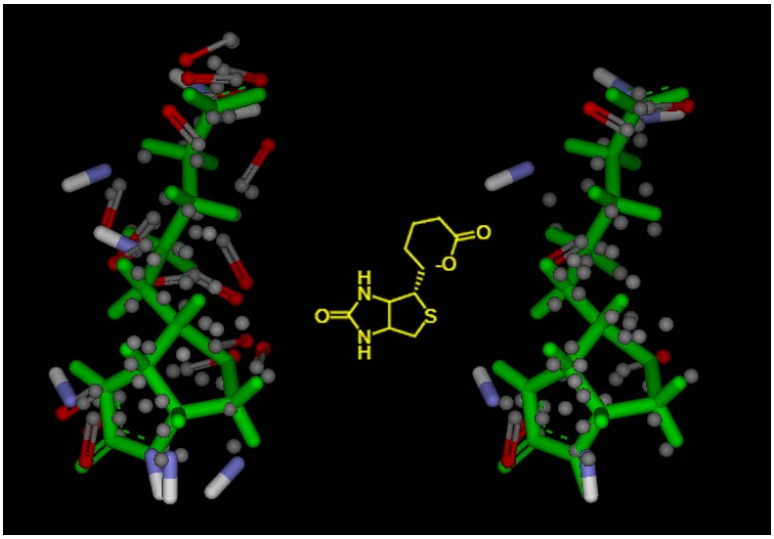


Figure 4.4. Protomol for streptavidin (PDB: 1STP) before and after probe redundancy elimination.

4.3.1. Probe redundancy elimination

Here we introduce our first upgrade to the protomol: probe redundancy elimination. By saving only the best scoring polar probe for any particular protein atom in the binding pocket, we keep only perfect hydrogen bond exemplars. Thus, probes will be oriented at an optimal direction and distance from a corresponding protein polar atom. Similarly, CH_4 probes, which sample the hydrophobic surface of the protein, are culled based on similar rmsd. These enhancements enforce a level of specificity in a matching

ligand alignment that was not possible with the original protomol. Recall our streptavidin protomol from Figure 4.4. The protomol on the right has eliminated redundant probes, going from 54 CH_4 probes to 51, 8 N-H probes to 5, and 19 C=O probes to 6. The end result is a protomol that exhibits more realistic features, much like an actual ligand.

4.3.2. Larger molecular fragments

The second introduction to the protomol is the inclusion of larger, more feature-rich probes. We augment the set of probes available to the protomol to include the following molecular fragments: carboxyl (carb) and amidine (amid) (Figure 4.5). These new probes mimic recurring moieties often seen in ligand chemistry. In practice, the amid probe is replaced by AMN-T and AMN-Y probes. Both are able to capture all of the interactions available to a single amid probe. This eases computational complexity as well as increases the likelihood of probe inclusion in the protomol.

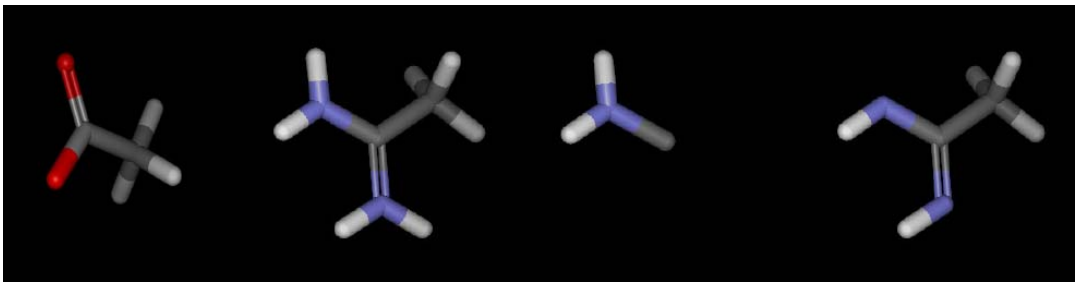


Figure 4.5. Big probes: CARB, AMID, AMN-T, AMN-Y

The major advantage held by using larger probes over smaller probes is the ability to form bidentate interactions. As such, all larger probes are required to participate in two or more hydrogen bonds to be considered for the protomol. Furthermore, bigger probes take precedence over smaller probes making the same interactions. Thus, several small probes can be replaced by a single large probe as in Figure 4.6. The aromatic ring system

of methotrexate is shown interacting with ASP-27 of dihydrofolate reductase (PDB: 4DFR). On the left we observe several high scoring donor probes pointing out the available hydrogen bonds offered by the aspartic residue. On the right, all six of the donor probes have been replaced by a single AMN-Y probe. It should be stressed that only information provided by the protein is used to generate the protomol. Given that no knowledge of the native ligand structure was used to generate these probe placements, note the accuracy with which the AMN-Y probe mimics the bidentate interaction of the crystallized cognate ligand.

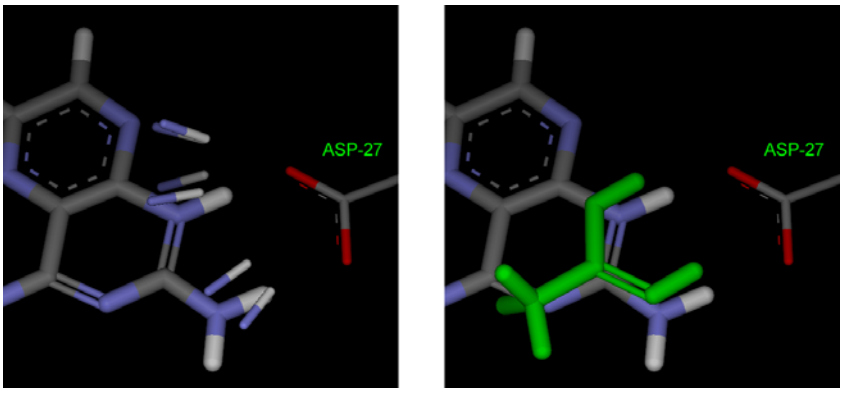


Figure 4.6. AMN-Y probe replacing several donor probes in the DHFR protomol (PDB: 4DFR).

4.3.3. Dataset and validation experiment setup

The creation of a protomol with rich, familiar moieties from ligand chemistry should speed the search of ligand pose and alignment space. With more specific, ligand-like features, poor alignments to the protomol can be avoided, leading to better starting points for docking molecules. To test this hypothesis, we utilized a previously published benchmark: the 81 complex set,[30] a derivative of the GOLD dataset[38] that has been pre-filtered to include only ligands with: (1) 15 of fewer rotatable bonds; (2) no covalent bonds with the protein; (3) no obvious errors in structure. Each complex is represented by

a protein and a cognate ligand. Ten random conformations were generated from the minimized ligand structure.

This dataset will be used to gauge Surflex's docking accuracy and pose recognition using the default and enhanced protomols. Default protomols were generated using Surflex-Dock v2.11;[70] enhanced protomols used Surflex-Dock v2.203 with the option `-fancy`. All protomols were generated using the option `proto_bloat 0.5`. Each of the 10 random ligand conformations were docked twice into the protein using Surflex-Dock v2.11; once using the default protomol and once using the enhanced protomol. Typically for any docking run, Surflex-Dock outputs the 10 highest scoring docked poses. We used the `–ndock_final 1` option to output only the single highest scoring pose. This docked pose is then evaluated for correctness by rmsd to the crystallographic pose of the cognate ligand.

All code implementing the described protomol development is linked against the Surflex Library v2208. It is available freely to investigators at www.jainlab.org/Surflex. Detailed usage information and code documentation is provided in Appendix B.

4.4. Results & Discussion

Table 4.1 summarizes the performance of the default and enhanced protomols in accurately docking 10 random poses of each ligand in the 81 complex set. The proportion of dockings within 2.0Å rmsd of the crystallographic pose is reported. The table is sorted by the difference in proportion between the enhanced and default protomols. In 35 cases, we see that use of the enhanced protomol has improved the docker's ability to recapitulate the experimental pose; in 30 cases performance remains the same; and in 16 cases the default protomol is better. Thus, in 65/81 cases the enhanced protomol performance is equal to or better than the default performance. Clearly, enhanced protomols are not worse than the default protomols ($p << 0.001$ by exact binomial). The converse however is not true. In 46/81 cases, the default protomols are as good as or better than the enhanced protomols, which allows for the possibility that they are worse. Given that there are interactions between protomol characteristics and internal parameters that control search within Surflex, it is likely that optimization of these parameters in the context of the enhanced protomols will yield a more substantial improvement in performance.

Table 4.1. Proportion of dockings with rmsd < 2.0Å for the 81 complex set

		proportion rmsd < 2.0A		
pdb	nrot	default	enhanced	diff
1bma	14	0.2	0.9	0.7
1dbb	1	0.5	1.0	0.5
1rob	6	0.1	0.6	0.5
2r07	8	0.0	0.5	0.5
1tnl	1	0.1	0.5	0.4
1frp	8	0.4	0.8	0.4
1tka	8	0.2	0.6	0.4
1lna	1	0.6	0.9	0.3
4dfr	6	0.6	0.9	0.3
1dbj	9	0.6	0.9	0.3
1ukz	10	0.5	0.8	0.3
1hdc	6	0.3	0.6	0.3
2dbl	6	0.6	0.8	0.2
1acm	4	0.0	0.2	0.2
1atl	7	0.3	0.5	0.2
2lgs	11	0.2	0.4	0.2
1dwd	11	0.0	0.2	0.2
1fkg	11	0.1	0.3	0.2
1phg	3	0.4	0.6	0.2
4cts	3	0.7	0.8	0.1
6abp	4	0.7	0.8	0.1
3hvt	1	0.3	0.4	0.1
1lpm	7	0.0	0.1	0.1
1baf	8	0.0	0.1	0.1
1cbs	0	0.4	0.5	0.1
1hsl	1	0.9	1.0	0.1
1hyt	3	0.4	0.5	0.1
1trk	3	0.8	0.9	0.1
6rsa	3	0.6	0.7	0.1
1aha	4	0.9	1.0	0.1
1coy	5	0.9	1.0	0.1
1lcp	5	0.8	0.9	0.1
3tpi	5	0.9	1.0	0.1
1tni	7	0.2	0.3	0.1
1cbx	8	0.6	0.7	0.1
1etr	0	0.0	0.0	0.0
1hri	0	0.2	0.2	0.0
1epb	0	0.0	0.0	0.0
1snc	1	0.0	0.0	0.0
1mrg	2	0.9	0.9	0.0
8gch	3	0.2	0.2	0.0
1aco	3	0.9	0.9	0.0
1bbp	3	0.0	0.0	0.0
1dr1	4	1.0	1.0	0.0
1fen	4	1.0	1.0	0.0
1hdy	4	1.0	1.0	0.0
1lah	4	1.0	1.0	0.0
1lic	4	0.0	0.0	0.0
1lst	5	1.0	1.0	0.0
1mdr	5	1.0	1.0	0.0
1nco	5	0.1	0.1	0.0
1stp	5	1.0	1.0	0.0
1tng	6	1.0	1.0	0.0
1ulb	6	0.9	0.9	0.0
1wap	7	1.0	1.0	0.0
2cgr	7	0.6	0.6	0.0
2gbp	8	1.0	1.0	0.0
2phh	9	1.0	1.0	0.0
2sim	9	1.0	1.0	0.0
3aah	10	1.0	1.0	0.0
3cpa	10	0.0	0.0	0.0
6rnt	11	0.3	0.3	0.0
7tim	11	1.0	1.0	0.0
1abe	14	0.9	0.9	0.0
1tmn	15	0.0	0.0	0.0
1fki	0	0.9	0.8	-0.1
1ldm	1	1.0	0.9	-0.1
1mrk	1	0.9	0.8	-0.1
2cht	3	0.9	0.8	-0.1
2ada	5	1.0	0.9	-0.1
3ptb	6	1.0	0.9	-0.1
1eap	11	0.1	0.0	-0.1
1srj	4	0.4	0.3	-0.1
2cmd	4	0.7	0.5	-0.2
1com	6	1.0	0.8	-0.2
1rds	11	0.7	0.5	-0.2
1glq	15	0.2	0.0	-0.2
2ak3	6	0.7	0.4	-0.3
1ack	3	0.7	0.2	-0.5
2ctc	4	0.8	0.3	-0.5
1acj	0	0.7	0.1	-0.6

Figure 4.7 illustrates how our protomol modification might reduce the ligand search space within the binding pocket. Depicted in green is the bound conformation of BMA inhibitor to porcine pancreatic elastase (PDB: 1BMA). According to

Table 4.1, 90% of the random ligand conformations docked using enhanced protomols resulted in poses within 2.0Å rmsd from correct vs. only 20% with the default protomols. The default and enhanced protomols are presented in the top row left to right, respectively. Shown in the bottom row is the docked conformation using the default (left) and enhanced protomol (right). Both docked poses were generated from the same initial random configuration.

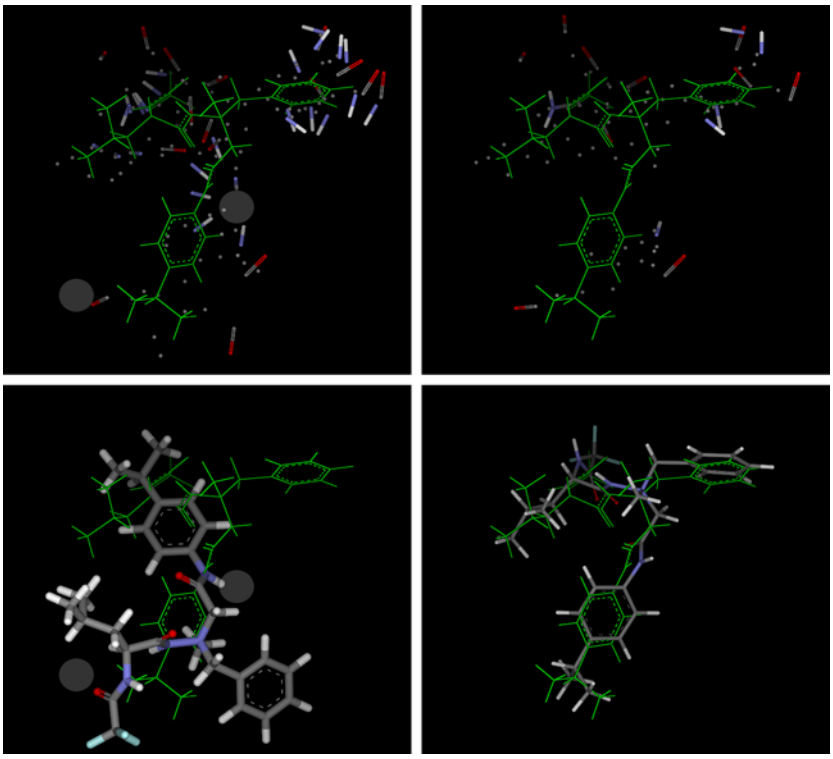


Figure 4.7. Example dockings using default and enhanced protomols

Pictured here is benzyl methyl aminimide inhibitor docked into porcine pancreatic elastase (PDB: 1BMA). The crystallographic pose is shown in green. The top row shows the default protomol (left) and the enhanced protomol (right). The bottom row illustrates the docked pose generated *from the same initial random conformation* using the default protomol (left) and the enhanced protomol (right). The rmsd of the docked poses are 9.3Å and 1.3Å, respectively.

This receptor site is largely open and exposed to solvent. The ligand is oriented in space such that we are looking into the pocket. Such a cavernous space means that there are fewer steric constraints imposed on the ligand pose by the protein. Coupled with the large BMA inhibitor molecule (14 rotatable bonds), there are an infinite number of possible poses. A protomol culls this search space by providing an archetype of suitable interactions. The default protomol is a vague model with polar regions overrepresented by numerous probes of the same type. For a large, flexible ligand, this complicates search as many alignments will score well; there will be many local minima in pose space that a search may get lost in. The docked pose returned using the default protomol seems to have fallen into such a well. Note the translucent circles in the left column of pictures in Figure 4.7 – these represent hydrogen bond interactions that the docked pose considers important, but are absent in the experimental pose. The default protomol offered a variety of possible starting alignments; the docker unfortunately followed this particular one to a dead end in search space. Local optimization using the scoring function can only move poses so much. The end result is a poor pose (rmsd = 9.3Å) that scored 2 orders of magnitude less than the highest scoring pose returned for this complex.

Conversely, a protomol that captures specific interactions can guide alignment to the most important interaction orientations. The enhanced protomol offers a single, perfectly oriented probe for each polar interaction with the protein. Unless the putative ligand pose aligns its polar moieties in a similar orientation, it will not be able to use that hydrogen bond as a starting point. This effectively places a constraint on pose space that limits the starting alignments to only those with very good hydrogen bond interactions. For a large molecule in an open pocket, having fewer choice matches for hydrogen

bonding can efficiently lead to more reasonable poses. The one shown in Figure 4.7 achieves an rmsd of 1.3Å and a score about 1 order of magnitude less than the highest scoring pose.

This result leads us to believe that enhanced protomols may function better for larger ligands and less so for smaller ligands with fewer rotational bonds to sample. This bears out to an extent when we examine the rotatable bonds column of

Table 4.1. Of the 16 complexes where the new treatment is outperformed, 13 have 6 of fewer rotatable bonds. If we plot the distribution of good dockings for proteins whose ligands have more than 6 rotatable bonds, we see a separation between the default and enhanced protomols (Figure 4.8). Recall that a good docking is defined as one where the docked ligand pose is within 2.0Å rmsd of the crystallographic pose. Of the 28 large ligand dockings, 25% of the cases using enhanced protomols resulted in a success rate of 70% or greater. Only 10% of the cases using default protomols can say the same. This evidence suggests that employing a hybrid strategy of enhanced protomols for large ligands and default protomols for small ligands could benefit search for this molecular docker.

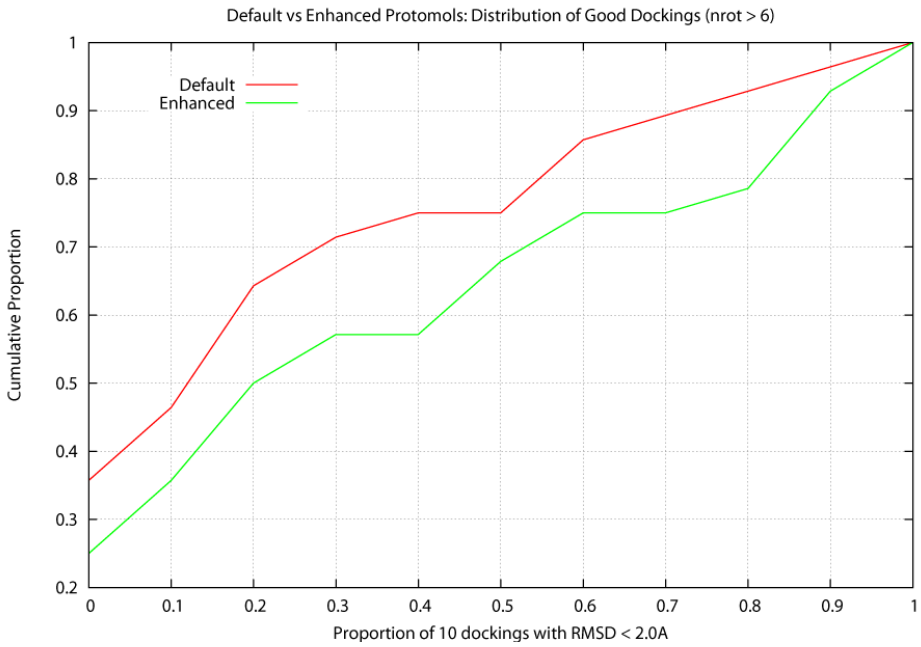


Figure 4.8. Distribution of good dockings for large ligands (number of rotatable bonds > 6) from the 81 complex set

4.5. Conclusion

By implementing a more feature-rich protomol that better captures specific ligand chemistry, we sought to improve the search of ligand pose space during docking. Validation results on the 81 complex set indicate incremental improvements are possible with a more realistic looking protomol, particularly when docking more flexible ligands. Further study is necessary to make the strong statistical arguments necessary to suggest methodological changes to Surflex's search engine. To this end, the open source enhanced protomol code was written for extensibility to aid future development. Its documentation is located in Appendix B.1.2.

The protomol strategy of "probing" space for interesting interactions is an interesting technique with wide-ranging applications. One such example is in the field of ligand-based modeling where a receptor structure is unknown. One could potentially probe the molecular surface of a known binder or superposition of many binders to recreate a pseudo protein pocket. Used as a proxy for the receptor structure, this "protopocket" could then screen for new unknown ligands.

More generally, any search problem benefits from additional information about the space. We will revisit this idea again in the following chapters as we seek to comb the parameter space of a function in order to optimize it.

Chapter 5
Refining Protein-Ligand Scoring Functions Using Negative Data

5.1. Abstract

Several of the most effective tools for small molecule docking employ empirically derived scoring functions to rank putative protein-ligand interactions. Among these programs, Surflex-Dock has been shown to be competitive with the best programs in geometric docking accuracy and, in limited tests, been shown to be superior in terms of screening utility. The scoring function employed by Surflex was developed purely based on *positive* data, comprised of non-covalent complexes of proteins and ligands with known binding affinities. Consequently, scoring function terms for improper interactions received little weight in parameter estimation, and an *ad hoc* scheme for avoiding protein-ligand interpenetration was adopted. In order to construct a more rigorous scoring function, a generalized method for incorporating synthetically generated *negative* training data was developed, which allows for direct estimation of all scoring function parameters. Geometric docking accuracy remained excellent under the new parameterization over a large number of complexes. In addition, a broad test of screening utility in 29 cases,

covering a diverse set of proteins and corresponding ligand sets, showed improved performance. Maximal enrichment of true ligands over non-ligands exceeded 20-fold in over 80% of cases, with enrichment of greater than 100-fold in over 50% of cases. Further generalization of this approach, with the inclusion of additional positive *and* negative data, should yield more complex scoring functions for molecular docking with substantially improved performance. Note: the work presented in this chapter was published in 2006.[83]

5.2. Introduction

In the review of the literature in Section 2.2.1, the problem of scoring protein-ligand interactions has typically been addressed with either a physics-based or empirical approach. Both strategies attempt to understand the forces that stabilize and destabilize binding. Physics-based strategies usually leverage an established molecular mechanics force field, whereas empirical strategies will glean the important forces from observing actual protein-ligand data. In the development of such scoring functions, only *positive* data have generally been used, encompassing protein-ligand complexes with known binding affinities. One consequence of this choice is that repulsive terms, which include effects such as improper steric clashes, same charge atomic interactions, and desolvation penalties, can receive little weight. Physics approaches will embed these penalties in the form of the Lennard-Jones potential, Coulomb's Law, and complicated solvation models such as Generalized-Born. The problem with an empirical approach using only positive data is that the training ligands generally fit well within protein active sites; they typically do not make same charge close contacts, and do not bury hydrophobic ligand surfaces against hydrophilic protein surfaces (or vice-versa). Thus, repulsive forces will rarely be

observed; the inductive bias of the training regime used for the Surflex scoring function is such that if it is not observed, it must be unimportant. Yet we know these repulsive forces exist – how can we force the scoring function to see them? In this chapter, we introduce the idea of employing *negative* data in training scoring functions for molecular docking.

There are three obvious constraints that can be used in the context of estimating parameters for a scoring function: 1) that the computed scores for ligands of known geometry correspond closely to the known affinities of the ligands, 2) that the computed scores for the highest-scoring poses of non-ligands be poor relative to some value, and 3) that the computed scores for geometrically incorrect dockings of ligands be poorer than the score for poses that are very close to correct. The first constraint is the typical use of positive training data, recently made more robust by the availability of large numbers of such complexes from databases such as PDBbind.[77] The second constraint carries with it two questions. What should be the source of non-ligands, and what should be the value of the bound on score? The third constraint offers a direct method for tuning scoring functions using protein-ligand complexes where the binding affinity is *not known*. Where a particular scoring function incorrectly identifies a ligand pose as scoring better than the correct pose, a dynamic constraint can be computed to penalize the incorrectly scored pose.

In this chapter, we combined the first two types of constraints to re-estimate parameters for the Surflex-Dock scoring function. For the positive data, we employed the same 34 complexes used originally in parameter estimation (we did not make use of larger, newer data sets in order to avoid complications with testing the method). For the negative data, we screened a random compound library against each of the proteins in the

positive data set and retained ligands that scored better than a predetermined value but which, based on molecular similarity, did not look at all like the native ligands of the proteins. In employing the negative data, we imposed a penalty on the objective function for parameter optimization if negative ligands exceeded a fixed score.

We developed a large set of screening test cases, totaling 29 sets of proteins with associated true positives, which cover a very diverse set of proteins active sites and corresponding ligand properties. The newly formulated scoring function obviated the need for *ad hoc* treatment of improper clashes, and screening enrichment remained the same or improved in 21/29 cases. Maximal enrichment of true ligands over non-ligands exceeded 20-fold in over 80% of cases, with enrichment of greater than 100-fold in over 50% of cases. In the 6 cases of poorest performance by the new scoring function, use of multiple protein conformations exhibited promise in improving screening enrichment. We also established that docking accuracy was essentially unchanged with the new scoring function using a set of 81 protein-ligand complexes.

By simply adding automatically generated negative data to the training of the Surflex-Dock scoring function, we were able to estimate parameters that previously received so little weight that *ad hoc* terms were required to make use of the function in docking. The new scoring function yielded excellent performance, over a wide variety of test cases, both in terms of docking accuracy and in terms of screening utility, without requiring knowledge-based post-processing of docking scores to incorporate interpenetration values. Further generalization of this approach to estimate additional parameters (e.g. involving desolvation effects), with the inclusion of more positive *and*

negative data, should yield more complex scoring functions for molecular docking with substantially improved performance.

Surflex-Dock is available free of charge to academic researchers for non-commercial use (see http://www.jainlab.org/downloads.html for details on obtaining the software). The data sets used for benchmarking are freely available to all researchers via the same web site.

5.3. Methods

The focus of the work is on improving the treatment of the repulsive terms of the Surflex-Dock scoring function. We will briefly review the scoring function, since additional details are presented elsewhere.[51] We employed multiple sources of data to construct test cases for screening enrichment, and we employed our previous benchmark of 81 complexes to assess docking accuracy. The following reviews the scoring function, the data sets and preparation, the optimization procedure for re-tuning the scoring function, and the procedures for assessment of performance.

5.3.1. Scoring Function

The Surflex-Dock scoring function (originally used within Hammerhead[84]) was tuned to predict the binding affinities of 34 protein/ligand complexes, with its output being represented in units of $-\log(K_d)$.[51] The range of ligand potencies in the training set ranged from 10^{-3} to 10^{-14} and represented a broad variety of functional classes. The parameterization of the function models the non-covalent interactions of organic ligands with proteins, including proteins with bound metal ions in their active sites. The function is continuous and piecewise differentiable with respect to ligand pose, which is important

for the gradient-based optimization procedures employed within Surflex-Dock. The terms, in rough order of significance, are: hydrophobic complementarity, polar complementarity, entropic terms, and solvation terms (negligible). The full scoring function is the sum of each of these terms.

The dominant terms are the hydrophobic contact term and a polar contact term that has a directional component and is scaled by formal charges on the protein and ligand atoms. These functional terms are parameterized based on distances between van der Waals surfaces, with negative values indicating interpenetration. Each atom on the protein and ligand is labeled as being nonpolar (e.g. the H of a C-H,) or polar (e.g. the H of an N-H or the O of a C=O), and polar atoms are also assigned a formal charge, if present. Figure 5.1 shows plots of the hydrophobic term and the polar term for a hydrogen bond. The hydrophobic term (bottom curve, in red) yields approximately 0.1 units per ideal hydrophobic atom/atom contact. A perfect hydrogen bond yields about 1.2 units and has a peak corresponding to 1.97Å from the center of a donor proton to the center of an acceptor oxygen (learned based entirely on the empirical data and corresponding quite closely to the expected value range). Despite the large difference in the value of a single hydrophobic contact versus a single polar contact, the hydrophobic term accounts for a *larger* total proportion of ligand binding energy on average. This is because there are many more hydrophobic contacts than ideal polar contacts in a typical protein-ligand interaction.

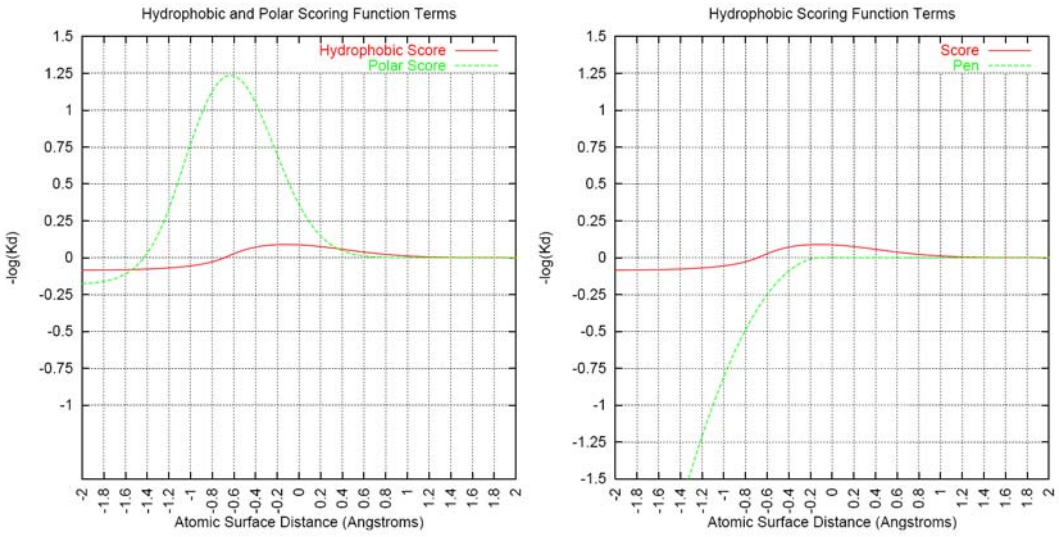


Figure 5.1. The default scoring function.

Left: hydrophobic and polar terms. Right: hydrophobic term and adhoc penetration term

Apart from the hydrophobic and polar terms, the remaining important terms include the entropic term and the solvation term. The entropic term includes a penalty that is linear in the number of rotatable bonds in the ligand, intended to model the entropic cost of fixation of these bonds, and a term that is linearly related to the log of the molecular weight of the ligand, intended to model the loss of translation and rotational entropy of the ligand. The solvation terms are linearly related to a count of the number of missed opportunities for appropriate polar contacts at the protein-ligand interface.

However, neither the solvation term nor any of the terms intended to guard against improper clashes received much weight in the original training (the solvation term was, in fact, 0.0). This was due to the fact that no negative data were employed; only ligands with their cognate proteins were used in parameter estimation, thus there were essentially no data from which to induce such penalty terms. In particular, the linear weights on the terms for improper steric clashes, non-complementary polar contacts, and

solvation effects were, respectively, -0.08 (l_1 in the original paper), -0.15 (l_5), and 0.0 (l_6)**.** All of these were very small relative to, for example, the magnitude of a single ideal hydrogen bond (1.23). In order to make use of the scoring function for molecular docking, it was necessary to superimpose a term to prevent atomic overlap between the protein and ligand (and within the ligand itself): $-10.0(d_{ij}+\delta)(d_{ij}+\delta)$. In this equation d_{ij} is the distance between atomic surfaces (negative for surfaces that interpenetrate), and δ was 0.1 for all contacts except those between complementary polar atoms, where δ was 0.7. In the re-implementation of the Hammerhead scoring function for Surflex-Dock, this term was normalized to a value called "pen" by multiplying by 4.0 and dividing by the number of atoms in the ligand. A docked ligand yielded two values, score and pen. The user was required to choose a cutoff for pen beyond which a ligand was rejected (or alternatively construct a combination score by weighting the two terms). The formulation of the term was unsatisfying since the parameters were chosen in a largely arbitrary fashion, and the requirement for selecting a threshold for interpenetration made for an extra methodological complexity.

In this work, we sought to address this term in a systematic fashion by making use of negative training data. However, this required that the new penetration term be treated in an absolute sense, both with respect to protein-ligand interactions and with respect to ligand self-interpenetration. The latter required a change in the internal computation of self clashing, eliminating atom pairs from consideration if they were connected by non-rotatable bonds. Without this modification, ligands with, for example, bicyclic ring systems, were at a disadvantage relative to other ligands due to the inherent nominal clashing among atoms within constrained covalent systems. Earlier versions of Surflex-

Dock used a heuristic method to estimate the best possible self-penetration for each ligand and normalized the self-penetration by subtracting this value, but this estimate was not sufficient for systematic parameter tuning. Also, in order to obtain the most reliable final scores for docked ligands, the final gradient-based ligand pose optimization was enhanced in thoroughness to ensure convergence of the scoring function. Incomplete convergence would effectively add noise to the scores of ligands during scoring function optimization as well as during the evaluation of the methodology.

Software versions: Surflex v1.24 was in widest release prior to this work, and was used in data set generation and for certain control experiments. This version implemented the original Hammerhead scoring function, as described above, with the *ad hoc* interpenetration treatment. Versions up to 1.28 continued to use this formulation. The modified scoring function, with the new treatment of penetration values, more aggressive gradient-based pose optimization, and a switch to select the original scoring function, begin with Surflex-Dock versions 1.31 and higher. The experiments and results described herein may be reproduced using the optimization software described in Appendix C.

5.3.2. Negative Data Sets

Two sources were used for nominal negative ligands. The screening data set from the comparative paper of Bissantz et al.[67] was used, as in our previous report.[30] The original data set included 990 randomly chosen non-reactive organic molecules chosen from the ACD ranging from 0 to 41 rotatable bonds. The data set was used with two modifications. First, all ligands were subjected to an automatic protonation procedure and energy minimization in order to eliminate differential bias between positive and negative ligands (positives were treated the same, see below). Second, ligands with greater than 15

rotatable bonds were eliminated, resulting in 861 negative ligands. This eliminated decoys that were clearly not drug-like and better reflected the composition of the positive ligands.

The second source was ZINC (see http://blaster.docking.org/zinc). We randomly selected 1000 compounds from the drug-like subset (1,847,466 total) of the 07-26-2004 version of the database. These compounds had molecular weight ≤ 500, with computed $\log_p \leq 5$, h-bond donors ≤ 5, and h-bond acceptors ≤ 10. The compounds were processed identically to the ligands above, and the number of rotatable bonds in the set ranged from 0 to 12. In the remainder of the chapter, "negative ligand set" refers to the 861 compound set derived from Bissantz et al.[67] unless otherwise noted specifically as the "ZINC negative set."

5.3.3. Training Data Set

Re-estimation of the scoring function parameters relating to improper interactions required *both* positive and negative data; otherwise the scoring function could be trivially modified to include very large penalty terms. In order to simplify evaluation of the new function, we employed the 34 complex training set that was used in constructing the original Hammerhead scoring function.[51] All of the complexes dated from 1992 or earlier, reducing the possibility that the training set could contain information relevant to our tests, which were based largely on more recent data. The original complexes were converted from PDB to Sybyl mol2 format and protonated per expectation at physiological pH, with active site rotamers of hydroxyls and thiols and tautomers of imidazoles optimized for cognate ligand interactions.

Table 5.1. Training Data Set

complex	ligand	N negative	pKd
7cpa	ZFVp(O)F	78	14
1stp	biotin	46	13.4
6cpa	A-ZAAp(O)F	115	11.52
4tmn	ZFpLA	72	10.19
4dfr	methotrexate	115	9.7
4phv	L700,417	207	9.15
1dwd	NAPAP	92	8.52
5tmn	ZGpLL	100	8.04
2gbp	galactose	1	7.6
1etr	MQPA	66	7.4
1tlp	phosphoramidon	87	7.33
1tmn	CLT	78	7.3
1rbp	retinol	115	6.72
1ppc	NAPAP	27	6.46
5tln	HONH-BAGN	96	6.37
1pph	3-TAPAP	34	6.22
1ett	TAPAP	121	6.19
1phf	4-Phe-imidazole	141	6.07
4dfr*	2,4-diaminopteridine		6
5cpp	adamantone	3	5.88
2xis	xylose	0	5.82
2ifb	C15COOH	173	5.43
1ulb	guanine	67	5.3
2ypi	phoshoglycilic acid	45	4.82
3ptb	benzamidine	28	4.74
2phh	*p-hydroxybenzoate*	58	4.68
2tmn	PLN	101	4.67
3ptb*	phenylguanidine		4.14
1dwd*	amidinopiperidine		3.82
4tln	Leu-NHOH	98	3.72
3ptb*	benzylamine		3.42
4cha	indole	0	3.1
1dwb	benzamidine	110	2.92
3ptb[α]	butylamine		2.82

***** Indicates that the respective protein contributes more than one ligand to the training set.

[α] Indicates that the respective ligand was docked in or generated by a direct modification of the native ligand in the complex.

For the negative data, we employed the negative ligand set above (restricted, for computational efficiency, to the 600 least flexible molecules). For each of the protein structures of the 34 complex positive data set, we docked all negative ligands using Surflex-Dock version 1.24, using default parameters. Ligands that scored greater than 4.0

(in units of pK_d, ignoring penetration values) were presumed to be false positives. To reduce the likelihood of including a true ligand as a negative in the training set, we further screened the ligands based on molecular similarity to the bound pose of the native ligand for each protein using Surflex-Sim.[85] Those ligands that scored worse than 0.5, on a scale of 0 to 1, were retained as negatives for the purpose of parameter estimation. Table 5.1 lists the PDB codes, true ligands, and number of negative ligands for each protein in the training data set. The total number of negative ligands was 2,274 (2245), with 34 positive ligand examples.

5.3.4. Test Data Sets

Four sources were used to generate 29 test cases for screening utility (see Figure 5.2). The two data sets from the comparative paper of Bissantz et al.[67] were used, as in a previous report.[30] The original data sets included protein structures for HSV-1 thymidine kinase (1KIM) and estrogen receptor alpha (3ERT), ten known ligands of TK in arbitrary initial poses, and ten known ligands of ERα in arbitrary initial poses. The data sets were used with the modification (as above) that ligands were subjected to an automatic protonation procedure and energy minimization in order to eliminate differential bias between positive and negative ligands.

One limitation of the foregoing two cases is that the ligands are either drugs or are drug-like in their potency and physicochemical properties. Therefore, they are likely to form "easy" cases for docking tools. To address this issue, we took molecular structures from two papers which reported the results of combinations of both virtual screening and high-throughput screening to form two new cases where the true positives were reflective of the type of hits that can be found in library-based screening. The protein PARP

(poly(ADP-ribose) polymerase), PDB code 2PAX, along with 15 true ligands formed one case, based on Perkins et al.[86] The protein PTP1b (protein tyrosine phosphatase 1b), PDB code 1PTY, along with 11 true ligands formed the second, based on Doman et al.[87]

We used the PDBbind database[77] to generate a large number of additional cases for testing screening utility. From the full 800 complex set, we identified all proteins that were represented with at least 5 different ligands. For each of these proteins, we arbitrarily selected one of the PDB structures to serve as the screening target, and we generated a Sybyl mol2 format protein (prepared as above for the positive training data). The cognate ligands of the proteins were subjected to the same automatic protonation and minimization above. In cases where more than 20 ligands existed for a protein, we selected the 20 most diverse, based on molecular similarity, in an analogous procedure to the IcePick method.[88] The overall procedure yielded 25 proteins, with a total of 226 ligands. At the time this research was done, this represented the largest set of screening test cases currently available. As shown in Figure 5.2, the functional diversity of proteins and the structural diversity of ligands were large. The set included four serine proteases (row 3 of the figure), kinases, phosphatases, isomerases, aspartyl proteases, metalloproteases, and a number of other protein types. Importantly, the range of ligand binding affinities was large, with a substantial number of lower affinity ligands. Half of the ligands had pK_d less than 6.0 (micromolar or worse K_i or K_d), with just one fifth having pK_d greater than 9.0 (sub-nanomolar or better).

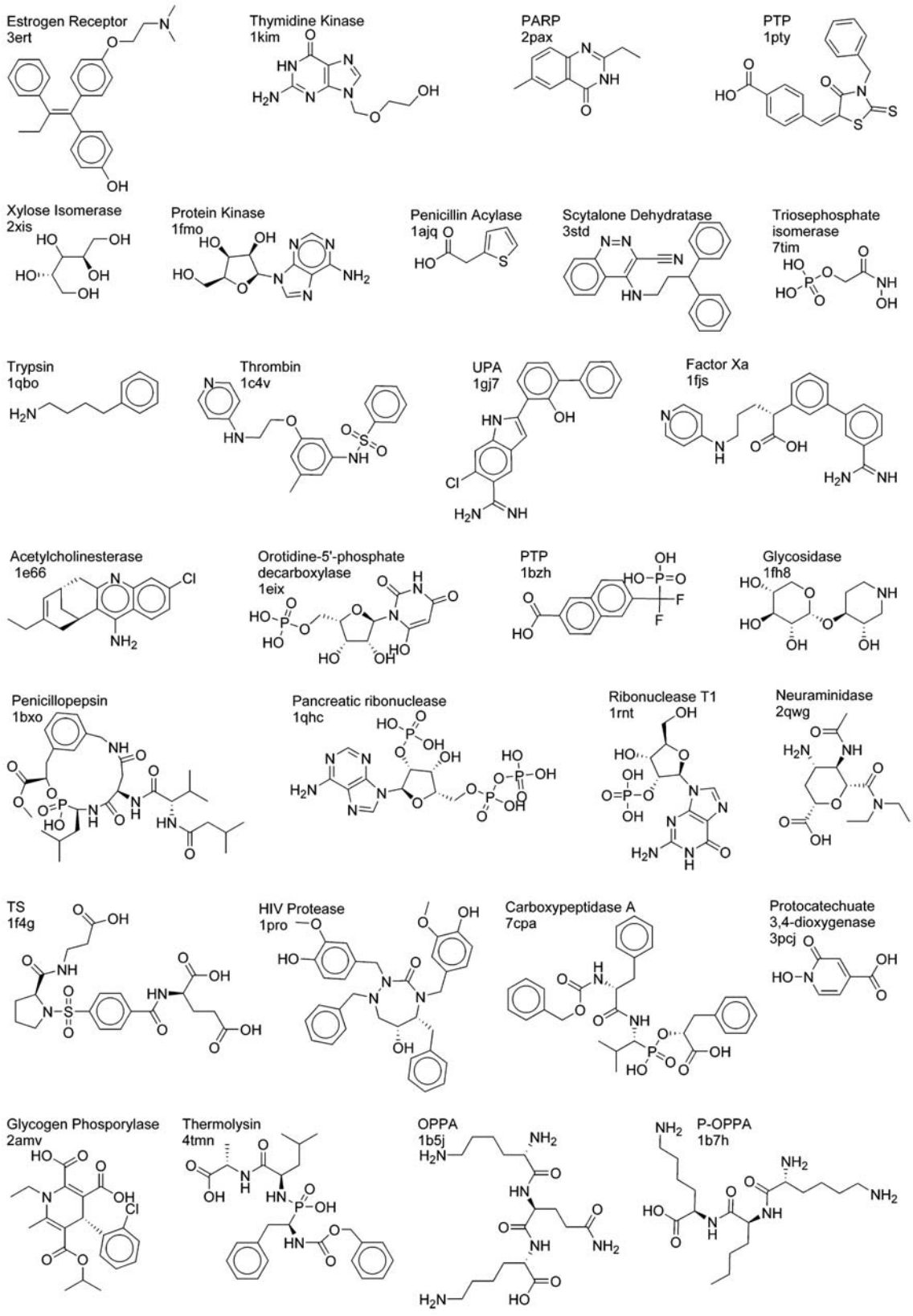


Figure 5.2. Example structures for the 29 screening enrichment test cases

5.3.5. Optimization Procedure

To demonstrate the feasibility of the approach of using negative data, we chose to optimize two parameters: the weight of the term for non-complementary (same charge) polar contacts and the weight of the term for protein-ligand and ligand-ligand clashes. The former term was parameterized exactly as in the original scoring function, and it will be referred to in what follows as `sf_pr` (Surflex polar repulsion). Owing to the relative success of our *ad hoc* approach to modeling interpenetration, employing the "pen" value, we chose to add a new term to the scoring function by including an analogous quadratic penalty. However, rather than scaling the term by the number of ligand atoms, which has no theoretical basis, we added the following new term to the original scoring function: `sf_hrd`*$(d_{ij}+\delta)(d_{ij}+\delta)$, with variables defined as above. This formulation can be thought of in terms of another additive energy term. The parameters `sf_hrd` (Surflex hard penetration) and `sf_pr`, when optimized, would be expected to be both significant and negative.

In searching for an optimum parameterization given some objective function, there is a complexity that is somewhat unique to docking and shared with 3D QSAR. As the function being optimized changes, the optimal poses for the ligands within the training set change as well. As in our previous work,[5, 51, 89, 90] we addressed this problem by interleaving parameter optimization with ligand pose optimization. In this approach, each time a ligand is optimized, the resulting pose is added to the *pose cache* for that ligand. In the inner loop of evaluation of ligand scores for computing the overall objective function, all cached ligand poses are evaluated, with the highest scoring one

defining the score for the ligand. For this work, it was sufficient to retain only the highest scoring pose on each iteration (essentially a pose cache size of 1).

Our objective function was a straightforward generalization of the common mean squared error function. For positive ligands, their contribution was the square of the difference between their maximal score under pose optimization and their experimentally determined score (in units of pK_d). For negative ligands, if their score was greater than 4.0, their contribution to the error function was the same as for positive ligands (squared difference), but if their score was 4.0 or less, their contribution was zero. So, any deviation from the correct score for a positive ligand induced a corrective pressure during optimization, but only in the case of inappropriately high scores would negative ligands contribute to the error function. The last complexity was that there were about two orders of magnitude more negative examples than positive examples. So, a simple optimization of the total error would have vastly overweighed the contribution of the negative ligands. We balanced the relative contributions of the negative ligands and positive ligands to be equal in order to avoid this.

Since there were only two parameters to optimize, we used a simple approach that combined broad sampling with bounded random search with fine-grained line-search and small random parameter perturbation. In principle, since the error function and the scoring function are continuous and differentiable, more complex approaches could have been employed, but they were not necessary. A single stable solution that minimized error was reached with `sf_pr` = -2.52 and `sf_hrd` = -0.945.

5.3.6. Computational Assessment

The old and new scoring functions of Surflex-Dock were evaluated for screening utility using our large set of 29 cases and for docking accuracy using our previous 81 complex data set.[30] We used Surflex-Dock v1.24 to generate protomols in all cases, using standard parameters. In computing scores for comparison between the old version (which returns values of both score and pen) and the new version (which returns only a score), we avoided the use of arbitrary thresholds by simply adding the score and pen values of the old scoring function to yield a single scalar combination score. This approximates the newer functional treatment and provides an apples-to-apples comparison.

In order to differentiate effects of the new scoring function from the treatment of ligand self-penetration and ligand pose optimization, we conducted two separate comparisons. To test the effects of the new scoring function, we compared performance of Surflex-Dock v1.31 with and without specifying the `-old_score` parameter, which selects the old scoring function (but does not change any other behavior). These effects are the primary focus of the research and are reported in detail in the Results and Discussion, below.

We conducted a separate comparison between the older version of Surflex-Dock (v1.24) and v1.31 using the old scoring function (`-old_score`) in order to assess the effects of the changes in the ligand-self penetration computation and ligand pose optimization. Figure 5.3 shows ROC curves for the ER and TK test cases using the old approach and new approach. While the ER case showed no real difference, the TK case shows better enrichment for the older Surflex-Dock version. The right-hand plot of Figure 5.3 illustrates the reason. While the scoring function is identical between the two

versions, the distribution of negative ligand scores using the more aggressive pose optimization procedure of the new version is shifted to the right of the old version. This reduces the separation between the positive and negative ligands (the positive ligand distribution does not change significantly). Despite poorer performance in this case, if we considered the ROC areas of the old and new versions in all 29 screening examples, we observed significantly improved performance using the new version ($p < 0.05$ by t-test). This is expected, given that frequently non-convergent pose optimization would simply add a degree of noise to ligand scores. In what follows, the only difference between versions is use of the `-old_score` switch within version 1.31 of the Surflex-Dock software.

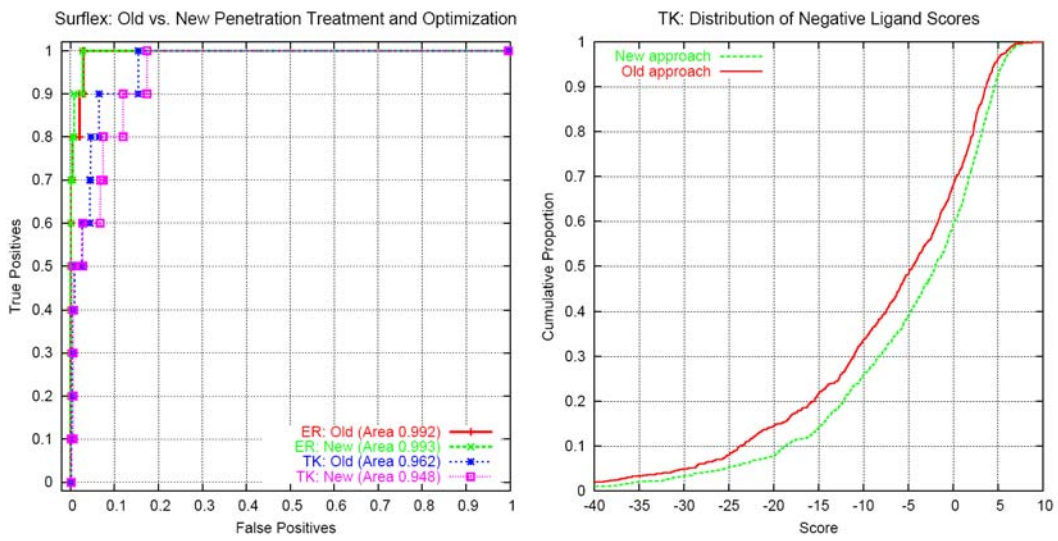


Figure 5.3. Screening performance under old and new treatment of penetration and pose optimization

Left: ROC screening performance for ER and TK using the old and new treatment of self-penetration. Right: cumulative distribution of negative ligand scores for TK.

5.4. Results & Discussion

We focused our attention on the results attributable to the differences between the old and new scoring functions, which lay in the effects of inappropriate atomic interpenetration and non-complementary polar contacts. Figure 5.4 shows the cumulative histograms of the scores for the negative ligands used in training the new scoring function.

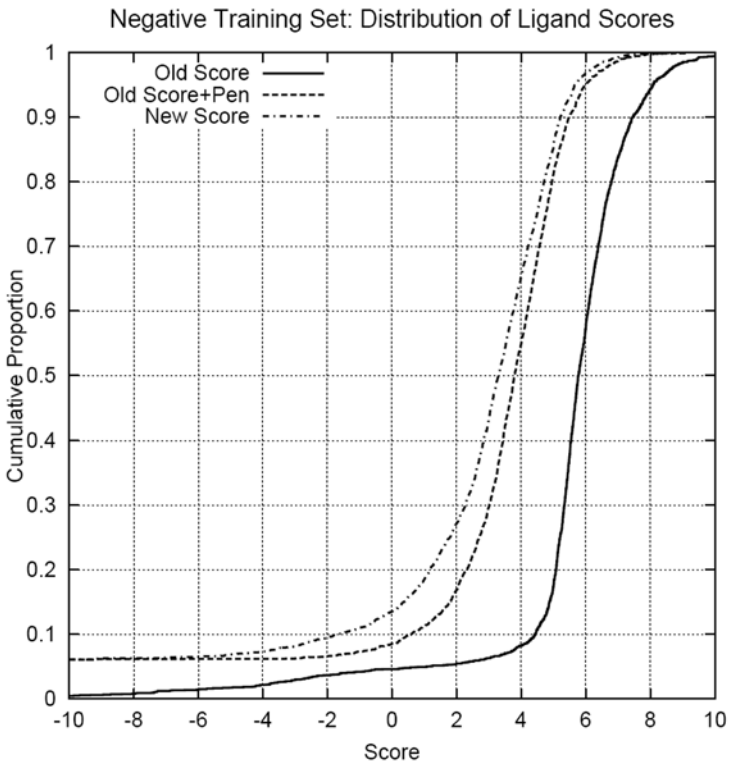


Figure 5.4. Cumulative histograms of negative ligand scores before and after parameter refinement.

The original scores (ignoring the penetration term) were the rightmost curve, with nearly all ligands scoring greater than 4.0 (not all ligands scored greater than 4.0 due to minor changes in protein preparation between negative data set generation and final evaluation). The scores corresponding to the new function are represented by the left-most curve. Note that following parameter optimization, approximately 70% of the negative ligands scored *less* than 4.0. It was not possible, using just the two parameters that were optimized, to simultaneously eliminate all 100% of the nominal false positives while retaining accurate scores for the positive examples. The middle curve is the cumulative histogram of the *sum* of the score and pen values for the old scoring function. While this curve was closer to that of the new function, the penalties that were learned

through systematic optimization in the presence of negative training data yielded uniformly lower scores.

Figure 5.5 (left) shows a plot of the old and new hydrophobic term, which reflects the negative contribution of the quadratic interpenetration penalty with its linear weight of –0.945 (sf_hrd). While this term is much more stringent than the sigmoidal component of the original function (with a maximal penalty of 0.08 log units), it is less stiff than a standard 6-12 potential. Figure 5.5 (right) shows the fit to the 34 positive ligand scores, with a mean error of 1.1 log units, which is quite comparable to the original report of the scoring function as parameterized solely on positive data. So, without significantly affecting the scores of known ligands, we were able to make an impact on the scores of the synthetically generated negative ligands.

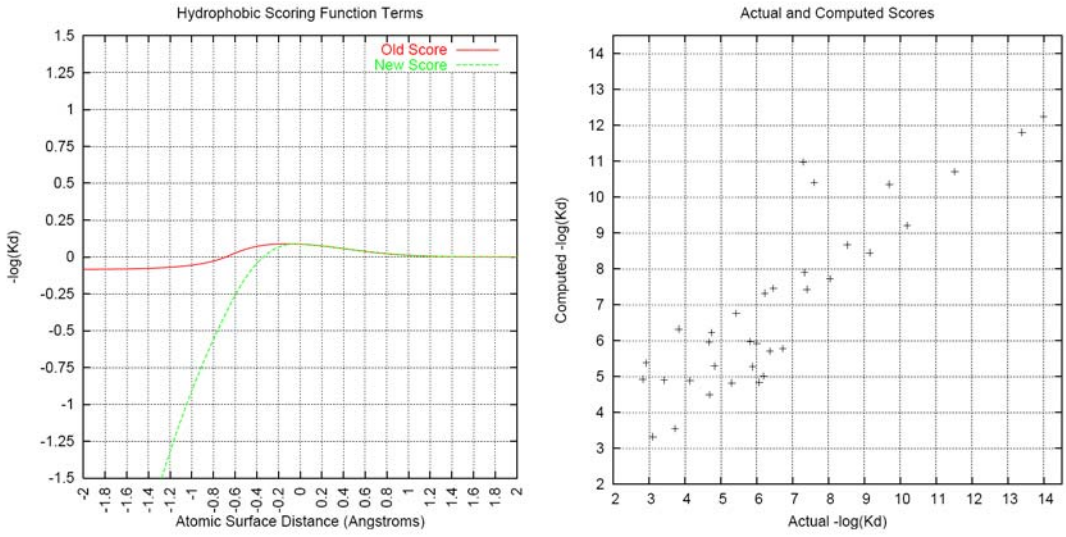


Figure 5.5. Left: New hydrophobic term. Right: New scoring function's K_d prediction performance.

5.4.1. Assessment of New Scoring Function in Screening Enrichment

The ER and TK cases, which have been the subject of numerous reports, [30, 37, 62, 67] deserve special attention. Figure 5.6 shows the full ROC curves and underlying

cumulative histograms of positive and negative ligand scores for the TK and ER test cases. Performance was excellent in both cases with both scoring functions, with all ROC areas exceeding 0.9. Maximal enrichment (ratio of true ligands to expected number of hits at all percentages of database screening) occurred at very low false positive rates for both scoring functions and exceeded 20-fold for TK and 500-fold for ER. Since ROC areas are much more stable to small changes in the scores of true and false ligands than maximal enrichment values, we will focus the quantitative comparisons between the scoring functions on ROC areas in what follows.

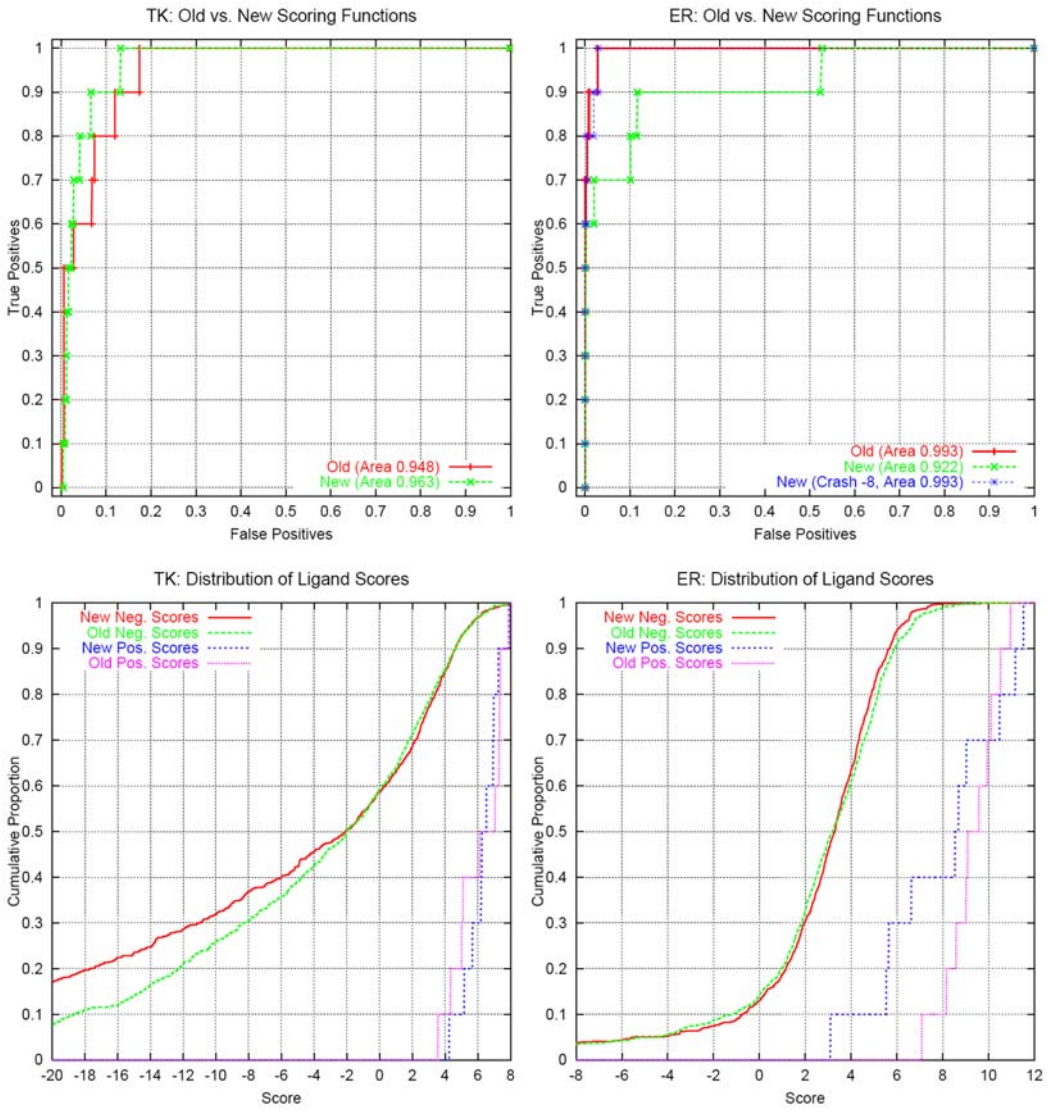


Figure 5.6. Screening performance and ligand score distribution for TK & ER

In the TK case, the new function leads to a reduction in the false positive rates at true positive rates of 70% and higher. In the ER case, the opposite is true. The bottom plots in Figure 5.6 show the cumulative histograms of positive and negative ligand scores for both cases. Surprisingly, while there were differences in the distribution of negative scores in both cases between the different scoring functions, the differences that drove the discrepancies in ROC curves were the result of changes in the *positive* ligand scores. In

the ER case, some of the large positive ligands were penalized by the new scoring function's harsher treatment of interpenetration. While this is not desirable, it is an expected effect in some cases. In cases like this, where very large ligands are desired, it is possible, as before, to treat the interpenetration portion of the score heuristically. By allowing all docked ligands a degree of penetration with no penalty, we observed performance equivalent to the old version (blue curve in Figure 5.6).

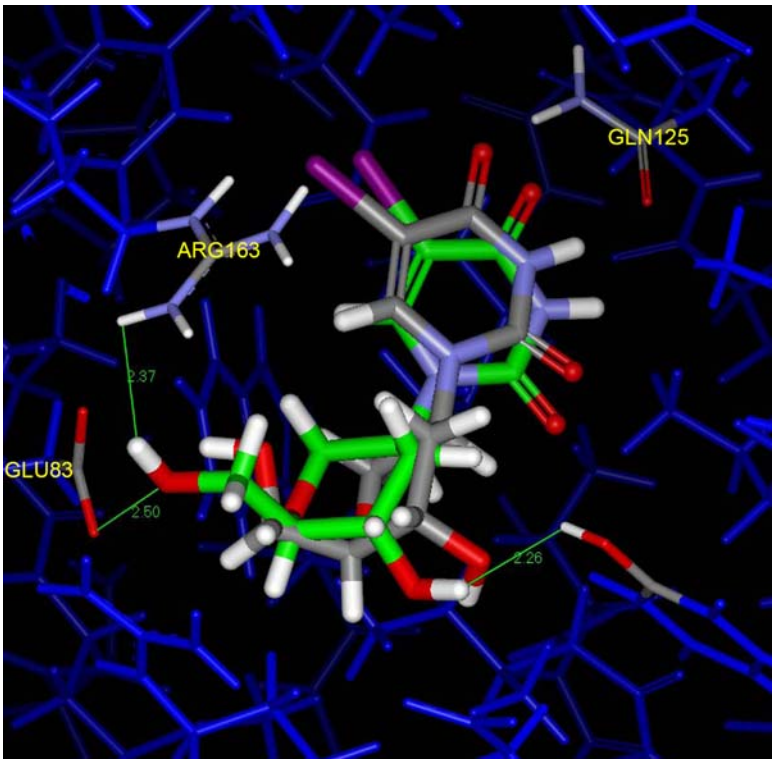


Figure 5.7. Docked pose of known TK ligand AHIU using the old and new scoring function

The old function pose is shown with green carbons; the new function pose is shown with gray carbons. Despite both poses being accurate by rmsd (< 1.5Å), the old pose scores 6 orders of magnitude less than the new pose due to inappropriate same-charge contacts.

In the TK case, we saw an unexpected effect. The lowest scoring of the true positives scored *higher* using the new scoring function than with the old. This is surprising because the new function has *harsher* penalties for inappropriate contacts.

However, since the scoring function of Surflex-Dock is used deep in the search process, we observed different solutions to the docking problem using the different functions. Figure 5.7 shows an example of this effect using the dockings observed for a single positive ligand of thymidine kinase. The solution using the new scoring function is depicted in atom color, and the solution using the old scoring function is depicted using green carbons. In the case of the old scoring function, very little weight was given to non-complementary polar contacts, and in the pose shown, there were three very close contacts between pairs of atoms with charges of the same sign. With the new scoring function, this pose scores more than 6.0 log units worse than with the old scoring function, owing to the large negative weight given to the sf_pr term. The pose returned by the new scoring function avoids all of the improper contacts while retaining as many appropriate contacts. While both poses were very close to the experimentally determined pose (< 1.5Å rmsd), the pose returned by employing the new scoring function was clearly superior.

Table 5.2. TK screening performance: true positive rates for several algorithms

FP%	DOCK	FlexX	Fred	Glide	GOLD	QXP	Slide	Surflex	Surflex-New
2.5	0	20	0	20	10	0	0	40	60
5	10	40	0	50	40	20	0	80	80

Since these two cases have been used in a number of other studies, it is possible to make direct comparisons among different methods. Table 5.2 shows the true positive rates reported by Kellenberger et al.[62] for thymidine kinase for eight docking methods, amended to include the Surflex result with the new scoring function. As evidenced by the plots in Figure 5.6, the Surflex results did not change much, with a slight improvement at the 2.5% level of false positives. Note, however, that the new results did *not* require the

choice of a threshold penetration value, which was required in the previous studies. Table 5.3 shows enrichment factors reported by Halgren et al.[37] amended to include the Surflex result with the new scoring function, again without any special treatment of protein interpenetration. In both the TK and ER cases, the Surflex enrichment factors were substantially better than the other methods.

Table 5.3. Screening enrichment factors for several algorithms

Complex	DOCK	FlexX	Glide	GOLD	Surflex-New
TK	3	11.1	19.3	8.2	37.9
ER	6.7	8.9	47.1	28.5	90.7

While these results are encouraging, they represent a limited test, given just two proteins and 20 positive ligands, all of which are either drugs or drug-like in potency and physicochemical properties. Figure 5.8 shows ROC curves and score histograms for PARP and PTP. In these cases, the positive ligands were discovered through combinations of virtual and high-throughput screening. They were all of relatively poor potency and reflect the makeup of common screening libraries. In both cases, the new scoring function improved performance, though it did so based on different effects. In the case of PARP, the slight leftward shift in the distribution of negative ligand scores (lower left plot, upper part of red curve) was responsible for the difference observed in the ROC curves. In the case of PTP, as with TK above, the effect was a substantial right shift of positive ligand scores. Again, it appears that the new scoring function guided the docking search algorithm more effectively to better solutions.

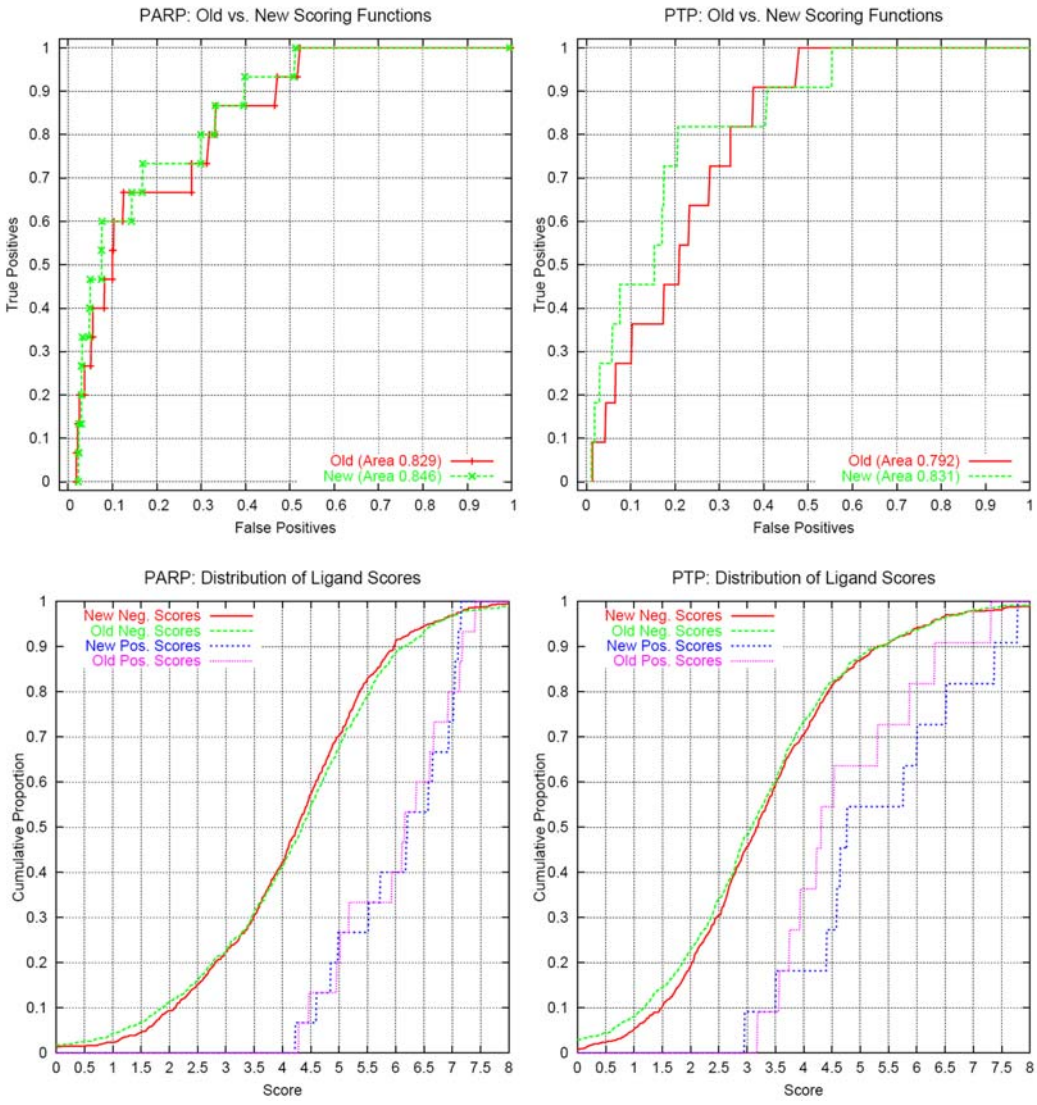


Figure 5.8. Screening performance and ligand score distribution for PARP and PTP

Table 5.4 summarizes Surflex-Dock screening performance on all 29 cases tested (ligand examples shown in Figure 5.2) for the old and new scoring functions, using ROC areas to characterize the separation of positive and negative ligand sets. Differences of less than 0.005 are considered negligible. The 29 cases included a diverse set of 226 ligands, with a large number having poor binding affinities (half with micromolar or worse K_d or K_i). Overall, the new scoring function performed as well as or better than the old scoring function in 21/29 cases, so it is clearly not worse than the old function (p =

0.01 by exact binomial). The converse is not true. That is, the old scoring function performs as well as or better than the new one in 15/29 cases, which allows the possibility that the old scoring function is worse. However, the number of test cases is too small to make a strong statement that the new approach is significantly better in terms of the proportion of cases where ROC area is clearly improved. With the new scoring function, maximal enrichment of true ligands over non-ligands exceeded 20-fold in over 80% of cases, with enrichment of greater than 100-fold in over 50% of cases. Given that many of these cases were clearly much more difficult, based on ligand affinities, than the widely used TK and ER cases, performance on par with those two examples in the *majority* of cases suggests that Surflex-Dock should yield strong performance in terms of screening utility in a wide variety of cases.

Table 5.4. Screening performance of new and old scoring functions for 29 cases

Name	Nmols	New Score	Old Score	Difference
1F4G	10	0.693	0.594	0.098
1FMO	8	0.764	0.722	0.041
PTP	11	0.831	0.792	0.039
2XIS	5	0.958	0.923	0.035
7TIM	6	0.966	0.935	0.031
3STD	5	0.844	0.814	0.030
1AJQ	6	0.922	0.897	0.025
4TMN	13	0.828	0.810	0.018
PARP	15	0.846	0.829	0.017
2AMV	5	0.709	0.693	0.017
TK	10	0.963	0.948	0.016
1QBO	20	0.990	0.978	0.011
1FJS	6	0.980	0.974	0.007
1GJ7	12	0.953	0.948	0.005
1EIX	5	0.996	0.995	0.002
1BZH	12	0.917	0.916	0.002
1FH8	6	0.997	0.995	0.002
1B5J	16	1.000	1.000	0.000
1B7H	6	0.999	1.000	-0.001
1E66	6	0.764	0.767	-0.003
3PCJ	8	0.948	0.952	-0.003
1RNT	5	0.952	0.966	-0.015
7CPA	8	0.901	0.916	-0.015
2QWG	7	0.965	0.987	-0.022
1C4V	20	0.876	0.900	-0.023
ER	10	0.922	0.993	-0.071
1PRO	20	0.862	0.955	-0.093
1QHC	6	0.791	0.886	-0.095
1BXO	5	0.746	0.985	-0.239

We further tested the performance of the system using an entirely new negative screening set of ligands, derived from the ZINC database. This was done since it was theoretically possible that the scoring function optimization procedure could have "learned" something specific about properties of the negative set derived from the Rognan benchmarks, which was used to produce the putative negative examples for the training set. However, the ROC areas derived using the ZINC negative set with the new scoring function were statistically indistinguishable from those presented above. In fact,

the ROC area differences between the scoring functions using the ZINC versus Rognan negative sets were almost perfectly correlated, with a Pearson r^2 of 0.987.

5.4.2. Effects of Protein Conformation

Table 5.5. Effect of protein conformation on screening performance

Protein		Original Structure		New Structure		Combination
Name	N	PDB	ROC Area	PDB	ROC Area	ROC Area
Penicillopepsin	5	1bxo	0.746	1apw	0.941	0.946
Pancreatic ribonuclease	6	1qhc	0.791	1afk	0.915	0.957
Acetylcholinesterase	6	1e66	0.764	1gpn	0.914	0.847
Thymidylate synthase	10	1f4g	0.693	1tsl	0.7	0.707
cAMP dep protein kinase	8	1fmo	0.764	1stc	0.665	0.73
Glycogen phosphorylase	5	2amv	0.709	3amv	0.59	0.684

In the six cases with poorest performance of the new scoring function, the old scoring function performed better only in two (1bxo and 1qhc), reflecting the possibility that these six cases may be difficult proteins in some intrinsic sense. Another possibility is that the particular conformations of the protein structures used for screening were not propitious due to structural uncertainties or induced fit. We arbitrarily selected different protein structures for each of these six cases and re-tested the performance of Surflex-Dock's new scoring function. Table 5.5 shows the results from the original structures, the new structures, and from combining both screens by taking the maximal ligand score from both structures in each case. In 3/6 cases, the new structures yielded much improved performance, suggesting that these cases may have been outliers. These included the two cases in which the old scoring function had outperformed the new one. In the remaining cases, there was no significant change for one (1f4g) but reduced performance for two (1fmo and 2amv). Clearly, protein conformation can have unpredictable effects. However, it appears that the simple approach of using multiple structures (as described in the Receptor Flexibility Section 2.2.4) and reporting the maximum score of each ligand

might be an appropriate safe strategy. In 6/6 cases, this approach performed better than the worst of the single-protein runs ($p = 0.02$ by exact binomial), and in 3/6 cases performed better than either single structure alone. These results stand in some contrast to the interesting, but counterintuitive, result reported by Wei et al.[91] where they observed *worse* performance using this approach unless they corrected for cavitation energies in different protein structures.

5.4.3. Solvation Effects

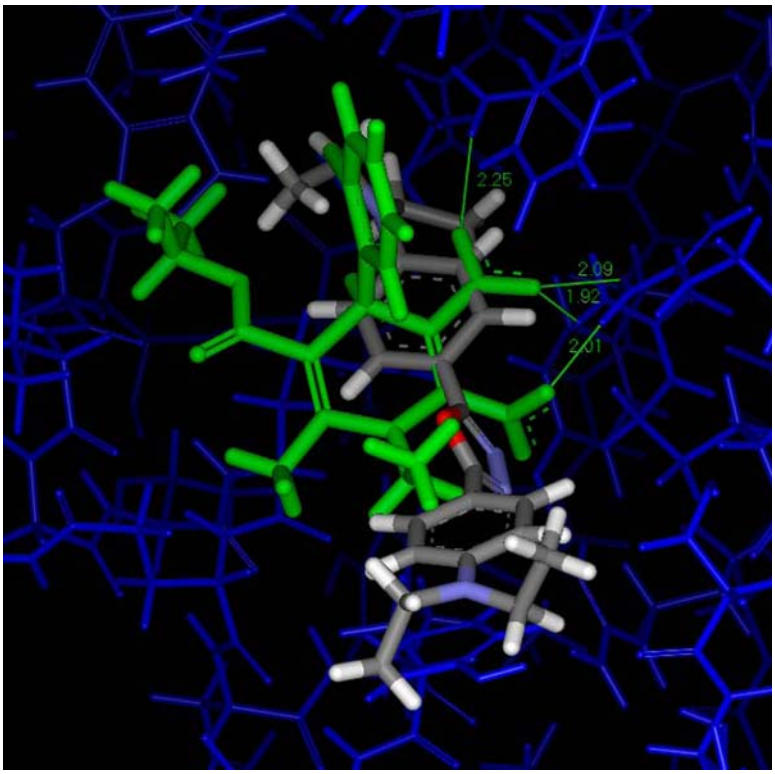


Figure 5.9. Native ligand and docked false positive poses for 2AMV

The native ligand pose is shown in atom color; the docked false positive in green. Distances to several polar moieties on the protein are also shown in green.

The protein for which Surflex-Dock yielded the poorest performance was glycogen phosphorylase (2amv), and performance was not improved using multiple

conformations. The active site of this protein is quite hydrophilic, with 20 protein atoms capable of making polar interactions with a ligand in the binding pocket. Figure 5.9 shows a positive ligand (green) and a non-ligand (atom color) in the active site of the protein. The known ligand makes a network of polar interactions with three guanidine moieties. However, the non-ligand makes no successful polar interactions in the binding site at all, while still scoring 5.0 pKd. Further, the ligand effectively buries multiple polar atoms of the protein, rendering them inaccessible even to solvent. The scoring function, even in its new form, does not account for the cost of desolvating the protein (important in this case) or the ligand.

We approximated this intention in this case by requiring that the ligands of glycogen phosphorylase received a polar score of at least 3.0 in either of the two structures. If so, we recorded the maximum score of the ligand, else we recorded a zero. Employing this simple heuristic, we saw an improvement from 0.684 to 0.862 in ROC area. While this is an *ad hoc* procedure, it motivates the development of an effective strategy for modeling desolvation costs. Such a strategy should include some computation of the degree of buriedness of each polar atom (i.e. the degree to which it is inaccessible to solvent in the docked state), the solvated polar score of the protein and ligand, and the degree to which complementary polar contacts between the protein and ligand ameliorate loss of interactions with solvent molecules. A sufficiently refined treatment will require several parameters and will benefit from additional positive training data in addition to the negative data that has been the subject of this work.

5.4.4. Docking Accuracy and Speed

We evaluated docking accuracy on the same data set used previously, consisting of 81 protein-ligand complexes.[30] Docking accuracy was not significantly different between the old and new scoring functions, with 58/81 complexes (72%) in both cases having rmsd of top-ranked poses within 2.0A of crystallographic observation using either scoring function.

Docking speed was not significantly affected by any of the changes from the previous reported version to the modified algorithm reported here. On standard workstation hardware (Intel Xeon 2.80 GHz, 1GB RAM, Windows XP Professional, Surflex-Dock version 1.31 with default options), the mean docking time over the 81 complexes was 17 seconds, with ligand flexibility ranging from 0 to 15 rotatable bonds. Docking time was roughly linear in the number of rotatable bonds, with a mean of 3.0 seconds (stddev. 1.54) required for a single docking per rotatable bond. Note that this is approximately 10-fold faster than the report from Kellenberger et al., which relied on older SGI hardware, and for which much less efficient compiler optimization strategies were employed.

5.5. Conclusion

Our results clearly demonstrate that synthetically generated negative data can be used effectively in estimating parameters for scoring functions in molecular docking. Over a large variety of test cases, both with respect to screening utility and docking accuracy, the newly parameterized scoring function performed at least as well as the old scoring function, which relied on a less systematic, hand-tuned, approach for addressing repulsive interactions.

Apart from pure performance issues, the new approach is clearly an improvement methodologically, in three respects. First, the new scoring function reformulates the formerly *ad hoc* penetration term (which included a dependence on ligand size) into one with a theoretically more satisfying form that can be thought of as another additive energetic effect. Second, in an operational sense, the new function returns a *single* value, which has direct interpretation in ranking ligands. While heuristic methods may be layered on top of a straightforward score-based ranking, none are required. Third, both the penetration term and the term related to non-complementary polar contacts received significant weight in the re-tuned function, which comports with both intuition and theory.

By incorporating negative training data, we have been able to address two of the key challenges we set out in the original Surflex report:[30] consolidation of scoring and penetration terms and inclusion of negative training data. There is still much room for improvement. Based on the preliminary results here regarding treatment of desolvation effects, development of a term that treats both protein and ligand desolvation symmetrically, while taking into account issues of solvent exposure, is a high priority. This will benefit from a larger training set of positive examples, which could be greatly increased by leveraging efforts such as PDBbind.[77]

The benchmark data set established in this work is publicly available and offers a large number of diverse cases for testing screening performance of docking methods. Surflex-Dock v1.31, incorporating the new scoring function, performed extremely well on 13/29 cases, with ROC areas of 0.95 or greater, very well on 10 additional cases (ROC area > 0.80), and showed weaker performance on the remaining 6 cases. In those cases, a

simple approach that made use of two protein conformations was remarkably successful in improving performance. It is our hope that other methodological researchers in the field of molecular docking will make use of (and add to) this benchmark data set.

This chapter demonstrated the feasibility of applying robust parameter refinement to an empirical scoring function. This was proof-of-concept that destabilizing forces that we know affect protein-ligand binding could be learned via creative consumption of data. The next chapter generalizes this idea, describing a framework for leveraging a user's knowledge of their input data to construct custom scoring functions.

Chapter 6
Customizing Scoring Functions for Molecular Docking

6.1. Abstract

Empirical scoring functions used in protein-ligand docking calculations are typically trained on a dataset of complexes with known affinities with the aim of generalizing across different docking applications. We report a novel method of scoring-function optimization that supports the use of additional information to constrain scoring function parameters, which can be used to focus a scoring function's training towards a particular application, such as screening enrichment. The approach combines multiple instance learning, positive data in the form of ligands of protein binding sites of known and unknown affinity and binding geometry, and negative (decoy) data of ligands thought *not* to bind particular protein binding sites or known *not* to bind in particular geometries. Performance of the method for the Surflex-Dock scoring function is shown in cross-validation studies and in 8 blind test cases. Tuned functions optimized with a sufficient amount of data exhibited either improved or undiminished screening performance relative to the original function across all eight complexes. Analysis of the changes to the scoring

function suggests that modifications can be learned that are related to protein-specific features such as active-site mobility. Note: the work presented in this chapter was submitted to the Journal of Computer-Aided Molecular Design and was currently in press at the time of writing this dissertation.[92]

6.2. Introduction

The previous chapter introduced the idea of using *negative* training data to provide a sensible basis for optimizing the repulsive parameters of an empirical scoring function[83]. These data took the form of computationally generated putative decoy ligands, which were produced by docking a decoy library to each of the protein structures from the complexes used in the original parameter estimation.[51] By making use of such data, it was possible to estimate the value of repulsive terms such as protein-ligand interpenetration instead of relying on an *ad hoc* value. The difficulty with an approach relying only upon positive data (protein-ligand complexes of known affinity) is that the inductive bias of the most parsimonious estimation regime is to assume that if an example of an interaction does not exist (e.g. a ligand atom penetrating a protein atom) then nothing can be concluded, which leads to a value of zero on the associated term. This is in contrast with PMF-type approaches, where the normalization procedures lead to an inductive bias wherein the absence of an observation is indicative of low probability, which results in a preference against unobserved interactions.[53, 54]

When docking methods are evaluated, there are three criteria applied. First, *docking accuracy* measures the probability that a ligand will be docked in a pose that matches the experimental determination. Second, *screening utility* measures the ability of a docker to rank a list of known ligands of a protein above a set of decoys. Third, *scoring*

accuracy measures the ability to rank a list of active ligands in order of binding affinity. In most work with scoring function development, the actual data for parameter estimation relates to scoring accuracy.[49, 51, 56, 93] Parameters are sought to minimize the difference between computed and experimental affinities for ligands with known bound geometries. In our recent report, we showed that it was possible to make use of *negative* data which related to screening utility. The approach sought parameters for a scoring function that would simultaneously minimize computed/experimental affinity differences *and* minimize the excursion of computed decoy affinities beyond a fixed threshold.[83]

In this chapter, we generalize this concept so that information from each of the three areas of docking application may be used to influence the refinement of Surflex's scoring function. Data of the following form may be used to refine the scoring function:

Protein/ligand complexes of known affinity (as before). The constraint is that the computed score should be as close as possible to the experimental one for the *highest scoring pose* that is close to the experimentally determined one.

Ligands known *not* to bind a protein beyond some threshold. The constraint is that the computed score for *any pose* (expressed as pK_d) *should not* exceed a settable threshold.

Ligands known to bind a protein, but without a precise determination of affinity. The constraint is that the computed score for the *best pose* (expressed as pK_d) *should* exceed a settable threshold.

A set of ligands known to bind a protein along with a set of ligands thought not to bind. The constraint is that the separation of the *best poses* of actives and decoys be maximized.

The correct pose of a ligand for a protein along with incorrect poses of the same ligand. Here the constraint is that the score for the *best close-to-correct pose* must exceed all scores for clearly incorrect poses.

The first three types of data bear on scoring, the fourth bears on screening utility, and the last bears on geometric docking accuracy. The optimization procedure implements a weighted objective function for parameter optimization based on simultaneous consideration of all types of data. In such an optimization problem, the issue of *which* pose of a ligand to consider becomes important. As with our previous work,[5, 51, 89] we explicitly address this problem by making explicit choices of pose as the scoring function evolves. For example, given a protein/ligand complex with known affinity, it is appropriate to make use of the experimental ligand pose as the initial pose in parameterizing the scoring function. However, while the experimental pose may be a very good static approximation of the true interaction between the ligand and protein, small variations in the ligand position (within the accuracy of the crystallographic experiment) may yield different scores. Consider a computed pK_d of 7.0 at the precise crystallographic pose of a ligand whose known pK_d is 8.0. If a very close pose yields a maximum for the function of 8.0, one should use the 8.0 score, which entails no error for the scoring function. This issue is discussed in detail in our earliest work on scoring functions for docking,[51] which was based on earlier work in 3D QSAR.[90] The approach has been formalized within the machine-learning community as multiple-instance learning,[4] and it has a substantial impact on the performance of systems where hidden variables (here the precise pose of a ligand) are present.

In this chapter, we demonstrate that this generalized multiple-constraint optimization procedure is able to improve the screening performance of Surflex-Dock in a protein-target specific manner. Given that operational use of docking programs typically involves a user with large amounts of non-public data relating to the particular target under study, we expect that the ability to specifically tune docking parameters based on such data will lead to substantial practical benefits in all three areas of docking performance.

The optimization procedure has been implemented as a standalone Surflex program (Surflex-Dock-Optimize, version 1.0). The scoring function parameter files can be used by the released version of Surflex-Dock which has been updated to allow for loading parameter files (version 2.11-lp). A future release of the Surflex-Dock software will incorporate the optimization feature directly. The software that implements the algorithms described here is available free of charge to academic researchers for non-commercial use (contact the corresponding author for details on obtaining the software). Molecular data sets presented herein are also available.

6.3. Methods

The optimization procedure described herein is general enough for use with any parameterized scoring function. For the purposes of this work, results are reported for the scoring function used in Surflex-Dock. A relatively brief review of this scoring function and its parameters will be given as other work offers a more detailed account.[51, 69] This will be followed by a description of the data used for training and testing optimized scoring functions. The last section will describe the optimization procedure itself. All training and testing data sets used in this study have been taken from published docking

benchmarks that are freely available at (http://www.jainlab.org). Detailed usage of the optimization software can be found in Appendix C.1.1. The code base is open source and documented in Appendix C.1.2.

6.3.1. Scoring Function

The scoring function employed by Surflex-Dock was originally trained on 34 protein-ligand complexes representing a variety of functional classes whose dissociation constants ranged from 10^{-3} to 10^{-14}. This function was optimized to predict the experimental binding affinities of each complex, resulting in an effective means for modeling the non-covalent interactions between small organic molecules and proteins. The function is continuous and piece-wise differentiable with respect to pose. Listed in order of import, the terms of the scoring function are hydrophobic complementarity, polar complementarity, and entropy. Parameters are listed in Table 6.1. The following four equations define the scoring function:

Eq. 6.1

$$steric\_score = l_1 \exp^{\frac{-(r+n_1)^2}{n_2}} + \frac{l_2}{1+\exp^{n_3(r+n_4)}} + l_3 \max(0, r+n_5)^2$$

Eq. 6.2

$$polar\_score = \left[l_4 \exp^{\frac{-(r+n_6)^2}{n_7}} + \frac{l_5}{1+\exp^{n_3(r+n_8)}} + l_3 \max(0, r+n_9)^2 \right] \left[\frac{1}{1+\exp^{n_3(-(b_{ij}\bullet v_i)(b_{ij}\bullet v_j)-n_{10})}} \right] \left[(1+n_{11}c_i)(1+n_{11}c_j) \right]$$

Eq. 6.3

$$polar\_repulsion\_score = \left[l_6 \exp^{\frac{-(r+n_{12})^2}{n_{13}}} \right] \left[\frac{1}{1+\exp^{n_3(-(b_{ij}\bullet v_i)(b_{ij}\bullet v_j)-n_{10})}} \right]$$

Eq. 6.4

$$entropy\_score = (l_7 \cdot n\_rot) + (l_8 \log(molweight))$$

Table 6.1. Surflex scoring function parameters

Equation Variable	Parameter Name	Explanation
l1	stz	Steric Gaussian attraction scale factor
l2	str	Steric sigmoid repulsion scale factor
l3	hrd	Steric hard penetration scale factor
l4	poz	Polar Gaussian attraction scale factor
l5	por	Polar sigmoid repulsion scale factor
l6	pr2	Polar mismatch scale factor
l7	ent	N rotatable bonds scale factor
l8	con	Molecular weight scale factor
n1	stm	Steric Gaussian location
n2	sts	Steric Gaussian spread
n3*	STT	Sigmoid steepness
n4	srm+stm	Steric sigmoid inflection point
n5*	bump_thresh	VdW allowance for hard clashing
n6	pom	Polar Gaussian location
n7	pos	Polar Gaussian spread
n8	srm+pom	Polar sigmoid inflection point
n9*	pbump_thresh	VdW allowance for hard clashing (polar)
n10	hpl	Polar direction sigmoid inflection point
n11	csf	Charge scale factor
n12	prm	Polar repulsion Gaussian location
n13	ms	Polar repulsion Gaussian spread

* Variable was considered a constant and not optimized

The hydrophobic and polar terms (Eq. 6.1 and Eq. 6.2) dominate the scoring function. These terms operate on the pair-wise van der Waals surface distance r between atoms, coupled with information such as element type, formal charge, and atom status as a hydrogen bond donor or acceptor. The distance dependence of the hydrophobic and polar interactions is composed of a Gaussian, sigmoid, and quadratic penetration term. The polar term is further scaled by directionality and formal charge. The directionality term between atoms I and J is computed based on three vectors (normalized to unit length): the vector from between I and J (b_{ij} in Eq. 6.2), the preferred direction of interaction of I (v_i), and the preferred direction of interaction of J (v_j). If multiple directional preferences are present (as for a carbonyl moiety), the preference that yields

the maximal polar interaction is used. Additional details can be found in the original paper describing the scoring function.[51]

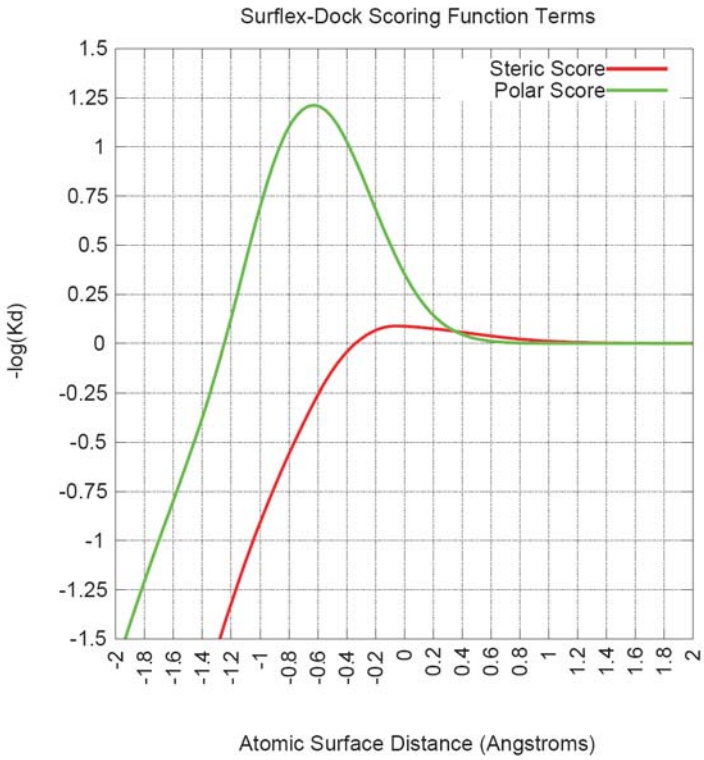


Figure 6.1. Hydrophobic and polar terms of the default scoring function

Figure 6.1 plots the relative hydrophobic and polar scores for an ideal contact. Due to the large number of hydrophobic contacts typically seen between a protein and a ligand, on average the hydrophobic term tends to dictate scoring despite a smaller peak value per ideal contact. An ideal hydrogen bond for the scoring function exists, for example, when the center of the O in C=O is 1.97Å away from the center of the H in an N-H and the four atoms are co-linear. This results in a contribution of 1.25 pK_d units to the interaction score.

The polar repulsion term (Eq. 6.3) measures the penalty for placing atoms of similar polarity in close proximity and is scaled by direction. The remaining entropic term (Eq. 6.4) captures the degrees of rotational and translational freedom lost to the ligand upon binding. This ligand-centric penalty scales linearly in the number of rotatable bonds and linearly with the log of its molecular weight.

As described in Chapter 5, our previous work refined the original scoring function, determining weights for penalty terms that govern steric interpenetration and non-complementary polar contacts.[83] This new function (Surflex-Dock v1.31 and all succeeding versions) was shown to be an improvement over the original and is the default scoring function used by the program. The set of parameters that define the default function are the starting point for further optimization in this work.

The protocol used for generating training data, performing scoring function optimization, and testing protein-specific scoring function optimization was implemented as a standalone Surflex module (Surflex-Dock-Optimize v1.0). Test set validation performance in this work was calculated using the standalone optimization suite. Scoring function parameter files generated by this process can be loaded into the latest released Surflex-Dock program[70] (v2.11-lp, which has an added `-lparam` command-line switch). The results presented here are statistically indistinguishable from those generated by employing the derived parameters from optimization to dock the test ligands using Surflex-Dock v2.11-lp with the `-lparam` option.

6.3.2. Training Data Set

This work employed several publicly available molecular datasets for training the scoring function. The original 34 complex set used to train the original scoring function

was used to provide an "anchor" for the scoring function during further optimization.[51] Each of the 34 complexes was annotated by an experimentally derived K_d. This set served as a control during optimization to ensure that the parameters of the tuned function do not stray exceedingly far from the values of the default function.

Screening enrichment data was gathered from two primary sources: the PDBBind database[77] and the DUD database[94]. The former was the basis for our previous work[83] that is used again here (called the Pham Benchmark). The Pham benchmark consisted of 27 protein structures with 256 known active ligands from the PDBBind database along with two decoy databases. Those targets from the Pham benchmark that overlapped with the DUD database[94] were chosen as case studies for training our scoring function. Table 6.2 lists the targets along with information about the number and composition of active ligands: acetylcholinesterase (AChE), estrogen receptor (ER), coagulation factor Xa (FXa), HIV-1 protease (HIVPR), poly(ADP-ribose) polymerase (PARP), thrombin, trypsin, and thymidine kinase (TK). The protein structures for these eight targets served as the docking targets for both the training and test sets. The 107 cognate ligands of these eight targets from the Pham set became the training data from which the scoring function would learn to distinguish active from non-active. Active ligands may be referred to as positive examples in what follows.

Table 6.2. Proteins and known actives selected as training data

Target	PDB code	Nmols Train	MW	Nrot	N Polar Negative (formally charged)	N Polar Positive (formally charged)
AChE	1e66	6	334.7	4.8	1.3 (0.0)	4.2 (2.7)
ER	3ert	10	465.6	9.9	3.3 (0.0)	2.6 (0.8)
FXa	1fjs	6	450.7	8.3	4.8 (0.7)	4.7 (3.2)
HIVPR	1pro	20	559.7	14.1	5.7 (0.3)	4.8 (1.0)
PARP	2pax	15	202.1	0.7	2.4 (0.0)	1.3 (0.0)
Thrombin	1c4v	20	429.2	8.8	3.6 (0.3)	5.2 (3.6)
TK	1kim	10	264.5	4.9	4.7 (0.0)	3.8 (0.0)
Trypsin	1qbo	20	386.8	6.5	3.5 (0.6)	5.6 (4.6)

Two decoy libraries taken from the Pham benchmark were used as the decoy training background in optimizing for screening enrichment. Two sources were used to test the potential of training bias towards a particular set of decoys. One library was derived from the work of Bissantz et al.[67] which contained 990 randomly selected nonreactive organic molecules with 0 to 41 rotatable bonds from the Available Chemicals Directory (ACD). This benchmark (hereon referred to as the Rognan set) was culled to a more drug-like set of 861 molecules with a maximum of 15 rotatable bonds.[30] The ZINC database (version 07.26.2004)[95] was the source for the second decoy set. This database was compiled from the catalogs of numerous small molecule vendors and represents a collection of purchasable compounds suitable for virtual screening. A random subselection of 1000 molecules was taken from the drug-like subset (1,847,466 total) to generate the ZINC1 decoy benchmark.[83] We will refer to compounds from a background library interchangeably as either decoys or negative ligands. Recent work by Irwin and Shoichet[94] considered multiple decoys sets and compared their physical properties as well as the degree of challenge they posed in screening. The DUD set itself, which was designed with knowledge of the specific active ligands for the targets under construction, was the most challenging in their experiments. Among the "agnostic" decoy sets

(constructed with no specific knowledge of the targets under consideration), the ZINC1 set was the most challenging, and the Rognan set was the least. Consequently, in what follows, we focus most of our attention on the ZINC1 results.

Data was uniformly prepared by an automated procedure. All protein structures were converted from PDB to Sybyl mol2 format and protonated at physiological pH. Active site rotamers such as hydroxyls and thiols, as well as imidazole tautomers, were sampled and selected for interaction with the co-crystallized cognate ligand. Ligands were minimized using a DREIDING-like force-field as implemented within Surflex.[26, 70] The active site models (called protomols) necessary for docking with Surflex were generated using the crystallized ligand (`surflex-dock –proto_bloat 1.0 proto xtal-lig.mol2 protein.mol2 p1`). Initial ligand poses used as input to the scoring function refinement algorithm were generated using Surflex-Dock-Optimize, which yields equivalent poses to Surflex-Dock version 2.11 with default screening parameters (`surflex-dock –pscreen dock_list mol-archive.mol2 p1-protomol.mol2 protein.mol2 log`).

6.3.3. Test Data Set

To cleanly assess the performance of our tuned function, we conducted screening enrichment experiments on positive and decoy ligands that were never encountered in training. As described above, each of the 8 test targets were shared between the Pham and DUD benchmarks. We made use of the Pham actives for training, which contained fewer examples of known ligands than present in the DUD set. We used the DUD actives that did not include any from the Pham set as ligands to test scoring functions that had been optimized with knowledge of the Pham actives. A fair test required a new decoy

background. Another 1,000 unique molecules were randomly selected from the drug-like subset of ZINC, this time from version 2007. The process of generating the new decoy set made use of 2D molecular similarity to eliminate the overlap between the testing and training decoy libraries. The test decoy set will be referred to as ZINC2.

6.3.4. Optimization Procedure

This work introduces a constraint based optimization scheme that allows the use of several different sources of data in customizing a scoring function. We will begin by defining the available constraints and how they might be utilized to create scoring functions optimized for a particular task. We will then cover the optimization protocol in detail, along with the options that govern its use.

During any parameter optimization regime, the goal is to extremize the value of an objective function as we explore the parameter space. Our objective function is described by user-defined constraints on training data. Constraints come in three flavors: scoring, screening, and geometric. Together these constraints combine to form the objective function.

Table 6.3. Constraint definitions and error impact on the objective function

Constraint	Input					Error
score	protein	protomol	ligand(s)	=	$score_{target}$	$(score_{predicted} - score_{target})^2$
score	protein	protomol	ligand(s)	<	$score_{target}$	$(score_{predicted} - score_{target})^2$ if $score_{predicted} > score_{target}$
score	protein	protomol	ligand(s)	>	$score_{target}$	$(score_{predicted} - score_{target})^2$ if $score_{predicted} < score_{target}$
screening	protein	protomol	$^{+}$ligand(s)	$^{-}$ligand(s)		$100 \cdot (1 - ROC_{area})^2$
geometric	protein	protomol	$^{+}$pose(s)	$^{-}$pose(s)		$(\text{highest } score_{+pose} - score_{-pose})^2$ if $score_{-pose} > \text{highest } score_{+pose}$

Score constraints relate a particular protein and a single ligand or set of ligands to a target score. The user can specify whether the predicted score should be exactly/above/below the target score. Moving in an undesired direction from the target score incurs a squared penalty (see Table 6.3). This is, in fact, the original training regime where the scoring function was tuned to fit experimental binding affinities.[51] In the current formulation, we would create 34 individual score constraints of equal weight, one for each of the 34 protein-ligand complexes, indicating success as an exact match to the experimental K_d. Using additional such constraints, a user could potentially tune the performance of a scoring function for more accurate rank-order prediction of novel ligands. By focusing, for example, on training data that was dominated by the lead series of interest, better predictions of potency for new ligands in the series could result.

Screening constraints allow a user to denote that one set of positive ligands (e.g. a set of cognate ligands) should score measurably higher than a set of negative ligands (e.g. a set of decoys). Performance is assessed by ROC AUC. A function that could flawlessly determine whether a ligand is positive or negative would have an AUC of 1.0. Conversely, a classifier which randomly assigned ligands a positive or negative label would achieve an AUC of 0.5 in the average case. The impact of a screening constraint on the objective function is formulated as the square of its ROC area's deviation from 1.0 (see Table 6.3), scaled by 100 to ensure that its value shares the same effective range as the other constraint types. Using such data, a user can tune a scoring function to perform well in finding new leads for a particular protein of interest in a screening experiment. This particular scenario will be presented in detail in the results that follow, owing to the existence of a large publicly available database for testing.

Geometric constraints offer a method for addressing what are termed "hard failures" in docking. Given an incorrect prediction of a ligand's pose, it may stem from either a failure of the search method (the best pose was not found, but it would have scored best, termed a soft failure). Or, it may stem from a problem in the scoring function: the best-scoring pose may actually score higher than the correct one (a hard failure). A geometric constraint enforces the rule that no incorrect pose may score higher than the best correct pose. Any deviation results in a squared penalty (see Table 6.3). In focused medicinal chemistry efforts that are guided in part by docking, the geometric predictions can be very important. By providing a method to learn from hard docking failures, a user can take advantage of structures where docking predictions were wrong to improve future performance.

Constraints can be organized further into weighted groups. This feature allows one to arbitrate the influence of certain constraints over the objective function. Consider the following scenario: one has 34 protein-ligand complexes whose scores the function should predict exactly (34 score constraints). One also has a set of known actives and inactives for a given protein, necessitating the need for a single screening constraint. It is important to explicitly be able to control the relative importance of these two types of constraints in modifying the scoring function. To ensure that a single constraint is not overwhelmed by the presence of numerous competing constraints, we can place the 34 score constraints in one group and the single screening constraint in a second group. The optimization procedure is implemented such that each constraint group has an equal bearing on the objective function. In this example, the objective function essentially will see first the 34 individual scoring constraints and the single screening constraint as

having equal relative importance. Users may additionally specify a weight be given to a group, providing more control of influence of different data on the objective function.

Figure 6.2 depicts a high level view of the optimization procedure. Our method can be organized concisely into three components: Input → Optimize → Output. The input consists of constraint information and an initial set of parameters from which the optimization will begin. The constraint information is simply a set of proteins and ligands coupled with metadata informing the objective function as to how it should interpret its training data. The initial values used in all experiments were the default Surflex-Dock parameters reported previously.[51, 83]

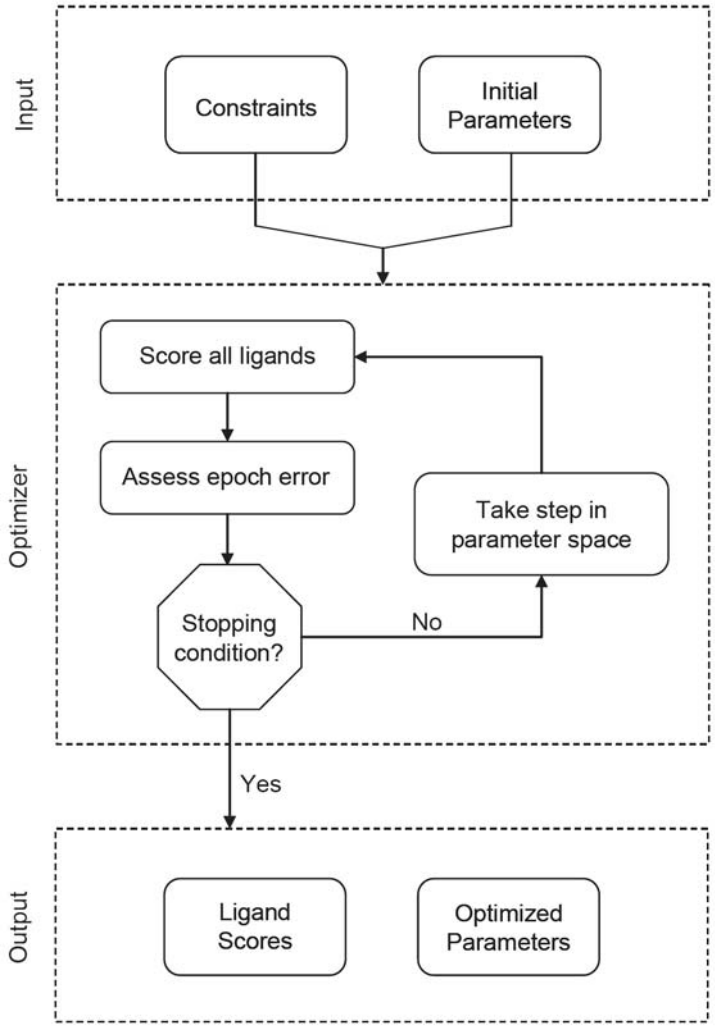


Figure 6.2. Flowchart of the optimization procedure

Each epoch of optimization proceeds as follows:

1. Score all ligands with the current parameters
2. Assess error as defined by the objective function
3. Check for a stopping condition:
    a. Have we exceeded the maximum number of epochs?
    b. Have we reached our error goal?

c. Have we not found a new error extremum for some maximum number of epochs?

4. If we have satisfied a stopping condition, generate output
5. Otherwise, take a step in parameter space
6. Repeat from step 1

The individual steps are described in more detail below.

Step 1: Scoring all ligands

We use the scoring function with the current set of parameters to score each ligand pose. As discussed in the Introduction, one complication that arises from the optimization exercise is that as the scoring function changes, so too does the optimal pose which extremizes the value of the function. Initially, we begin with poses provided as input by the user with the underlying assumption that the provided pose is also the highest scoring pose. However, as parameters change, the original pose may no longer lie at the extremum of the scoring function. The solution is to interleave local pose optimization along with parameter optimization. Pose optimization occurs on a schedule during the overall procedure when a certain number of successful function parameter modification steps have been taken. Following a local gradient-based optimization of the current ligand pose, the new pose is added to a "pose cache" for that ligand. Each time a ligand is scored, all cached poses are scored with the highest score returned as the representative score for this ligand. The results reported here used a pose cache that stored five of the most recent high scoring poses.

Note that the most general approach would require re-docking of ligands whose true pose was unknown. However, due to computational complexity concerns, this was

not implemented. The effect may be approximated by interleaving re-docking between separate invocations of the optimization procedure.

Step 2. Assessing error

The objective function is defined as the mean squared error (MSE) over all constraints n:

Eq. 6.5

$$Objective\_Function = \frac{\sum_{n} \left(Error_{constraint\,i} \right)^2}{n}$$

Refer to Table 6.3 for the error forms of each constraint type. Since the best possible MSE is zero, the procedure seeks to minimize MSE. A *good* step in the course of optimization is defined as one in which the current epoch MSE is lower than the previous epoch.

Step 3. Checking stopping conditions

All three stopping conditions (maximum number of epochs, MSE goal, and maximum number of epochs with no MSE improvement) are user definable options. In this work, we used values 100,000, 0.0001, and 200, respectively.

Step 4. Generate output

The most important output is the newly optimized parameter set, which is a text file containing scoring function parameter values (e.g. `"new.param"`). These can be used immediately by Surflex-Dock to perform a task of interest (scoring function parameters are loaded with `-lparam new.param` as an argument to Surflex-Dock v2.11 or later).

Step 5. Take a step in parameter space

This scheme interleaves two ways of sampling the parameter space: random walking and line optimization. A random walk is used to ensure broad parameter space exploration and to overcome local minima. Line optimization yields precisely optimized local minima from any given starting point. Each search method is used for a number of iterations, then the search method is switched. Of course, many more complex search strategies exist. However, this procedure yielded robust results and required little time for optimization. On a typical example requiring both scoring and screening constraints, the parameter optimization process took under an hour on typical desktop hardware.

6.3.5. Cross-validation: Selecting the proper training regime

In order for the tests on the eight protein targets described to be fair and appropriately "blind" we needed to determine the preferred optimization regime using other data. The goal is to combine protein-specific screening constraints with the scoring constraints that gave rise to the original Surflex-Dock scoring function. The critical issue has to do with the relative weighting of the two types of constraints. To understand how these constraints interact, we selected two DUD proteins not in our prediction set: P38 MAP kinase (P38) and dihydrofolate reductase (DHFR). These were chosen because of their large number of known actives (256 and 201 actives, respectively). We performed 10-fold cross-validation using several group weight combinations. For each training fold iteration, actives were randomly partitioned 30%-70% into training and testing sets. So, while the 34 score constraints provided an anchor for the current scoring function, the protein-specific screening constraint provided pressure to learn to score the active training molecules above the ZINC1 decoy set. The optimizer, which is stochastic, was run beginning with default scoring function parameters three times. The best scoring-

function parameter set by MSE for each fold was chosen to run a screening enrichment test on the remaining 70% of active compounds against the ZINC1 background. We tested multiple constraint group weight combinations, and we computed the mean ROC AUC over the ten cross-validation folds for each weight combination (Table 6.4 summarizes the results). Note that in testing a particular scoring function, a full docking was carried out.

Table 6.4. 10-fold cross validation results

	P38 (Default Function: 0.549)				DHFR (Default Function: 0.750)			
	Group Weights		Tuned Function		Group Weights		Tuned Function	
	Score	Screen	ROC	Stddev	Score	Screen	ROC	Stddev
Scoring overweighted	5	1	0.604	0.027	5	1	0.866	0.015
	4	1	0.61	0.03	4	1	0.877	0.03
	3	1	0.621	0.024	3	1	0.891	0.017
	2	1	0.638	0.032	2	1	0.895	0.009
Equal	1	1	0.663	0.029	1	1	0.911	0.023
Score only	1	0	0.537	0.028	1	0	0.713	0.031
Screen only	0	1	0.699	0.031	0	1	0.941	0.012
Equal	1	1	0.663	0.029	1	1	0.911	0.023
Screening overweighted	1	2	0.676	0.027	1	2	0.941	0.009
	1	3	0.67	0.019	1	3	0.942	0.01
	1	4	0.676	0.021	1	4	0.945	0.007
	1	5	**0.683**	0.024	1	5	**0.945**	0.007

When using default parameters, the scoring function is just better than random on P38 with a mean ROC AUC = 0.549 over the ten folds screened against ZINC1. The scoring combination that gives the screening data zero weight gives nearly the same results, as it should (see row 6 in Table 6.4). Utilizing this weight mixture is equivalent to freeing all scoring function parameters as we re-optimize using only the binding affinities for the 34 complex set. The original parameters appear to be stable under re-optimization, despite employing a more exhaustive search procedure in the present work. Note, the same effect was seen in the DHFR case.

Conversely, zeroing the scoring constraint weight (row 7 of Table 6.4), improved ROC area to almost 0.70 for P38 (from 0.549) and to 0.941 for DHFR (from 0.750). However, by ignoring the scoring constraint, occasionally pathological behavior resulted in terms of the magnitude of the scores computed using the optimized scoring functions. Since the internal docking search strategy makes use of some thresholds on scores, it was important to retain a similar scale. As we increased the relative weight of the screening constraint, we observed both the improvements in screening performance under cross-validation while maintaining a sensible scale where the scores could be interpreted as pK_d. Note that increasing weights on the scoring constraints yielded the expected regression toward the use of only the scoring constraints.

Given the evidence from the cross validation study on P38 and DHFR, we chose to test the optimization scheme on the blind data for the eight targets with two constraint groups: a scoring constraint group defined by the scoring constraints within the 34 complex set; and a screening constraint group comprised of that complex's training actives and a set of decoys. The scoring and screening constraint groups were assigned weights of 1 and 5, respectively.

6.4. Results & Discussion

Table 6.5. ROC areas for the default and tuned function for 8 test cases

						Trained with ZINC1 Decoys			Trained with Rognan Decoys	
	Target	PDB code	N Train	N Test	Default Function ROC AUC	Tuned Function ROC AUC	95% CI	dROC	Tuned Function ROC AUC	95% CI
Pham set training data only	PARP	2pax	15	31	0.888	**0.987**	0.98-1.00	0.099	0.974	0.95-0.99
	HIVPR	1pro	20	38	0.913	**0.964**	0.93-0.99	0.050	0.948	0.90-0.99
	ER	3ert	10	32	0.956	0.968	0.94-0.99	0.013	0.970	0.95-0.98
	Thrombin	1c4v	20	58	0.975	0.980	0.95-1.00	0.005	0.978	0.94-1.00
	TK	1kim	10	12	0.811	0.813	0.66-0.95	0.002	0.814	0.67-0.94
	Trypsin	1qbo	20	33	0.999	0.994	0.98-1.00	-0.005	0.994	0.98-1.00
	FXa	1fjs	**6**	131	0.962	0.921	0.89-0.94	-0.041	0.951	0.93-0.97
	AChE	1e66	**6**	103	0.675	0.534	0.48-0.59	-0.142	0.524	0.47-0.58
More training data	FXa	1fjs	26	111	0.960	**0.978**	0.96-0.99	0.020		
	AChE	1e66	26	83	0.664	0.698	0.65-0.74	0.034		
Multi structure	$AChE_{\alpha}$	1h23	26	83	0.732	**0.753**	0.69-0.81	0.021		

The primary test of the scoring function optimization method is in a screening enrichment assessment against eight different protein targets (see Table 6.3). We have been careful to avoid any contamination of the test by either the active ligands used for scoring function tuning or by the decoys used. The test data for each of the eight targets includes novel active ligands and employs a different set of decoy molecules (the ZINC2 set). We also uniformly applied the procedure that was developed in our preliminary work (which included cross-validation on two other targets). The overall numerical results are presented in Table 6.5, with plots of the relevant ROC curves presented in Figure 6.3 and Figure 6.9.

As has become standard practice, we have characterized screening performance in terms of ROC AUC, and we have also computed 95% confidence intervals to bracket the

performance of the tuned function in each of the eight test cases. The results are broken into three groups, based on the performance changes.

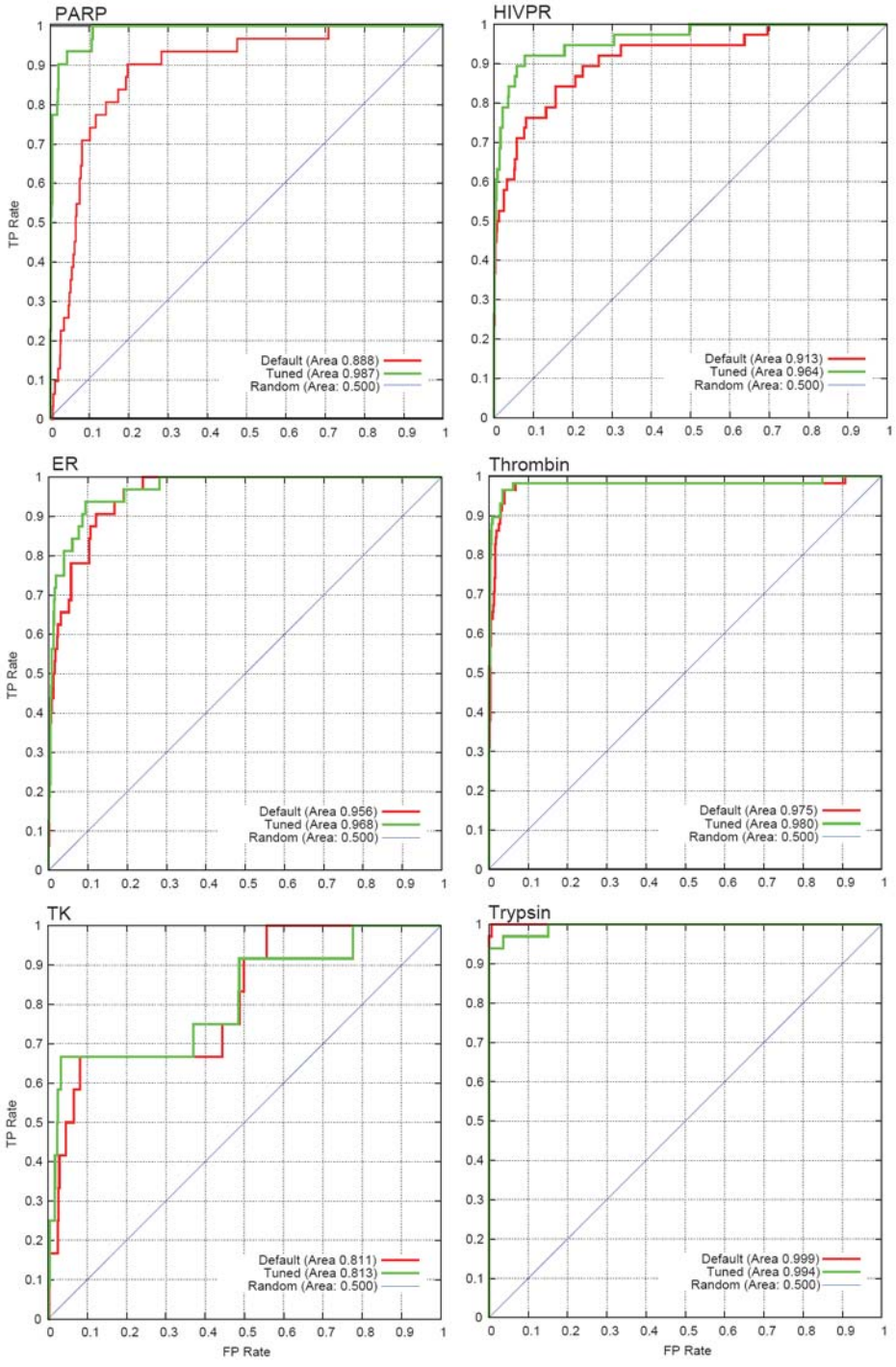


Figure 6.3. ROC plots for 6 targets with sufficient data

6.4.1. Improved Performance: PARP and HIVPR

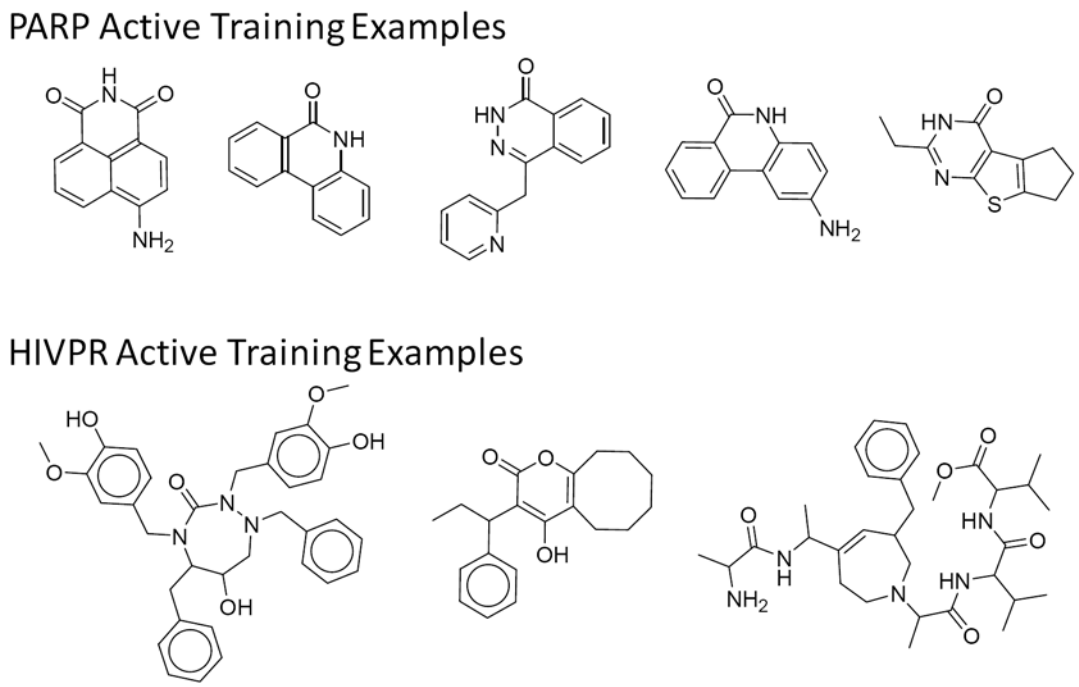


Figure 6.4. Example structures for PARP and HIVPR training ligands

In the six cases where 10 or more active ligands were available in the Pham set, we observed increased or unchanged performance in all cases, with significant improvements in two cases. These two cases (PARP and HIVPR) will be discussed in detail here.

6.4.1.1. PARP

Poly-(ADP-ribose)-polymerase is involved in the response to genomic damage that results in strand breaks. For specific proteins, PARP can add up to 200 residues of ADP-ribose to form branched polymers, which act as binding sites for repair proteins that play a central role in DNA metabolism.[86] The majority of inhibitors used to tune the scoring function for PARP were small and had relatively weak binding, typically in the

micromolar range (see Figure 6.4 for example structures). The first ROC plot of Figure 6.3 corresponds to the test of the PARP-focused tuned scoring function on the blind test data. The improvement in screening enrichment for the blind test molecules in this case was pronounced, with an improvement in ROC AUC of 0.10, corresponding to an increase in true-positive rate from approximately 20% to 90% at a false positive rate of less than 5%.

6.4.1.2. HIVPR

HIV-1 protease is an aspartic protease with a large, solvent-accessible active site with several charged polar moieties both interior and proximally exterior to the pocket. Crystallographic studies have shown that interaction with the interior catalytic triad Asp25-Thr26-Gly27 as well as surface residues, Asp29 and Asp30, is important for enzyme inhibition.[96, 97] The majority of inhibitors used in training bind in the nanomolar range (example structures are shown in Figure 6.4). The second plot of Figure 6.3 shows the ROC curves for HIVPR. The tuned function shows a substantial increase in true positive rates at a false positive rate of 5% relative to the default function from roughly 60% to roughly 85%, corresponding to enhanced early enrichment.

6.4.1.3. Effects on test ligand scores

The ROC plots are sensitive to the relative separation of active from decoy ligands. Increases in the scores for active ligands, decreases for decoys, or a combination of both can lead to improvements in recovery of active ligands and increase in ROC AUC. The cumulative distributions of positive and negative scores for the default and tuned functions (Figure 6.5) reveal the underlying impetus for enrichment improvement.

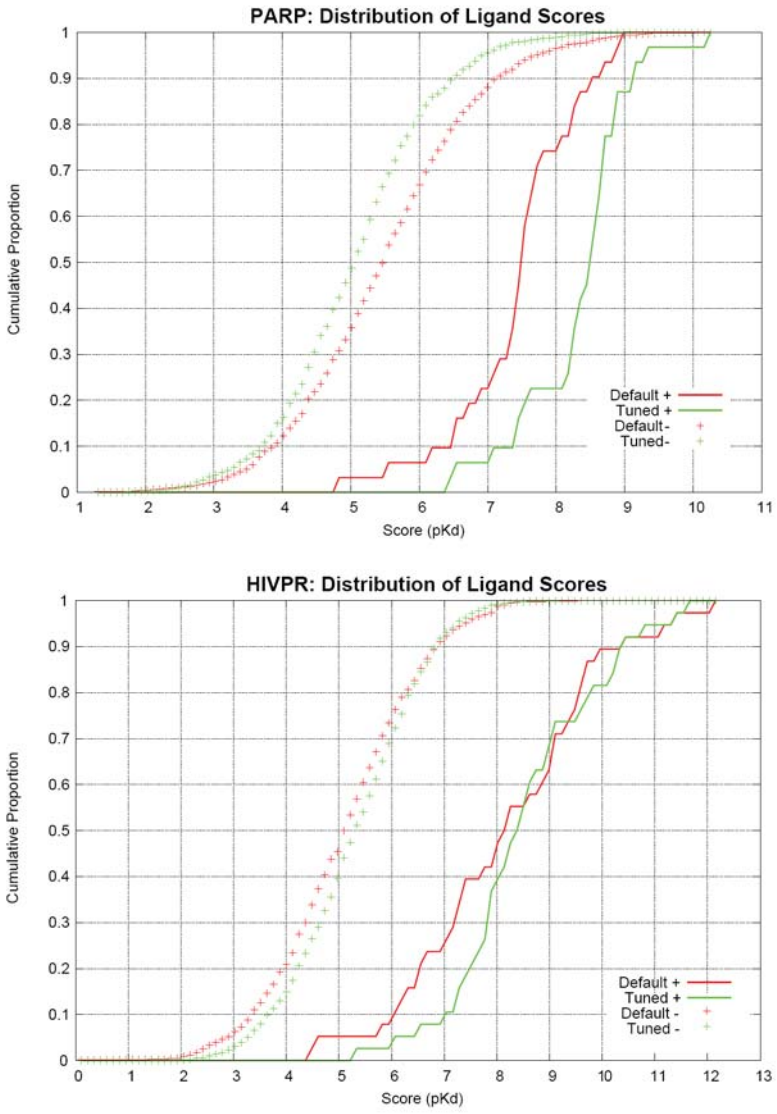


Figure 6.5. Cumulative distribution of test ligand scores for PARP and HIVPR

In the case of PARP, the increase in ROC AUC from 0.89 to 0.99 for the tuned function stemmed from a decrease in the scores of the decoys relative to the untuned function with a simultaneous increase in the scores of the active ligands. The bulk of the actives, when docked with the tuned scoring function, had scores approximately 1 log unit higher than when docked using the default scoring function (this corresponds to the rightward shift from the solid red curve to the solid green curve in the top plot of Figure

6.5). Conversely, the inactives exhibited decreases of roughly 0.5 log units. In the case of HIVPR, performance increased from a ROC AUC of 0.913 to 0.964. However, in this case, the distribution of decoy scores changed only slightly and did so in the *wrong* direction. The improvement in enrichment came from a significant upward shift of the lowest scoring active ligands by about 1.0 log units. With the default function, 40% of actives had $pK_d < 7.5$, but only 20% of actives scored by the tuned functions had $pK_d < 7.5$.

6.4.1.4. Effects on Surflex-Dock function terms

The underlying reasons for the performance increases observed with PARP and HIVPR stemmed from different sources. In the former case, we observed increased ability to recognize actives and reject decoys. In the latter case, both sets of scores increased, but with a specific advantage to the actives. Inspection of the individual terms of the scoring function before and after the optimization procedure (Figure 6.6) lends insight into the reasons for these differences. Three plots are given for each case, showing the default and tuned functions for the hydrophobic, polar, and polar mismatch terms. The axes are the same as for Figure 6.1, with the Y axis being the interaction score in pK_d, and with the X axis being the inter-atomic surface distance in Angstroms. Negative distances indicate nominal interpenetration of van der Waals radii; note that radii for polar atoms are not scaled, so ideal polar contacts exhibit numerical interpenetration.

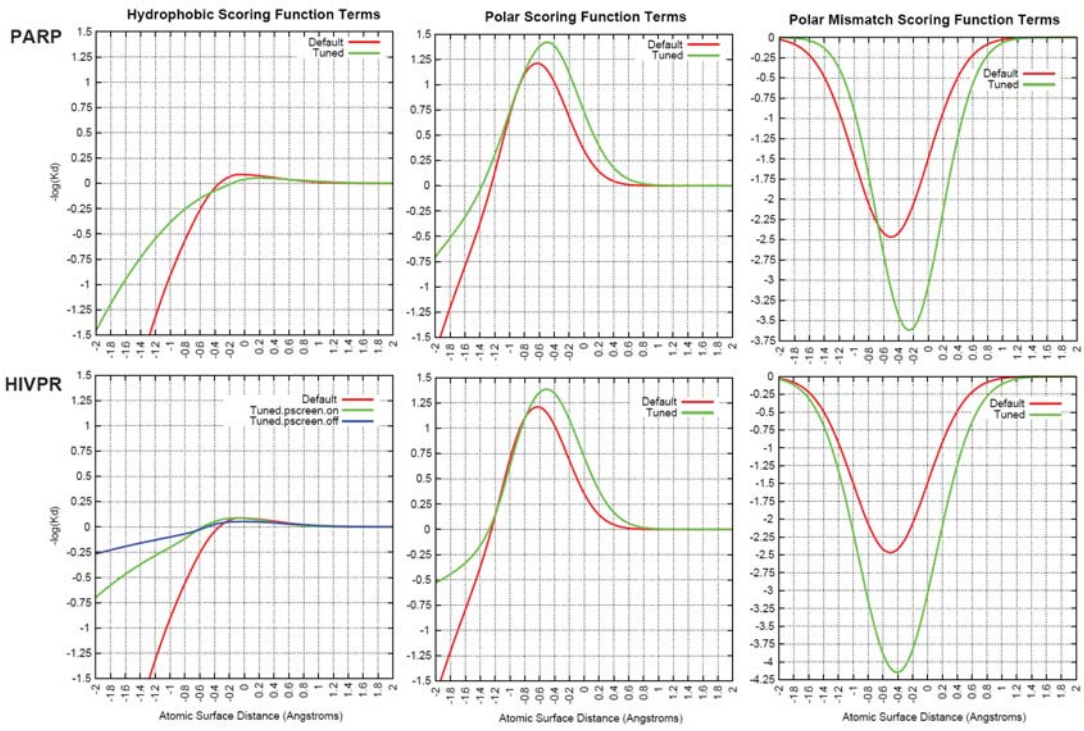


Figure 6.6. Key function terms for PARP and HIVPR: Effects of tuning

The hydrophobic terms show markedly different modifications in response to tuning for PARP and HIVPR. In the former case (top left plot), the penalty for atomic surface interpenetration is decreased somewhat, and the area of positive hydrophobic interaction (from the Gaussian in Eq. 6.1) is both more narrow and has lower amplitude. In the latter case, the softening of the overlap penalty is more significant, and the area of positive interaction *increases*. The tuned scoring function parameters are given in Table 6.6. The decrease in sensitivity to inter-atomic clashes is reflected in the value of the **`hrd`** parameter, which changed from -0.95 (default function) to -0.16 (tuned function). For HIVPR, we also considered the effect of generating the training poses (and testing the resulting tuned function) *without* the use of Surflex's ligand pre-minimization and post-docking all-atom optimization. These procedures are part of the default screening

protocol of Surflex, since they help decrease dependence on input ligand preparation and allow access to Cartesian movements that can ameliorate clashes between the protein and ligand.[70] This can be especially important for large ligands. The blue curve in Figure 6.6 shows that the tuned clash penalty is even softer when the docking process is restricted to a ligand's internal coordinates. In order to obtain reasonably high scores for the large HIVPR ligands, it is necessary to relax the clashing penalty, and this effect is larger when the ligands are unable to bend outside of torsional and alignment space. Differences in docking protocol can yield marked differences in the resulting tuned functions, so particular attention must be given to replicating the protocol used for generating training data as will be used for operational application of the resulting tuned function.

Table 6.6. Parameter values of default and tuned functions for PARP and HIVPR

Param	Default	PARP Tuned	HIVPR Tuned
stz	0.0898	0.0614	0.0891
str	-0.0841	-0.0911	-0.0756
sts	0.6213	1.1162	0.4461
stm	0.1339	0.1191	0.151
srm	0.488	0.007	0.5279
hrd	**-0.945**	**-0.3602**	**-0.1634**
poz	**1.2388**	**1.5443**	**1.4769**
por	-0.1796	-0.1514	-0.282
pos	0.3234	0.4196	0.3908
pom	0.6313	0.5422	0.5098
hpl	0.6139	0.6787	0.7248
csf	0.5	0.1895	0.1753
pr2	**-2.52**	**-3.7662**	**-4.4127**
prm	0.501	0.2568	0.4102
ms	0.5	0.3966	0.5437
ent	-0.2137	-0.4551	-0.259
con	-1.0406	-0.2445	-0.965

The differences in clashing penalty between the PARP and HIVPR cases can be seen in the polar terms (middle plots of Figure 6.6), since the **`hrd`** parameter also controls excessive interpenetration between polar atoms. Apart from that, the positive

aspect of polar interactions exhibited similar behavior in the two tuned function, with both increases in the maximal value of a single polar contact (controlled by the **poz** parameter) and a slight increase in the distance from which complementary polar contacts obtain positive scores (controlled by the **pom** parameter, and corresponding to a change from 1.97 Å to 2.09 Å in inter-atomic center distances for N-H and C=O).

In contrast to the decrease in the repulsive effect of inter-atomic clashes, we see a marked *increase* in the repulsive effect of proximal same-charge moieties for both PARP and HIVPR. The rightmost plots of Figure 6.6 show increases both in the overall magnitude of same-charge repulsion penalty (controlled by the **pr2** parameter) as well as an increase in the distance at which the effect becomes important (controlled by the **prm** parameter). In the case of HIVPR, the magnitude of the same-charge repulsion penalty increased 75% over the default function, and for PARP, it increased approximately 45%.

6.4.1.5. Examples of effects on docked actives and decoys

The changes in the tuned scoring functions are evident in the behavior of specific test ligands. Figure 6.7 (left panel) shows the experimentally determined pose of a cyclic urea HIVPR inhibitor bound to the protease (PDB code: 1BVE). Note the position of the hydroxyl groups of the central 7-member ring relative to the catalytic aspartic acids ASP-A25 and ASP-B25 in dark blue. The test ligand (ZINC03833842) has a very similar structure and is shown in its docked pose using the tuned scoring function in the middle panel. Despite a poor ring geometry that was present in the input structure (ring search within Surflex was not employed), the docked pose with the tuned scoring function is reasonable, with sensible interactions between the hydroxyls on the central ring system to the aspartic acid residues as well as good placement of the "arms" of the ligand. The right

panel shows the same ligand docked using the default scoring function. In this case, the inhibitor was clearly docked poorly.

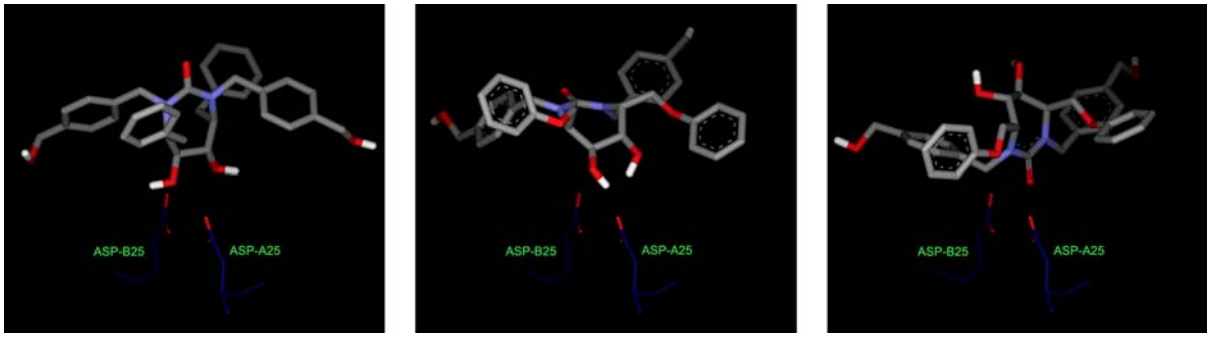


Figure 6.7. Behavior of an active test ligand within the HIVPR active site

This ligand was ranked 45th out of 1038 molecules (38 actives + 1000 ZINC2 decoys) by the default scoring function. However, when re-docked using the tuned function, it was ranked 1st. The pose resulting from application of the tuned function is very different, owing to the differences in the penalty terms. If we *rescore* the tuned function pose using the default function, the steric clashing term alone generates more than 10 pK_d units in additional penalty. The large difference in penalty terms between the two functions leads to widely different poses among the active ligands when using the different functions. Among all of the active test ligands, the typical deviation in top-scoring pose between the application of the two functions was quite high (mean rmsd of 5.5Å), reflecting both the flexibility of the ligands as well as the substantial change in the scoring function, especially the parameters that controlled steric clashing. With HIV protease, it is known that ligand binding causes substantial conformational changes to the enzyme.[97] Treatment of the protein structure as rigid has obvious computational benefits in terms of search complexity, but in cases where this treatment is especially inaccurate (e.g. with large ligands), lowering the steric penalty serves as a surrogate for modeling

induced fit. The scoring function optimization scheme provides a systematic method to exploit such protein-specific features.

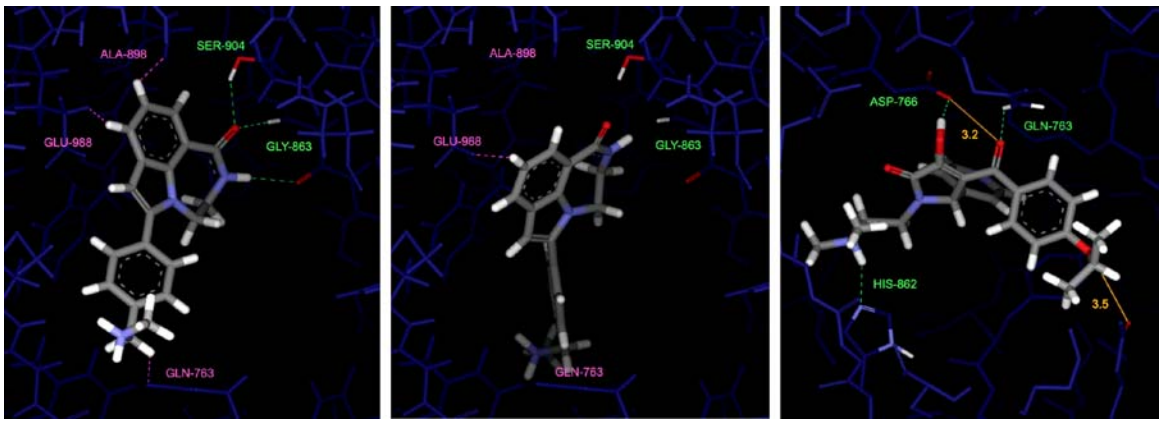


Figure 6.8. Test ligands for PARP

In the case of PARP, the active site is much smaller, and it appears to undergo a smaller degree of movement on binding inhibitors. This is evidenced by the stronger penalty for steric clashes as compared with HIVPR, and it also shows in the degree to which docking with the tuned scoring function yields different top scoring poses compared with docking with the default function. For PARP, 17/31 test ligands dock within 0.5Å rmsd between tuned and default functions, with only 4 ligands above 1.0Å rmsd, including 2 above 2.0Å rmsd. Figure 6.8 shows the ligand with the largest geometric deviation between docking with the two different scoring functions. The left panel shows the pose generated using the tuned function, and the middle panel shows the pose from the default function. While the pose is not grossly different, as in the HIVPR case shown above, the pose from the tuned function is clearly closer to correct, making the appropriate contacts common to PARP inhibitors. Note that this case was the exception. Most of the top-scoring poses changed very little, but the tuned function yielded systematically higher scores for the actives. In the rightmost panel, a relatively

high-ranking decoy (ZINC04819306) is shown as docked using the default scoring function (it ranked 138/1031 molecules). The tuned function, when used to rescore the poses produced by docking using the default function ranked the decoy at 736/1031. In the full docking that gave rise to the ROC performance shown in Figure 6.3, this decoy ranked 961/1031. So, while the changes in the scoring function had relatively subtle effects on the active ligands, the effects on the decoys were more substantial.

6.4.2. Small Performance Changes: Four targets

Optimization yielded small, but not statistically significant improvements in three of four cases (ER, Thrombin, and TK), and produced an insignificant decrease in performance in the other case (Trypsin). Perusal of the training results revealed that there was little information to be extracted from the input data. The default function yielded a mean ROC AUC in the training data of 0.95 (minimum 0.93, maximum 1.0). Following optimization, the average training performance was 0.98 (minimum 0.97, maximum 1.0). While the training procedure yielded the desired effect on the training data, given that there was very little room for improvement, the net result was that little improvement was seen in the test data. Lacking a significant number of examples that are poorly ranked, there should be no expectation of a significant change after training. However, the fact that the function parameters are stable in this situation is a useful characteristic. This is aided by inclusion of the original set of 34 complexes as part of the weighted training regime.

6.4.3. Performance Decreases: Too little training data

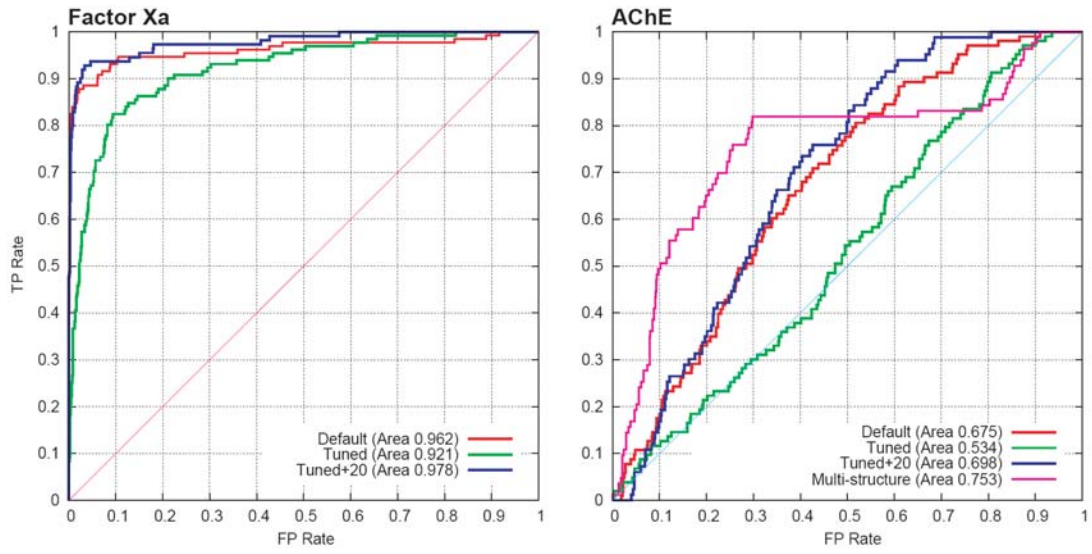


Figure 6.9. ROC plots for Factor Xa and ACHE: the effect of increased training set size

In the two cases (FXa and AChE) where just six molecules were available as active ligands from which to tune the scoring function, overall performance on the test libraries was reduced in a statistically significant fashion (see Table 6.5). To test whether the lack of active ligand examples was the source of the reduction in performance, we added 20 randomly selected actives from the test sets for FXa and AChE to their training sets. After retuning the scoring functions using the same procedure as before, the addition of active training ligands was shown to reverse the degradation. Figure 6.9 shows the ROC plots corresponding to these additional experiments. In the case of FXa, the tuned function yielded a significant increase, though just marginally in a statistical sense. In the case of AChE, while performance was improved (instead of substantially decreased), the screening utility was still low.

Training on the very small number of actives pushed the optimization protocol toward specific parameter changes for a skewed population of actives. Increasing the

number of training examples avoided this skew, but still did not address the problem of weak screening performance. AChE contains a long, narrow binding pocket formed by the aromatic rings of 14 conserved residues.[98] Two active sites are known to exist: a main site located at the bottom of the aromatic gorge, and a secondary site 14Å away near the opening of the binding cavity.[99] This target represents a difficult case in that inhibitors may occupy just the main site or interact with both sites. The active site used in our study (PDB code: 1E66) was taken from the structure of AChE in complex with huprine X,[100] a small molecule with only 40 atoms that binds the primary active site at the bottom of the long pocket. We repeated the optimization under identical conditions with the extended training set, but we used a different structure for AChE (PDB code: 1H23). In this structure, the bound ligand was much larger, huperzine A, which occupies both the primary and secondary sites.[101] We then executed the screening enrichment test making use of *both* structures, keeping the highest score from *either* run as the representative ligand score. Here we used the strategies introduced in Section 2.2.4 on receptor flexibility. The results from this experiment were encouraging. Under this treatment, the ROC AUC of the tuned function improved to 0.753, which was significantly better than the default function performed using only a single structure. In this case, scoring function tuning alone was not sufficient to overcome serious limitations imposed by the structure that was used for docking.

6.4.4. The Effect of Decoy Sets

Our results show very little effect of changing the decoy set used in training. Using either the Rognan decoy set or the ZINC1 set yielded nearly identical performance (see Table 6.5). This makes sense, since the effect of a decoy set in the optimization

exercise is based upon the small proportion of difficult cases that show up as nominal false positives when using the default scoring function. As long as a decoy set contains some reasonable candidates to be such false positives, it will serve adequately. Note, however, that there are limits to this. A decoy set containing only a large collection of different Fullerenes probably would be of no utility in refining scoring functions for the proteins under study. With respect to the effect of different decoy sets on *testing* the performance of docking systems, experience is somewhat mixed. While our results[83] agree with those of Irwin and Shoichet[94] that the ZINC1 set (referred to as the "Jain set" in that work) is more challenging than the Rognan set, the difference we observed was much smaller in magnitude.[70]

In this work, we have chosen to continue to use decoy sets that have been constructed with no specific knowledge of active ligand structures. We have done so for three reasons. First, it provides a direct comparison to our previous studies, which employed the same (or similarly constructed) decoy sets as well as overlapping protein structures.[30, 70, 83] Second, while the statistical likelihood of finding true ligands among a random collection of screening compounds is known to be low (1/1000 to 1/10,000), it is not at all clear what the likelihood might be if one selects a set of decoys that have similar size, charge, and hydrophobicity characteristics, though it is almost certainly higher. Third, even decoy sets that have been shown to have relatively non-drug-like properties are sufficient to distinguish the performance of many docking protocols.[69]

6.4.5. Accuracy of Training Poses

One might expect that having close to correct poses for active ligands used in training would have a beneficial impact on the tuned scoring functions. This is a difficult

effect to measure, in part because one typically employs a single protein structure in screening, so we have used single structures in our experiments. While all of the active ligands in the Pham set (by construction) had known bound poses, since protein conformations change, not all of those poses would serve as appropriate starting points using a single protein structure. Rather than using those directly, we re-docked the active ligands using more aggressive search parameters. In cases where a pose existed within 2Å rmsd of correct, and whose score was within 80% of the highest score for any pose, we replaced the highest scoring pose with this pose for purposes of training. After this filtering method was applied, 76% our training poses were within 2Å rmsd of correct, vs. 46% without filtering. We repeated the optimization experiment summarized in Table 6.5. Virtually no difference in test performance was detected across all eight complexes. To a degree, this parallels what was found by Warren et al.,[102] where they observed little relationship between docking accuracy and screening utility. However, this is not an intuitive result and requires more investigation.

6.5. Conclusion

The results reported here clearly demonstrate that the parameters governing a scoring function for protein-ligand interactions can be optimized to improve performance for a particular task. Moreover, the multiple constraint approach for constructing an objective function for optimization of scoring functions introduces an extensible framework for making use of many types of data. In this work, we have optimized the Surflex-Dock scoring function to enhance screening enrichment for particular targets. Significant screening improvement was possible when training on as few as 15 known actives, with substantial increases in early enrichment for HIV protease and PARP. In all

cases with 10 or more actives, screening performance was improved or stayed the same. For those complexes with less than 10 training ligands, use of the very small data sets was problematic but was reversed by including additional data.

As a practical matter, many practitioners of docking spend a great deal of effort on very small numbers of targets. Frequently, such situations involve access to large quantities of proprietary crystallographic structures as well as structure-activity data. While refinement of scoring functions for docking will continue toward addressing the general case of application to any target, focused refinement may prove to be of great utility to those whose interests lie in studying a *particular* target as opposed to caring about the generality of the methodology. By providing the tools for rapid optimization of scoring function parameters to *users*, we hope that the subtle parameter refinements seen here to yield large changes in performance will be demonstrated on targets "in the wild."

As a theoretical matter, a rigorous treatment of the multiple instance problem (which pose do we listen to?) coupled with creative use of objective functions (can we enforce a constraint that this ligand or pose is supposed to score better than these others?) may prove to be of use beyond scoring functions in docking or methods in 3D QSAR. The place where such an approach has obvious applicability, but has not yet been tried to our knowledge, is in the development and refinement of empirical scoring functions for use in protein folding.

Chapter 7

Conclusion and future directions

This dissertation has presented a body of work for the development and validation of empirical scoring functions used in molecular docking. These functions paint a picture of the complicated binding free energy surface that arises from interactions between a protein and ligand. Supervised learning techniques allowed us to leverage informative datasets to extrapolate new details within the picture. By generalizing these techniques and making them openly available, we enable others to personalize their picture and highlight the specific features they would like to see.

Surflex[30] acted as our development testbed; we took a very good algorithm and improved it. Large amounts of quality molecular data are required both for tuning our function and validating that it worked. This is imperative to keep training and testing data completely separate – a well-accepted practice in the machine learning field that is only recently being adopted in the field of molecular docking. Chapter 3 details pdbgrind, a rapid and systematic method for creating molecular data sets from the Protein Data Bank.[2] Every molecular docker incorporates a search strategy and a score strategy. Chapter 4 sees some exploration into the search side of the coin when additional information is introduced in the form of enhanced protomols. Chapter 5 shows how

supervised learning can take advantage of relevant negative data to tune single function parameters.

The culmination of this dissertation is Chapter 6 where we take the learning method established in Chapter 5 and generalize it to work for the entire scoring function. Constraint-Based Optimization embeds user knowledge specific to their training data to restrain the optimization. In doing so, we unlock the power of parameter refinement to the user and allow them to customize their functions to suit a particular task. This is particularly useful for those in possession of proprietary datasets that they would like to see leveraged into a customized scoring function.

Our method provides an open and robust way of optimizing empirical functions for scoring accuracy, docking accuracy, and screening utility. The open infrastructure (see Appendix C.1.2) allows the implementation of additional constraints. These are limited only by the creativity of the designer in their insertion of knowledge into the training regimen. One possible constraint could be a cross-docking requirement where off-target effects are minimized. Another might model protein flexibility by employing multiple receptor structures and using consensus scoring on both to optimize screening utility. An experimental group with assay capabilities may want to modify the existing screening constraint to listen to only the top 1-5% of the ranked library since molecules below that threshold may never be validated experimentally.

This platform can also be used to extend the capabilities of a scoring function. The desolvation term within the Surflex function was given insignificant weight during its initial training.[51] This term counted the number of missing hydrogen bonds between the protein and ligand upon binding. Preliminary work aimed to revise the solvation

model by penalizing uncompensated polar moieties as weighted by their solvent accessible surface area before and after protein-ligand binding. Our constraint-based optimization system was used to find the controlling parameters for this penalty. We found traction for our term by enforcing a scoring constraint on high scoring decoys (score < 4.0 pK_d) with the assumption that some of those decoys exhibited patterns of improper desolvation. Such improvements become possible only with a general optimizer and a strong intuition of the data.

The broader theoretical impact of this work is not limited solely to molecular docking. Our methods contain successful strategies for machine learning problems with multiple instantiations (i.e. different conformations and alignments of molecules represented as 3D objects). Parameter estimation regimes that yield good solutions for modeling protein-ligand interactions may be generalizable to related problem areas such as protein folding. The sibling field of 3D QSAR (Quantitative Structure Activity Relationships) shares a similar problem space to molecular docking absent only a known protein structure. Here our search strategy of docking probes to create a "negative" image of the binding site could be applied in reverse; by docking probes to a superposition of known actives, we create a "positive" image of the protein active site. Additionally, the functions used to score such ligand-based models could readily be optimized using our methods.

Beyond its theoretical contributions, this technology should be useful in practical terms as well. Therapeutic development stands to benefit from our method's facilitation of rapid identification of active lead compounds from high-throughput virtual screening or through directed design. Drug discovery and development is a risky and expensive

endeavor which is hampered by inefficient exploitation of knowledge. With the recent explosion of biological data, the need for methods that can leverage data efficiently into actionable knowledge has never been greater. Bioinformatics applies computational science to problems in the biological domain. In the end by harnessing the power of the computer, we enable greater opportunities for discovery in the pursuit of human health.

References

1. Liolios, K., et al., *The Genomes On Line Database (GOLD) in 2007: status of genomic and metagenomic projects and their associated metadata.* Nucleic Acids Res, 2007.
2. Berman, H., K. Henrick, and H. Nakamura, *Announcing the worldwide Protein Data Bank.* Nat Struct Biol, 2003. **10**(12): p. 980.
3. Dolle, R.E., et al., *Comprehensive Survey of Chemical Libraries for Drug Discovery and Chemical Biology: 2006.* J Comb Chem, 2007.
4. Dietterich, T.G., Lathrop, R. H., & Lozano-Perez, T., *Solving the Multiple Instance Problem with Axis-Parallel Rectangles.* Artificial Intelligence, 1997. **89**(1-2): p. 31-71.
5. Jain, A.N., et al., *A shape-based machine learning tool for drug design.* J Comput Aided Mol Des, 1994. **8**(6): p. 635-52.
6. Walsh, G., *Biopharmaceutical benchmarks 2006.* Nat Biotechnol, 2006. **24**(7): p. 769-76.
7. Aggarwal, S., *What's fueling the biotech engine?* Nat Biotechnol, 2007. **25**(10): p. 1097-104.
8. PhRMA, *Drug Discovery and Development.* 2007(February).
9. DiMasi, J.A., Grabowski, Henry G., *The Cost of Biopharmaceutical R&D: Is Biotech Different?* Managerial and Decision Economics, 2007. **28**(4-5): p. 469-479.
10. Kaitin, K.I.e., *Therapeutic class a critical determinant of drug development time & cost.* Tufts Center for the Study of Drug Development Impact Report, 2007. **9**(5).
11. Lipinski, C. and A. Hopkins, *Navigating chemical space for biology and medicine.* Nature, 2004. **432**(7019): p. 855-61.
12. Smith, A.e., *Screening for Drug Discovery.* Nature, 2002. **418**: p. 452-459.
13. Walters, W.P. and M. Namchuk, *Designing screens: how to make your hits a hit.* Nat Rev Drug Discov, 2003. **2**(4): p. 259-66.
14. Creighton, T.E., *Proteins: Structures and Molecular Properties*. 2002.
15. Harper, D., *Online Etymology Dictionary.* 2001.
16. Dill, K.A., Bromberg, Sarina, *Molecular Driving Forces: Statistical Thermodynamics in Chemistry and Biology*. 2003.
17. Krumrine, J., Raubacher, F., Brooijmans, N., Kuntz, I., *Principles and Methods of Docking and Ligand Design*, in *Structural Bioinformatics*, P.E. Bourne, Weissig, Helge, Editor. 2003. p. 443-476.

18. Kollman, P.A., *Free Energy Calculations: Applications to Chemical and Biochemical Phenomena.* Chem. Rev., 1993. **93**: p. 2395-2417.
19. Head, M.S.G.J.A.G.M.K., *"Mining minima": direct computation of conformational free energy.* J.Phys.Chem.A, 1997. **101**: p. 1609-1618.
20. Jain, A.N., *Virtual screening in lead discovery and optimization.* Curr Opin Drug Discov Devel, 2004. **7**(4): p. 396-403.
21. Cheng, Y. and W.H. Prusoff, *Relationship between the inhibition constant (K1) and the concentration of inhibitor which causes 50 per cent inhibition (I50) of an enzymatic reaction.* Biochem Pharmacol, 1973. **22**(23): p. 3099-108.
22. Mackerell, A.D., Jr., *Empirical force fields for biological macromolecules: overview and issues.* J Comput Chem, 2004. **25**(13): p. 1584-604.
23. Cornell, W.D.C., P.; Bayly, C. I.; Gould, I. R.; Merz, K. M.;, D.M.S. Ferguson, D. C.; Fox, T.; Caldwell, J. W.;, and P.A. Kollman, *A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules.* J Am Chem Soc, 1995. **117**: p. 5179-5197.
24. Jorgensen, W.L.T.R., *Development and Testing of the OPLS All-Atom Force Field on Conformational Energetics and Properties of Organic Liquids.* 1996. **118**: p. 11226-11236.
25. MacKerell, A.D., Jr.; Bashford, D.; Bellott, M.; Dunbrack Jr., R. L.;, et al., *All-Atom Empirical Potential for Molecular Modeling and Dynamics Studies of Proteins.* J Phys Chem, 1998. **102**: p. 3586-3616.
26. Mayo, S.L., Olafson, Barry D., Goddard III, William A., *DREIDING: A Generic Force Field for Molecular Simulations.* J Phys Chem, 1990. **94**(26): p. 8897-8909.
27. Winston, P.H., *Artificial Intelligence.* 1992.
28. Duch, W., K. Swaminathan, and J. Meller, *Artificial intelligence approaches for rational drug design and discovery.* Curr Pharm Des, 2007. **13**(14): p. 1497-508.
29. Bellman, R., *Adaptive Control Processes: A Guided Tour*. 1961: Princeton University Press.
30. Jain, A.N., *Surflex: fully automatic flexible molecular docking using a molecular similarity-based search engine.* J Med Chem, 2003. **46**(4): p. 499-511.
31. Ewing, T.J., et al., *DOCK 4.0: search strategies for automated molecular docking of flexible molecule databases.* J Comput Aided Mol Des, 2001. **15**(5): p. 411-28.
32. Moustakas, D.T.P., Scott C.H.; Kuntz, Irwin D., *A Practical Guide to DOCK 5*, in *Virtual Screening in Drug Discovery*, J.S. Alvarez, Brian, Editor. 2005.
33. Rarey, M., et al., *A fast flexible docking method using an incremental construction algorithm.* J Mol Biol, 1996. **261**(3): p. 470-89.
34. Friesner, R.A., et al., *Glide: a new approach for rapid, accurate docking and scoring. 1. Method and assessment of docking accuracy.* J Med Chem, 2004. **47**(7): p. 1739-49.
35. Goodsell, D.S., G.M. Morris, and A.J. Olson, *Automated docking of flexible ligands: applications of AutoDock.* J Mol Recognit, 1996. **9**(1): p. 1-5.
36. Kitchen, D.B., et al., *Docking and scoring in virtual screening for drug discovery: methods and applications.* Nat Rev Drug Discov, 2004. **3**(11): p. 935-49.

37. Halgren, T.A., et al., *Glide: a new approach for rapid, accurate docking and scoring. 2. Enrichment factors in database screening.* J Med Chem, 2004. **47**(7): p. 1750-9.
38. Jones, G., et al., *Development and validation of a genetic algorithm for flexible docking.* J Mol Biol, 1997. **267**(3): p. 727-48.
39. Morris, G.M.G., D. S.; Halliday, R. S.; Huey R; Hart W. E.; Belew R. K.; Olson A. J., *Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function.* J. Comp. Chem., 1998. **19**(1639-1662).
40. Chivian, D.R., Timothy; Bonneau, Richard; Baker, David, *AB INITIO Methods*, in *Structural Bioinformatics*, P.E. Bourne, Weissig, Helge, Editor. 2003. p. 547-557.
41. Brooijmans, N. and I.D. Kuntz, *Molecular recognition and docking algorithms.* Annu Rev Biophys Biomol Struct, 2003. **32**: p. 335-73.
42. Mangoni, M., D. Roccatano, and A. Di Nola, *Docking of flexible ligands to flexible receptors in solution by molecular dynamics simulation.* Proteins, 1999. **35**(2): p. 153-62.
43. Pak, Y.W., Shaomeng, *Application of a Molecular Dynamics Simulation Method with a Generalized Effective Potential to the Flexible Molecular Docking Problems.* J.Phys.Chem.B, 2000. **104**(2): p. 354-359.
44. Leach, A.R., *Ligand docking to proteins with discrete side-chain flexibility.* J Mol Biol, 1994. **235**(1): p. 345-56.
45. Knegtel, R.M., I.D. Kuntz, and C.M. Oshiro, *Molecular docking to ensembles of protein structures.* J Mol Biol, 1997. **266**(2): p. 424-40.
46. Kumar, S., et al., *Folding and binding cascades: dynamic landscapes and population shifts.* Protein Sci, 2000. **9**(1): p. 10-9.
47. Kuntz, I.D., et al., *A geometric approach to macromolecule-ligand interactions.* J Mol Biol, 1982. **161**(2): p. 269-88.
48. Shoichet, B.K., A.R. Leach, and I.D. Kuntz, *Ligand solvation in molecular docking.* Proteins, 1999. **34**(1): p. 4-16.
49. Bohm, H.J., *The development of a simple empirical scoring function to estimate the binding constant for a protein-ligand complex of known three-dimensional structure.* J Comput Aided Mol Des, 1994. **8**(3): p. 243-56.
50. Eldridge, M.D., et al., *Empirical scoring functions: I. The development of a fast empirical scoring function to estimate the binding affinity of ligands in receptor complexes.* J Comput Aided Mol Des, 1997. **11**(5): p. 425-45.
51. Jain, A.N., *Scoring noncovalent protein-ligand interactions: a continuous differentiable function tuned to compute binding affinities.* J Comput Aided Mol Des, 1996. **10**(5): p. 427-40.
52. Sharp, K.A., *Potential Functions for Virtual Screening and Ligand Binding Calculations: Some Theoretical Considerations*, in *Virtual Screening in Drug Discovery*, J.S. Alvarez, Brian, Editor. 2005.
53. Muegge, I. and Y.C. Martin, *A general and fast scoring function for protein-ligand interactions: a simplified potential approach.* J Med Chem, 1999. **42**(5): p. 791-804.
54. Gohlke, H., M. Hendlich, and G. Klebe, *Knowledge-based scoring function to predict protein-ligand interactions.* J Mol Biol, 2000. **295**(2): p. 337-56.

55. Wang, R., Y. Lu, and S. Wang, *Comparative evaluation of 11 scoring functions for molecular docking.* J Med Chem, 2003. **46**(12): p. 2287-303.
56. Wang, R., L. Lai, and S. Wang, *Further development and validation of empirical scoring functions for structure-based binding affinity prediction.* J Comput Aided Mol Des, 2002. **16**(1): p. 11-26.
57. Gehlhaar, D.K., et al., *Molecular recognition of the inhibitor AG-1343 by HIV-1 protease: conformationally flexible docking by evolutionary programming.* Chem Biol, 1995. **2**(5): p. 317-24.
58. Charifson, P.S., et al., *Consensus scoring: A method for obtaining improved hit rates from docking databases of three-dimensional structures into proteins.* J Med Chem, 1999. **42**(25): p. 5100-9.
59. Perola, E.W., P.W.; Charifson, P.S., *An Analysis of Critical Factors Affecting Docking and Scoring*, in *Virtual Screening in Drug Discovery*, J.S. Alvarez, Brian, Editor. 2005.
60. Halgren, T.A., *Merck Molecular Force Field: I. Basis, form, scope, parameterization, and performance of MMFF94.* J. Comp. Chem., 1996. **17**: p. 520-552.
61. Verkhivker, G.M., et al., *Deciphering common failures in molecular docking of ligand-protein complexes.* J Comput Aided Mol Des, 2000. **14**(8): p. 731-51.
62. Kellenberger, E., et al., *Comparative evaluation of eight docking tools for docking and virtual screening accuracy.* Proteins, 2004. **57**(2): p. 225-42.
63. *FRED*, Open Eye Scientific Software: Santa Fe, NM.
64. Verdonk, M.L., et al., *Improved protein-ligand docking using GOLD.* Proteins, 2003. **52**(4): p. 609-23.
65. Zavodszky, M.I., et al., *Distilling the essential features of a protein surface for improving protein-ligand docking, scoring, and virtual screening.* J Comput Aided Mol Des, 2002. **16**(12): p. 883-902.
66. McMartin, C. and R.S. Bohacek, *QXP: powerful, rapid computer algorithms for structure-based drug design.* J Comput Aided Mol Des, 1997. **11**(4): p. 333-44.
67. Bissantz, C., G. Folkers, and D. Rognan, *Protein-based virtual screening of chemical databases. 1. Evaluation of different docking/scoring combinations.* J Med Chem, 2000. **43**(25): p. 4759-67.
68. Kortemme, T., A.V. Morozov, and D. Baker, *An orientation-dependent hydrogen bonding potential improves prediction of specificity and structure for proteins and protein-protein complexes.* J Mol Biol, 2003. **326**(4): p. 1239-59.
69. Jain, A.N., *Scoring functions for protein-ligand docking.* Curr Protein Pept Sci, 2006. **7**(5): p. 407-20.
70. Jain, A.N., *Surflex-Dock 2.1: robust performance from ligand energetic modeling, ring flexibility, and knowledge-based search.* J Comput Aided Mol Des, 2007. **21**(5): p. 281-306.
71. Dolinsky, T.J., et al., *PDB2PQR: an automated pipeline for the setup of Poisson-Boltzmann electrostatics calculations.* Nucleic Acids Res, 2004. **32**(Web Server issue): p. W665-7.
72. Word, J.M., et al., *Asparagine and glutamine: using hydrogen atom contacts in the choice of side-chain amide orientation.* J Mol Biol, 1999. **285**(4): p. 1735-47.

73. Guha, R., et al., *The Blue Obelisk-interoperability in chemical informatics.* J Chem Inf Model, 2006. **46**(3): p. 991-8.
74. Schuttelkopf, A.W. and D.M. van Aalten, *PRODRG: a tool for high-throughput crystallography of protein-ligand complexes.* Acta Crystallogr D Biol Crystallogr, 2004. **60**(Pt 8): p. 1355-63.
75. Hendlich, M.R., F.; Barnickel, G., *BALI: Automatic Assignment of Bond and Atom Types for Protein Ligands in the Brookhaven Protein Databank.* Journal of Chemical Information and Computer Sciences., 1997. **37**: p. 774-778.
76. *SYBYL*, Tripos International: 1699 South Hanley Rd., St. Louis, Missouri, 63144, USA.
77. Wang, R., et al., *The PDBbind database: methodologies and updates.* J Med Chem, 2005. **48**(12): p. 4111-9.
78. *Tripos Mol2 file format*, Tripos International: 1699 South Hanley Rd., St. Louis, Missouri, 63144, USA.
79. Sayle, R.A. and E.J. Milner-White, *RASMOL: biomolecular graphics for all.* Trends Biochem Sci, 1995. **20**(9): p. 374.
80. Pettersen, E.F., et al., *UCSF Chimera--a visualization system for exploratory research and analysis.* J Comput Chem, 2004. **25**(13): p. 1605-12.
81. Baber, J.C.H., E. E., *Automatic Assignment of Chemical Connectivity to Organic Molecules in the Cambridge Structure Database.* J. Chem. Inf. Comput. Sci., 1992. **32**: p. 401-406.
82. Jain, A.N., *Morphological similarity: a 3D molecular similarity method correlated with protein-ligand recognition.* J Comput Aided Mol Des, 2000. **14**(2): p. 199-213.
83. Pham, T.A. and A.N. Jain, *Parameter estimation for scoring protein-ligand interactions using negative training data.* J Med Chem, 2006. **49**(20): p. 5856-68.
84. Welch, W., J. Ruppert, and A.N. Jain, *Hammerhead: fast, fully automated docking of flexible ligands to protein binding sites.* Chem Biol, 1996. **3**(6): p. 449-62.
85. Jain, A.N., *Ligand-based structural hypotheses for virtual screening.* J Med Chem, 2004. **47**(4): p. 947-61.
86. Perkins, E., et al., *Novel inhibitors of poly(ADP-ribose) polymerase/PARP1 and PARP2 identified using a cell-based screen in yeast.* Cancer Res, 2001. **61**(10): p. 4175-83.
87. Doman, T.N., et al., *Molecular docking and high-throughput screening for novel inhibitors of protein tyrosine phosphatase-1B.* J Med Chem, 2002. **45**(11): p. 2213-21.
88. Mount, J., et al., *IcePick: a flexible surface-based system for molecular diversity.* J Med Chem, 1999. **42**(1): p. 60-6.
89. Jain, A.N., N.L. Harris, and J.Y. Park, *Quantitative binding site model generation: compass applied to multiple chemotypes targeting the 5-HT1A receptor.* J Med Chem, 1995. **38**(8): p. 1295-308.
90. Jain, A.N., K. Koile, and D. Chapman, *Compass: predicting biological activities from molecular surface properties. Performance comparisons on a steroid benchmark.* J Med Chem, 1994. **37**(15): p. 2315-27.

91. Wei, B.Q., et al., *Testing a flexible-receptor docking algorithm in a model binding site.* J Mol Biol, 2004. **337**(5): p. 1161-82.
92. Pham, T.A.J., A. N., *Customizing Scoring Functions for Molecular Docking.* Journal of Computer-Aided Molecular Design, 2007 (in press).
93. Wang, R.L., L.; Lai, L.; Tang, Y., *SCORE: A new empirical method for estimating the binding affinity of a protein-ligand complex.* Journal of Molecular Modeling, 1998. **4**: p. 379-384.
94. Huang, N., B.K. Shoichet, and J.J. Irwin, *Benchmarking sets for molecular docking.* J Med Chem, 2006. **49**(23): p. 6789-801.
95. Irwin, J.J. and B.K. Shoichet, *ZINC--a free database of commercially available compounds for virtual screening.* J Chem Inf Model, 2005. **45**(1): p. 177-82.
96. Sham, H.L., et al., *A novel, picomolar inhibitor of human immunodeficiency virus type 1 protease.* J Med Chem, 1996. **39**(2): p. 392-7.
97. Wlodawer, A. and J. Vondrasek, *Inhibitors of HIV-1 protease: a major success of structure-assisted drug design.* Annu Rev Biophys Biomol Struct, 1998. **27**: p. 249-84.
98. Axelsen, P.H., et al., *Structure and dynamics of the active site gorge of acetylcholinesterase: synergistic use of molecular dynamics simulation and X-ray crystallography.* Protein Sci, 1994. **3**(2): p. 188-97.
99. Silman, I., et al., *A preliminary comparison of structural models for catalytic intermediates of acetylcholinesterase.* Chem Biol Interact, 1999. **119-120**: p. 43-52.
100. Dvir, H., et al., *3D structure of Torpedo californica acetylcholinesterase complexed with huprine X at 2.1 A resolution: kinetic and molecular dynamic correlates.* Biochemistry, 2002. **41**(9): p. 2970-81.
101. Wong, D.M., et al., *Acetylcholinesterase complexed with bivalent ligands related to huperzine a: experimental evidence for species-dependent protein-ligand complementarity.* J Am Chem Soc, 2003. **125**(2): p. 363-73.
102. Warren, G.L., et al., *A critical assessment of docking programs and scoring functions.* J Med Chem, 2006. **49**(20): p. 5912-31.

Appendix A. Pdbgrind

A.1.1. Usage

This section will detail the usage of pdbgrind on the command line. Its format will be as follows:

Brief command description

General usage: **pdbgrind <command>** `args (optional outfile prefix)`
Example command

Convert a PDB file to MOL2.

```
pdbgrind PDB_file (outfile)
pdbgrind 1STP.pdb protein
```

Convert a PDB file to MOL2, but infer bond connectivity for the molecule using distance thresholds rather than residue information.

```
pdbgrind ligand PDB_file (outfile)
pdbgrind ligand 1STP.pdb protein
```

Retrieve `molecule` *information: number of protein atoms and bonds; number of ligands; number of cofactors; number of waters. Will also perform conversion of PDB file to MOL2.*

```
pdbgrind info molecule (outfile)
pdbgrind info 1STP.pdb protein
```

Trim the protein `molecule` *to include only atoms within a sphere of interest centered on the* `molecule` *centroid. Residues are kept intact.* `Radius` *is given in Angstroms.*

```
pdbgrind trim center radius molecule (outfile)
pdbgrind trim center 15 protein.mol2 protein_trim
```

Trim the protein located in a `PDB_file` *to include only atoms within a sphere of interest centered on a specific ligand's centroid. Residues are kept intact.* `Radius` *is given in Angstroms. The specific* `ligand_num` *can be found upon conversion of the PDB file in the output ligand filenames.*

```
pdbgrind trim ligand ligand_num radius PDB_file (outfile)
pdbgrind trim ligand 1 15 protein.mol2 protein_trim
```

Trim the protein `molecule` *to include only atoms within a sphere of interest centered on the centroid of a ligand found in* `ligand_file`. *Residues are kept intact.* `Radius` *is given in Angstroms.*

```
pdbgrind trim ligand ligand_file radius molecule (outfile)
pdbgrind trim ligand ligand_1.mol2 15 protein.mol2 protein_trim
```

Trim the protein `molecule` *to include only atoms within a sphere of interest centered on a specific point* `(x, y, z)`. *Residues are kept intact.* `Radius` *is given in Angstroms.*

```
pdbgrind trim x y z radius molecule (outfile)
pdbgrind trim 1.0 0.5 2.0 15 protein.mol2 protein_trim
```

Transform the existing molecule orientation found in a `PDBfile` *using a transformation matrix. PDB conversion to MOL2 is optional. A separate* `chain` *ID can be specified for the transformed molecule. The transformation matrix* `xform_matrix` *has the following form:*

```
   rotation matrix
   __|__
 [ a b c x ]
 [ d e f y ]  User-defined matrix
 [ g h i z ]
         |____translation vector
```

```
pdbgrind matrix xform_matrix convert=(0|1) chain PDBfile (outfile)
pdbgrind matrix transform.matrix 0 B 1STP.pdb 1STP-B
```

Transform a `molecule` *orientation using both a transformation and a scale matrix. The transformation matrix* `xform_m` *operates in the crystallographic coordinate frame. The scale matrix* `scale_m` *transforms the PDB deposited orthogonal coordinates into the crystallographic coordinate frame. The format of both matrices is shown above. Passing '0' (zero) as the* `scale_m` *argument to this command will use the scale matrix embedded in the PDB file, if found. A separate* `chain` *ID can be specified for the transformed molecule.*

```
pdbgrind matrix scale xform_m scale_m chain molecule (out)
pdbgrind matrix scale xform.m scale.m B 1STP.pdb 1STP-B
```

Generate the transformation matrix that transforms `monomer1` *into* `monomer2`.

```
pdbgrind getmatrix monomer1 monomer2 matrixName
pdbgrind getmatrix 1STP-A.mol2 1STP-B.mol2 xform.m
```

Merge multiple molecules together into one MOL2 `outfile`.

```
pdbgrind merge file1 file2 ... fileN outfile
pdbgrind merge protein.mol2 ligand1.mol2 ligand2.mol2 complex
```

Calculate the minimum distance between the protein embedded in the `PDB_file` *and the first parsed ligand.*

```
pdbgrind mindist PDB_file
pdbgrind mindist 1STP.pdb
```

Calculate the minimum distance between a `protein` *and* `ligand`.

```
pdbgrind mindist protein ligand
pdbgrind mindist protein.mol2 ligand.mol2
```

Optimize flexible proton rotamers (hydroxyls, thiols) and histidine tautomers for maximal steric and polar interaction within the active site for both the `protein` *and the* `ligand`*. Will also output the sampled proton orientations.*

```
pdbgrind opt_protons protein ligand (prot_outfile) (lig_outfile)
pdbgrind opt_protons protein.mol2 ligand.mol2 protein_opt lig_opt
```

Remove all hydrogens from the given `molecule`*.*

```
pdbgrind strip-h molecule (filename)
pdbgrind strip-h ligand.mol2 ligand_deprot
```

Force a bond to have a specific `bond_order`*. This is useful for fixing a molecule with incorrect protonation. The* `bond_number` *can be located within the molecule's* `MOL2_file`*.*

```
pdbgrind coerce MOL2_file bond_number bond_order filename
pdbgrind coerce ligand.mol2 11 2 ligand_bond_fixed
```

Check if two molecules are graph isomorphs of one another.

```
pdbgrind same ligand1 ligand2
pdbgrind same ligand1.mol2 ligand2.mol2
```

Calculate the centroid distance between two molecules.

```
pdbgrind cent_dist ligand1 ligand2
pdbgrind cent_dist ligand1.mol2 ligand2.mol2
```

A.1.2. Code Documentation

Contents

Chapter 1

pdbgrind Data Structure Index

1.1 pdbgrind Data Structures

Here are the data structures with brief descriptions:

Chapter 2

pdbgrind File Index

2.1 pdbgrind File List

Here is a list of all files with brief descriptions:

Chapter 3

pdbgrind Data Structure Documentation

3.1 AtomMiscData Struct Reference

`#include <pdbgrind-types.h>`

3.1.1 Detailed Description

Stores additional atom information necessary for pdbgrind.

Data Fields

- int num
  Atom index in mol->atoms[] and mol->conformers->atom[].
- char chain [4]
  Chain identifier, useful for multimers.
- int aromatic
  Identifies this atom as part of an aromatic bond.
- double score
  Individual atom score.
- Vector3 alternate

Alternate proton location for a histidine tautomer.

- int histinfo [8]

  Atoms/bonds of interest when scoring HIS tautomers.

3.1.2 Field Documentation

3.1.2.1 int AtomMiscData::num

Atom index in mol->atoms[] and mol->conformers->atom[].

3.1.2.2 char AtomMiscData::chain[4]

Chain identifier, useful for multimers.

3.1.2.3 int AtomMiscData::aromatic

Identifies this atom as part of an aromatic bond.

3.1.2.4 double AtomMiscData::score

Individual atom score.

3.1.2.5 Vector3 AtomMiscData::alternate

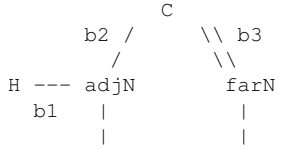
Alternate proton location for a histidine tautomer.

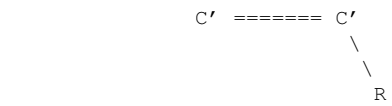
3.1.2.6 int AtomMiscData::histinfo[8]

Atoms/bonds of interest when scoring HIS tautomers.

See also:

setHISInfo()






```
index name    info
-------------------------------------------
0 adjN index into molecule->atoms[]
1 C index into molecule->atoms[]
2 farN    index into molecule->atoms[]
3 adjNLoc index into molecule->atoms[C].connected_atoms[]
4 farNLoc index into molecule->atoms[C].connected_atoms[]
5 b2 bond index into molecule->connections[]
6 b3 bond index into molecule->connections[]
7 b1 bond index into molecule->connections[]
```

The documentation for this struct was generated from the following file:

- pdbgrind-types.h

3.2 Complex Struct Reference

`#include <pdbgrind-types.h>`

3.2.1 Detailed Description

Stores parsed information from a pdb file.

Data Fields

- Molecule * protein
  Protein, can be multimer.
- Molecule ** ligands
  Non-covalent ligands.
- int numLigands
  Number of ligands parsed.
- Molecule ** cofactors
  Single atom cofactors.
- int numCofactors
  Number of cofactors.
- Molecule * water
  Included waters (oxygen location only).

3.2.2 Field Documentation

3.2.2.1 Molecule* Complex::protein

Protein, can be multimer.

3.2.2.2 Molecule** Complex::ligands

Non-covalent ligands.

3.2.2.3 int Complex::numLigands

Number of ligands parsed.

3.2.2.4 Molecule** Complex::cofactors

Single atom cofactors.

3.2.2.5 int Complex::numCofactors

Number of cofactors.

3.2.2.6 Molecule* Complex::water

Included waters (oxygen location only).

The documentation for this struct was generated from the following file:

- pdbgrind-types.h

3.3 LinkList Struct Reference

`#include <linklist.h>`

3.3.1 Detailed Description

Stores information pertaining to a LinkList.

Data Fields

- Node * head
  Head of the list.
- Node * tail
  Tail of the list.
- int len
  Current length.

3.3.2 Field Documentation

3.3.2.1 Node* LinkList::head

Head of the list.

3.3.2.2 Node* LinkList::tail

Tail of the list.

3.3.2.3 int LinkList::len

Current length.

The documentation for this struct was generated from the following file:

- linklist.h

3.4 MoleculeMiscData Struct Reference

`#include <pdbgrind-types.h>`

3.4.1 Detailed Description

Stores additional molecule information necessary for pdbgrind.

Data Fields

- int nres

  Number of residues.

3.4.2 Field Documentation

3.4.2.1 int MoleculeMiscData::nres

Number of residues.

The documentation for this struct was generated from the following file:

- pdbgrind-types.h

3.5 Node_struct Struct Reference

`#include <linklist.h>`

3.5.1 Detailed Description

Stores information pertaining to a node.

Data Fields

- struct Node_struct * next
  Pointer to next node in list.
- void * data
  Data stored within this node.

3.5.2 Field Documentation

3.5.2.1 struct Node_struct* Node_struct::next [read]

Pointer to next node in list.

3.5.2.2 void* Node_struct::data

Data stored within this node.

The documentation for this struct was generated from the following file:

- linklist.h

Chapter 4

pdbgrind File Documentation

4.1 linklist.c File Reference

4.1.1 Detailed Description

Linked list code.

```
#include "stdio.h"
#include "stdlib.h"
#include "linklist.h"
```

Functions

- LinkList * newLinkList (void *data)
  Allocates memory for linked list structure.
- void freeLinkList (LinkList *ll, void(*freeFn)(void *))
  Free memory associated with LinkList.
- void * pop (LinkList *ll)
  Remove first node in linked list.
- void pushQ (LinkList *ll, void *data)
  Push a node onto LinkList Queue-FIFO-style (in the back).
- void pushS (LinkList *ll, void *data)
  Push a node onto LinkList Stack-LIFO-style (in the front).

4.1.2 Function Documentation

4.1.2.1 void freeLinkList (LinkList * *ll*, void(*)(void *) *freeFn*)

Free memory associated with LinkList.

Needs a user defined free function to free user data stored in linked list.

Parameters:

ll A LinkList
freeFn A user defined free function

4.1.2.2 LinkList* newLinkList (void * *data*)

Allocates memory for linked list structure.

Can be passed initial data element.

Parameters:

data Head node data (optional)

Returns:

Pointer to newly allocated LinkList (empty if no data is passed)

4.1.2.3 void* pop (LinkList * *ll*)

Remove first node in linked list.

Parameters:

ll A LinkList

Returns:

Data stored in the head node

4.1.2.4 void pushQ (LinkList * *ll*, void * *data*)

Push a node onto LinkList Queue-FIFO-style (in the back).

Parameters:

ll A LinkList
data User data

4.1.2.5 void pushS (LinkList * *ll*, void * *data*)

Push a node onto LinkList Stack-LIFO-style (in the front).

Parameters:

ll A LinkList

data User data

4.2 linklist.h File Reference

4.2.1 Detailed Description

Linked list public interface.

Data Structures

- struct Node_struct

  Stores information pertaining to a node.

- struct LinkList

  Stores information pertaining to a LinkList.

Typedefs

- typedef struct Node_struct Node

Functions

- LinkList * newLinkList (void *data)

  Allocates memory for linked list structure.

- void freeLinkList (LinkList *ll, void(*freeFn)(void *))

  Free memory associated with LinkList.

- void * pop (LinkList *ll)

  Remove first node in linked list.

- void pushQ (LinkList *ll, void *data)

  Push a node onto LinkList Queue-FIFO-style (in the back).

- void pushS (LinkList *ll, void *data)

  Push a node onto LinkList Stack-LIFO-style (in the front).

4.2.2 Typedef Documentation

4.2.2.1 typedef struct Node_struct Node

4.2.3 Function Documentation

4.2.3.1 void freeLinkList (LinkList * *ll*, void(*)(void *) *freeFn*)

Free memory associated with LinkList.

Needs a user defined free function to free user data stored in linked list.

Parameters:

ll A LinkList

freeFn A user defined free function

4.2.3.2 LinkList* newLinkList (void * *data*)

Allocates memory for linked list structure.

Can be passed initial data element.

Parameters:

data Head node data (optional)

Returns:

Pointer to newly allocated LinkList (empty if no data is passed)

4.2.3.3 void* pop (LinkList * *ll*)

Remove first node in linked list.

Parameters:

ll A LinkList

Returns:

Data stored in the head node

4.2.3.4 void pushQ (LinkList * *ll*, void * *data*)

Push a node onto LinkList Queue-FIFO-style (in the back).

Parameters:

ll A LinkList

data User data

4.2.3.5 void pushS (LinkList * *ll*, void * *data*)

Push a node onto LinkList Stack-LIFO-style (in the front).

Parameters:

ll A LinkList

data User data

4.3 pdbgrind-main.c File Reference

4.3.1 Detailed Description

Command line entry point into pdbgrind.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "utils.h"
#include "pdbgrind.h"
```

Functions

- void printLicense ()

  Print license.

- void doTrimCmd (int argc, char **argv, int i)

  Perform trim command.

- void doMatrixCmd (int argc, char **argv, int i)

  Perform matrix operations.

- void doMergeCmd (int argc, char **argv)

  Merging multiple molecules together into one big molecule.

- void doOptProtons (int argc, char **argv, int i)

  Perform opt_protons operations.

- void doCopyElem (int argc, char **argv, int i)
- void printMinDist (Complex *comp, int argc, char **argv)

  Find the minimum distance b/w protein and ligand.

- int matrixCmdHelp (int argc, char **argv, int matrixScale, int scaleCmd, int inIndex)

  Not enough args for "matrix" command, return error.

- void printHelp (char *msg)

  Command line help and usage.

- int main (int argc, char **argv)

  Command line argument handler.

4.3.2 Function Documentation

4.3.2.1 void doCopyElem (int *argc*, char ** *argv*, int *i*)

4.3.2.2 void doMatrixCmd (int *argc*, char ** *argv*, int *i*)

Perform matrix operations.

Parameters:

argc Number of arguments

argv Array of arguments

i Index of command "matrix"

4.3.2.3 void doMergeCmd (int *argc*, char ** *argv*)

Merging multiple molecules together into one big molecule.

Parameters:

argc Number of arguments

argv Array of arguments

4.3.2.4 void doOptProtons (int *argc*, char ** *argv*, int *i*)

Perform opt_protons operations.

Parameters:

argc Number of arguments

argv Array of arguments

i Index of command "opt_protons"

4.3.2.5 void doTrimCmd (int *argc*, char ** *argv*, int *i*)

Perform trim command.

Parameters:

argc Number of arguments

argv Array of arguments

i Index of command "trim"

4.3.2.6 int main (int *argc*, char ** *argv*)

Command line argument handler.

Parameters:

argc Number of arguments

argv Array of arguments

Returns:

Exit code

4.3.2.7 int matrixCmdHelp (int *argc*, char ** *argv*, int *matrixScale*, int *scaleCmd*, int *inIndex*)

Not enough args for "matrix" command, return error.

Parameters:

argc Number of arguments

argv Array of arguments

matrixScale If true, processing a "matrix scale" command

scaleCmd If passed 0, parse scale from PDB file

inIndex Index into argv[] of PDB file

4.3.2.8 void printHelp (char * *msg*)

Command line help and usage.

Parameters:

msg Specific error message to output

4.3.2.9 void printLicense ()

Print license.

4.3.2.10 void printMinDist (Complex * *comp*, int *argc*, char ** *argv*)

Find the minimum distance b/w protein and ligand.

Parameters:

comp Complex

argc Number of arguments

argv Array of arguments

4.4 pdbgrind-types.h File Reference

4.4.1 Detailed Description

pdbgrind data structures.

Data Structures

- struct AtomMiscData

  Stores additional atom information necessary for pdbgrind.

- struct MoleculeMiscData

  Stores additional molecule information necessary for pdbgrind.

- struct Complex

  Stores parsed information from a pdb file.

Defines

- #define STERIC 0

  Atom type.

- #define ACCEPTOR 1

  Atom type.

- #define DONOR 2

  Atom type.

- #define PI 3.14159265

  Useful value.

- #define BIG 1000000

  Useful value.

- #define SMALL -1000000

  Useful value.

- #define PROTEIN_SIZE 0.15

  Threshold size of largest complex component that defines inclusion as part of protein.

- #define PLANAR_RING_SET_SIZE 5

  Number of interconnected atoms to test for planarity.

- #define BUMP_THRESH -0.33

  Threshold below which bump smoothing may occur (bump penalty is negative).

- #define BUMP_MARK 100

  Atom marked for bump smoothing, stored in atoms[N].close[0].

- #define MAX_OPT_PROTON_EPOCHS 360

  Number of epochs to attempt proton optimization.

- #define MAX_DONORS 128

  Maximum number of protons to optimize on either the protein or ligand.

- #define HYPO_PROTON_THRESH 0.53

  Threshold for proton polar interaction (pdb: 1b42, originally 0.6).

- #define PROTON_MARK 200

  Atom marked for protonation.

- #define COPLANAR_THRESH 0.6

  Threshold for coplanarity.

- #define COLLINEAR_THRESH 0.44

  Threshold for collinearity (pdb: 1it6, originally 0.4).

4.4.2 Define Documentation

4.4.2.1 #define ACCEPTOR 1

Atom type.

4.4.2.2 #define BIG 1000000

Useful value.

4.4.2.3 #define BUMP_MARK 100

Atom marked for bump smoothing, stored in atoms[N].close[0].

4.4.2.4 #define BUMP_THRESH -0.33

Threshold below which bump smoothing may occur (bump penalty is negative).

4.4.2.5 #define COLLINEAR_THRESH 0.44

Threshold for collinearity (pdb: 1it6, originally 0.4).

4.4.2.6 #define COPLANAR_THRESH 0.6

Threshold for coplanarity.

4.4.2.7 #define DONOR 2

Atom type.

4.4.2.8 #define HYPO_PROTON_THRESH 0.53

Threshold for proton polar interaction (pdb: 1b42, originally 0.6).

4.4.2.9 #define MAX_DONORS 128

Maximum number of protons to optimize on either the protein or ligand.

4.4.2.10 #define MAX_OPT_PROTON_EPOCHS 360

Number of epochs to attempt proton optimization.

4.4.2.11 #define PI 3.14159265

Useful value.

4.4.2.12 #define PLANAR_RING_SET_SIZE 5

Number of interconnected atoms to test for planarity.

4.4.2.13 #define PROTEIN_SIZE 0.15

Threshold size of largest complex component that defines inclusion as part of protein.

4.4.2.14 #define PROTON_MARK 200

Atom marked for protonation.

4.4.2.15 #define SMALL -1000000

Useful value.

4.4.2.16 #define STERIC 0

Atom type.

4.5 pdbgrind.c File Reference

4.5.1 Detailed Description

pdbgrind code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <assert.h>
#include "linklist.h"
#include "utils.h"
#include "pdbgrind.h"
```

Defines

- #define MOL "MOLECULE"
- #define ATO "ATOM"
- #define BON "BOND"
- #define RES "RESIDUE"
- #define GRP "GROUP"
- #define MAX_LIG_ATOMS 300
- #define CC_TRIPLE_BOND_LEN 1.20
- #define CC_DOUBLE_BOND_LEN 1.34
- #define CC_SINGLE_BOND_LEN 1.54
- #define CN_TRIPLE_BOND_LEN 1.175

  CN pdb: 1e55.

- #define CN_DOUBLE_BOND_LEN 1.29
- #define CN_SINGLE_BOND_LEN 1.47
- #define CO_TRIPLE_BOND_LEN 1.13
- #define CO_DOUBLE_BOND_LEN 1.20
- #define CO_SINGLE_BOND_LEN 1.43
- #define CS_DOUBLE_BOND_LEN 1.67

  CS pdb:1epp.

- #define CS_SINGLE_BOND_LEN 1.82
- #define N_RESONANT_BOND_LEN 1.35
- #define N_DOUBLE_BOND_LEN 1.26
- #define NN_TRIPLE_BOND_LEN 1.10
- #define NN_DOUBLE_BOND_LEN 1.265
- #define NN_SINGLE_BOND_LEN 1.45
- #define NO_DOUBLE_BOND_LEN 1.21
- #define NO_SINGLE_BOND_LEN 1.40
- #define S_DOUBLE_BOND_LEN 1.55
- #define SS_DOUBLE_BOND_LEN 1.49
- #define SS_SINGLE_BOND_LEN 2.05
- #define SO_DOUBLE_BOND_LEN 1.595

  pdb: 1f4f S=O (1.43)

- #define P_SINGLE_BOND_LEN 1.63
- #define P_DOUBLE_BOND_LEN 1.50

  pdb: 1TLP

- #define PO_DOUBLE_BOND_LEN 1.50
- #define PO_SINGLE_BOND_LEN 1.63
- #define PS_DOUBLE_BOND_LEN 1.86
- #define V_DOUBLE_BOND_LEN 1.76

  pdb: 6RSA (Vandalate)

- #define DOUBLE_BOND_SCORE -1.665
- #define TRIPLE_BOND_SCORE -2.00
- #define AROMATIC 2

  Sentinel value defines bond as aromatic type.

- #define DOUBLE -2

  Sentinel value defines bond as double type.

- #define LONGEST_HBOND 4.0

  Only atoms closer than this threshold can be considered for hbonds.

- #define MAX_BOND_LEN -1.25

  Only atoms closer than this threshold can be considered covalently bonded.

Functions

- Complex * makeComplex (int numLigands, int numCofactors, int numWaters)

  Allocate the Complex structure.

- Molecule * my_read_mol2_file (char *path, char *chainID, int stripH)

  Read a mol2 file.

- Molecule * my_read_mdl_file (char *path, int stripH, int copyElement)

  Handle reading of mdl files.

- Molecule * initStructures (FILE *file)

  Parse everything from file into one single molecule.

- void getResidue (char *line, char *residue, char *chain, int *resNum, char *altLoc)

  Retrieve residue information from ATOM/HETATOM line in a PDB file.

- void parseAtomInfo (FILE *file, Molecule *pdbMol, char *chainID, int stripH)

  Parse all atom info from a file into a single molecule.

- void parsePDBAtom (char *the_line, Conformer *conf, int atom)

  Parse and clean up the pdb atom name field.

- Complex * separateComp (Molecule *pdbMol, char *path)

  Separate the different unconnected components contained within the given Molecule.

- void initComp (Complex *complex, int *natoms_comp, int numDistinct, int max_id, char *path, Molecule *pdbMol)

  From the now distinctly connected components, initialize each component and store it appropriately.

- void processUnboundAtom (Complex *complex, Molecule *pdbMol, int atomNum, int *numCofactors)

  Process any unbound atoms leftover as either water or cofactors.

- void addWater (Molecule *water, Molecule *new)

  Add the well-formed water molecule to our global water structure.

- void grabAtomInfo (Molecule *mol, Molecule *pdbMol, int compID, int *atom_map)

  Assign atoms with the given compID from pdbMol to mol.

- void grabBondInfo (Molecule *mol, Molecule *pdbMol, int compID, int *atom_map)

  Recreate the bond structure for this component.

- void inferBondConnectivity (Molecule *mol)

  Find atoms close enough to form bonds.

- void processProtein (Complex *complex, int ligand)

  Process the protein information.

- void processLigands (Complex *comp)

  Process the ligand information.

- void inferProtBondOrder (Molecule *mol)

  Predict the correct bond order for known residues.

- void bondOrderAA (Molecule *mol, int bondNum)

  Assign bond order based on residue.

- void inferLigandBondOrder (Molecule *mol)

  Assign bond order based on distance.

- int passBondThresh (Molecule *mol, int a1, int a2, int order, double dist)

  Given two atoms and distance between them, determine if the appropriate bond threshold is passed.

- int isResonantBond (Molecule *mol, int a1, int a2, int order, double dist)

  Bond length is in between order and order-1.

- int passDoubleBondThresh (Molecule *mol, int a1, int a2, double dist)

  Thresholding code specific to double bonds.

- int doubleNNBond (Molecule *mol, int a1, int a2, double dist)

  Special case test for N=N.

- int doubleCNBond (Molecule *mol, int a1, int a2, double dist)

  Special case test for C=N.

- int doubleCOBond (Molecule *mol, int a1, int a2, double dist)

  Special case test for C=O.

- int doubleNOBond (Molecule *mol, int a1, int a2, double dist)

Special case test for N=O.

- int doubleSOBond (Molecule *mol, int a1, int a2, double dist)

  Special case test for S=O.

- int doubleSNBond (Molecule *mol, int a1, int a2, double dist)

  Special case test for thiodiimine N=S=N.

- int potentialPhosphoester (Molecule *mol, int a1, int a2, double dist)

  Special case test for potentialPhosphoester ——————.

- void bondOrderAromatic (Molecule *mol, int *arBonds, int numBonds)

  Infer ring bond order from bonds labelled aromatic.

- void labelAromAtoms (Molecule *mol, int bond)

  Given a single bond, locate connected atoms that are part of the aromatic system.

- int findLocalRing (Molecule *mol, int atom, int *localAtoms)

  Given a list of ring atoms from a 'ringSet' which may consist of 1+ rings, extract the local ring that this atom is a part of.

- int cutLocalLooseEnds (Molecule *mol, int *atoms, int nAtom)

  Mark "loose ends" atoms that are connected only to 1 atom (thus not part of local ring).

- void markLocalRing (Molecule *mol, int *atoms, int nAtom)

  Mark atoms that are being traversed as we detect local rings.

- int getNumAromBonds (Molecule *mol, int *atoms, int natom)

  Traverse the local ring atoms counting the number of aromatic bonds.

- int startAromBondRecursion (Molecule *mol, int *bonds, int nbonds, int startBond)

  Entry point into recursive aromatic bond assignment.

- void assignAromBondPlanarPref (Molecule *mol, int *arBonds, int numBonds)

  Assign aromatic bonds with a preference for planar carbons.

- int assignAromBond (Molecule *mol, int bond, int attemptOrder, int priority)

  Recursively assigns alternating pattern of double-single bonds in aromatic rings.

- int retryAssignAromBond (Molecule *mol, int bond, int attemptOrder, int bondOrder, int priority)

  Retry aromatic bond assignment but with reversed bond order.

- int propagateAromBond (Molecule *mol, int atom, int order, int priority)

  Continue propagating aromatic bonds from the given atom.

- void resetAllAromBonds (Molecule *mol)

  Reset all the bond assignments to single.

- void resetAromRing (Molecule *mol, int bond)

  Recursion entry point for resetting the aromatic assignment of a ring.

- void resetAromBond (Molecule *mol, int atom)

  Recursively traverses a ring cycle, resetting bond orders to single and aromatic atoms to unprocessed.

- int nearbyCarbonyl (Molecule *mol, int c)

  Test for a nearby carbonyl during processing of aromatic bonds.

- void checkCarbonyl (Molecule *mol, int a1, int a2)

  Suspected carbonyl bond - break adjacent bonds in a ring that prevent the C=O from forming.

- int checkAdjDoubleBond (Molecule *mol, int at)

  Swap out the adjacent amid double bond to a single bond.

- int checkCarbonylAmidMotif (Molecule *mol, int a1, int a2)

  Atom checks for carbonyl amid motif.

- int checkNearbyCarbonyl (Molecule *mol, int a1, int a2)

  Look for a nearby carbonyl.

- void processPlanarCarbons (Molecule *mol)

  Verify planarity of all carbons.

- int isPlanarSP2Carbon (Molecule *mol, int atom)

  Tests if given atom is a planar sp2 carbon in need of a double bond.

- int checkPlanarSystem (Molecule *mol, int a1, int a2)

  Verify substituents of 2 doubly bonded C are all coplanar.

- int checkPlanarCarbon (Molecule *mol, int a1, int a2)

Special carbon planarity test for use in isValidBond() only.

- int checkCarbonDoubleDouble (Molecule *mol, int a1, int a2)

  Verify collinearity of R = C = R.

- int checkCollinear (Molecule *mol, int a1, int a2)

  General check for collinear atoms.

- int isNitro (Molecule *mol, int N)

  Recognize this:.

- int checkDoubleNitrogen (Molecule *mol, int a1, int a2)
- void write_protein_mol2 (char *path, Complex *comp, FILE *fd, int trim, int knowNumAtoms, int knowNumBonds, int knowNumRes)

  Write out the protein to mol2 format.

- int write_atom_mol2 (FILE *file, Molecule *mol, int currAtom, int trim)

  Write out an atom in mol2 format.

- int write_bond_mol2 (FILE *file, Molecule *mol, int currBond, int trim)

  Write out bond in mol2 format.

- void get_trimmed_protein (Complex *comp, int *numAtom, int *numBond, int *numRes)

  Get number of remaining atoms, bonds, residues after trimming.

- void write_trimmed_protein_mol2 (char *path, Complex *comp)

  Function to call for writing out trimmed proteins.

- void write_substructure_mol2 (FILE *file, Molecule *mol, int trim)

  Write the substructure info (mostly just chainID).

- void my_write_mol2_file (char *path, Conformer *conf, FILE *fd)

  Write a molecule in mol2 format.

- int ringIsCoplanar (Molecule *mol, int *ringAtoms, int len)

  Test for ring coplanarity.

- void markPlanarAtoms (Molecule *mol, int *ringAtoms, int numAtom)

  Grab sets of 5 interconnected atoms and test for planarity.

- void descendPlanar (Molecule *mol, int at, int *set, int index)

  Recursive call does all the work of markPlanarAtoms.

- int isPlanarAtom (Molecule *mol, int at)
  Atom planarity test.
- void getRingSetAtoms (Molecule *mol, int at, int *ringAtoms, int *len)
  Gather up the atoms connected to this ring.
- void label_rings (Molecule *mol, int aromatic)
  Mark bonds that are part of rings.
- int bond_in_arom_ring (Molecule *mol, int bnum)
  Check if a bond is part of an aromatic ring.
- void mark_arom_cycle (Molecule *mol, int at)
  Mark atoms that are part of an aromatic cycle.
- void clear_marks_x (Molecule *mol, int mark)
  Reset all atoms with a given mark.
- int my_atoms_in_ring (Molecule *mol, int a1, int a2, int aromatic)
  Check if two atoms are part of a ring.
- void my_mark_connected_atoms_n (Molecule *mol, int at, int nmark)
  Mark connected atoms with a given integer.
- void sybylAtom (Molecule *mol, int atom, char *sybyl)
  Assign correct sybyl atom type to the given atom.
- void sybylC (Molecule *mol, int atom, char *sybyl)
  Sybyl atom types for carbon.
- void sybylO (Molecule *mol, int atom, char *sybyl)
  Sybyl atom types for oxygen.
- void sybylN (Molecule *mol, int atom, char *sybyl)
  Sybyl atom types for nitrogen.
- void sybylS (Molecule *mol, int atom, char *sybyl)
  Sybyl atom types for sulfur.
- void sybylP (Molecule *mol, int atom, char *sybyl)
  Sybyl atom types for phosphorus.

- int carboxylateO (Molecule *mol, int atom)

  Motif finder: carboxylate from oxygen atom.

- int amideN (Molecule *mol, int atom)

  Motif finder: amide from N.

- int sulfonS (Molecule *mol, int atom)

  Return number of double-bonded O's in a sulfonyl.

- int isAmidine (Molecule *mol, int n, int doubleN)

  Motif finder for amidine:.

- int isAmidine_p (Molecule *mol, int atom)

  Motif finder for amidine when it's protonated:.

- int isThiodiimine (Molecule *mol, int s)

  Motif finder for thiodiimine:.

- int isHydroxamicAcid (Molecule *mol, int n, int o)

  Motif finder for hydroxamic acid:.

- int carbonylAmidMotif (Molecule *mol, int c, int passThresh)

  Motif finder for carbonylAmid:.

- int isSulfonamide (Molecule *mol, int atom, Molecule *newMol)

  Motif finder for sulonamide:.

- Molecule * processMatrix (FILE *file, Molecule *pdbMol, char *matrixFile, char *scaleM)

  Transform the molecule by the given matrices.

- Matrix4x4 * parseMatrix (FILE *file, char *matrixFile, int xform)

  Parse a matrix from a file.

- void writeMatrix (Matrix4x4 *m, FILE *file, char *name)

  Write a matrix to file.

- Molecule * applyTransform (Molecule *pdbMol, Matrix4x4 *xform, Matrix4x4 *o2f)

  Apply matrix tranformation to the molecule.

- int verifyMatrix (Matrix4x4 *m)

Verify validity of transformation matrix.

- void initProtons (Molecule *mol1, int *protons, int numP, Molecule *mol2)

  Initialize proton scores to small number.

- void scoreProtonArray (Molecule *mol1, Molecule *mol2, int *protons, int numP)

  Get scores for all protons in the array.

- int updateProtonScore (Molecule *mol1, int proton, Molecule *mol2, int movement)

  Rescore all protons keeping improved scores.

- double getProtonScore (Molecule *mol1, int proton, Molecule *mol2, int movement, double *bump)

  Score given proton against all nearby acceptor atoms of another molecule.

- double getHypoProtonScore (Molecule *mol1, Vector3 *proton, int central, Molecule *mol2, double *bump)

  Score a hypothetical proton that has not yet added to the molecule.

- double scoreProton (Molecule *mol1, int protAtom, Molecule *mol2, int acceptAtom)

  Score the interaction between a proton and an acceptor based on distance and orientation.

- double scoreHypoPolarPair (Molecule *mol1, Vector3 *proton, double atomRadius, int atomType, Molecule *mol2, int central, int otherAtom)

  Score a hypothetical proton.

- void score_HIS_tautomer (Molecule *mol1, int proton, Molecule *mol2, double *hi)

  Processing a HIS tautomer, score the farN acceptor.

- double compute_atom_bump (Molecule *atomMol, int atom, Molecule *bumpMol)

  Calculate interpenetration score of one atom against all atoms of another molecule.

- double compute_hypo_atom_bump (Molecule *atomMol, Vector3 *atom, double atomRadius, int atomType, int central, Molecule *bumpMol)

  Calculate interpenetration score of a hypothetical atom against all atoms of another molecule.

- void moveProtons (Molecule *mol1, Molecule *mol2, int *protons, int numP, Molecule *testmol, int currIter)

  Randomly move the protons.

- void rotateProton (Molecule *mol, int proton, int click)

  Rotate a proton by a certain number of clicks.

- void swapHISProton (Molecule *mol, int H)

  Swap the H positions on the tautomer by moving the double bond.

- int grabProtons (Molecule *mol1, Molecule *mol2, int *protons)

  Grab interesting flexible protons.

- int isFlexibleProton (Molecule *mol, int atom)

  Test for identification of flexible protons.

- int isAtomClose (Molecule *mol1, int atom, Vector3 *v, Molecule *mol2)

  Test for out atom proximity to any atom of another molecule.

- int IS_HIS (Molecule *mol, int atom, int H)

  Find motif: histidine.

- void setHISInfo (Molecule *mol, int H, int adjN, int C, int farN)

  Set the information about HIS that's needed later in swapHISProton().

- void printProtonScores (Molecule *mol, int *protons, int numP)

  Informational print out of current proton scores.

- int checkHypoProton (Molecule *mol, int a1, int a2)

  Test for possible protonation of a primary amine.

- int checkOptProton (Molecule *mol, int atom)

  Test a hypothetical proton to see if protonating this atom makes sense.

- int protonNearAcceptor (Molecule *mol1, int proton, Vector3 *protLoc, Molecule *mol2, int acceptor)

  Check that the proton-acceptor distance is within the threshold for h-bonds.

- int acceptorNearProton (Molecule *mol1, int acceptor, Molecule *mol2, int proton)

  Check that the acceptor-proton distance is within the threshold for h-bonds Mirror image top protonNearAcceptor().

- void smoothBumps (Complex *comp, Molecule **ligand_noH, Molecule **ligand)

  Redo ligand bond ordering if there are obvious steric clashes due to current protonation state.

- int markBumps (Molecule *mol1, Molecule *mol2, Molecule *markMol)

  Mark the atoms that suffer large bump scores.

- int isGraphISO (Molecule *mol1, Molecule *mol2, double *min_rms, double *min_hrms, int **match, int *nmatch)

  Test to see if two molecules are graph isomorphisms.

- int isoDFS (Molecule *mol1, int a1, Molecule *mol2, int a2, int **match, int *nmatch, int mark, double *min_rms, double *min_hrms)

  Recursion entry point for graph isomorphism search.

- int equivEnviro (Molecule *mol1, int h1, Molecule *mol2, int h2)

  Test that two atoms are in equivalent environments.

- int equivBondOrder (int bo1, int bo2)

  Test for bond order equivalence.

- int getNumHeavy (Molecule *mol, int *nProton)

  Return the number of heavy atoms in molecule.

- void findMismatchedHeavy (Molecule *mol1, Molecule *mol2)

  Simple test for matched number of heavy atoms in two molecules.

- void calcRMS (Molecule *mol1, Molecule *mol2, int **match, int nmatch, double *min_rms, double *min_hrms)

  Given the match array, calculate the RMS deviation b/w the two molecules.

- Molecule * copy_molecule_without_bond (Molecule *source, int bond)

  Copy a molecule minus a specific bond.

- void removeBoundAtom (Molecule *mol, int a1, int a2)

  Remove the bound atom a2 from the connected_atoms[] of a1.

- void center_conformer (Conformer *conf)

  Center the conformer around its centroid.

- void trimMol (Molecule *mol, Vector3 pt, double radius)

  Given a pt and radius, trim any atoms not within the defined sphere of interest.

- Molecule * my_clean_molecule_deprot_acid (Molecule *old_mol)

  Remove hydrogens from a specific motif.

- Molecule * my_protonate_molecule (Molecule *old_mol)

  Protonate a molecule.

- int fixTerminalCarboxyl (Molecule *oldmol, Molecule *newmol, int c)

  Fix terminal carboxyl group of a protein.

- void add_sp2_co (Molecule *mol, int at)

  Add an O to an sp2 carbon.

- int is_planar_N (Molecule *mol, int at)
- int is_resonant_N (Molecule *mol, int at)

  Test that C-N bond is resonant.

- Vector3 calc_sp2_nh (Molecule *mol, int at)

  Calculate location of a hydrogen to connect to an sp2 N.

- void add_sp3_sh_6 (Molecule *mol, int at)

  Add a hydrogen to an S with valency 6 that needs it.

- int V3AllPlanar (Vector3 **vectors, int len)

  Determine if an arbitrary number of points are all coplanar.

- int V3Planar_sp3 (Vector3 *a, Vector3 *b, Vector3 *c, Vector3 *d)

  Test four points for planarity.

- int V3Collinear (Vector3 *v1, Vector3 *v2, Vector3 *v3)

  Determine if the given points are collinear.

- int inv_xform (Matrix4x4 *m, Matrix4x4 *inv)

  Calculate the inverse of the transformation matrix.

- double det (Matrix4x4 *m)

  Return the determinant of the top, left 3x3 corner of the 4x4 matrix.

- void removeH (Molecule *mol, int atom)

  Remove protons bonded to this atom.

- void countRes (Molecule *mol)

Count the number of residues in a molecule.

- void find_nmers (Complex *comp)
  Look for large subunits of a multimer structure.
- void markResidue (Molecule *mol, int atom, char *res, int resNum, int *val)
  Mark the atom with a specific value if it belongs to the given residue.
- int getMolBond (Molecule *mol, int a1, int a2)
  Find the bond.
- int getBoundAtomIndex (Molecule *mol, int fromAtom, int toAtom)
  Find the index of a bond partner in an atom->connected_atoms[].
- void setBondOrderConnectedAtoms (Molecule *mol, int bondNum)
  Change the bond order of a given bond.
- int possibleBond (Molecule *mol, int a1, int a2, int bondOrder)
  Test for potential bond of a given order between two atoms.
- int atomNeedsBond (Molecule *mol, int atom, int numBonds)
  Test if atom has satisfied valency.
- int isValidBond (Molecule *mol, int a1, int a2, int bondOrder)
  Check special cases for bond validity.
- int getAtomDoubleBondIndex (Molecule *mol, int at)
  Get the first bond partner that participates in a double bond.
- double getShortestBond (Molecule *mol, int atom, int *bondAtom)
  Find the length of an atom's shortest bond.
- void my_clean_atom_type (char id[ ])
  Clean the pdb id leaving only the element information.
- int isElement (char *id)
  Check that given string is a recognized element.
- int is_heavy_metal_atom (Molecule *mol, int at)
  Test if an atom is a heavy metal.
- int isWater (Molecule *mol, int atom, int molTest)

Residue name test for water.

- int isValidAABond (Molecule *mol, int a1, int a2)

  Determine if the AA bond we're looking at is valid.

- int weirdElement (char *string, Molecule *mol, int atom)

  Process idiosyncrasies of pdb atom names.

- char checkAmbigAtom (char *str, Molecule *mol, int atom)

  Handle atoms with ambiguous crystallographic positions.

- int countAtomInRes (char *elem, Molecule *mol, int atom)

  Count the number of elems in the residue to which a given atom belongs.

- int isCofactorCation (Molecule *mol, int atom)

  Test for cofactor cation.

- void printContents (char *key, void *data)

  Print to stderr hash contents for a given key.

- int * newInt (int num)

  Return a newly allocated int with value num.

- double myRound (double num, int pow)

  round the number to the given power of 10.

- int findAtom (Molecule *mol1, int atom, Molecule *mol2)

  Find the atom with identical coordinates.

- Molecule * merge_molecules (Molecule *mol1, Molecule *mol2)
- void plotPoint (Molecule *mol, Vector3 *pt, char *name)

  Add a "point" to a molecule.

- int isAA (Molecule *mol, int at)

  Test that an atom is part of a residue with standard amino acid names.

- void copyAtom (Conformer *target, int toAtom, Conformer *from, int fromAtom, int resetBonds)

  Deep copy of atom contents.

- void copyBond (Molecule *toMol, int toBond, Molecule *fromMol, int fromBond, int *atom_map)

  Deep copy of bond information.

- Molecule * copy_molecule_basic (Molecule *old_mol, int atoms_factor, int bonds_factor)

  Does shallow copy of molecule into newly allocated molecule.

- Molecule * copy_molecule_deep (Molecule *source, int atoms_factor, int bonds_factor)

  Perform a deep copy of a molecule.

- int my_is_sp2_atom (Molecule *mol, int at)

  Test for sp2 atom.

- Molecule * my_merge_molecules (Molecule *mol1, Molecule *mol2)

  Given 2 molecules, merge them into 1 molecule.

- int total_bonds (Molecule *mol, int at)
- double real_total_bonds (Molecule *mol, int at)
- void my_add_atom (Molecule *mol, int at, Vector3 *v, char *el)

  Add an atom to the given molecule.

- void my_label_atoms (Molecule *mol)

  Assign all atoms a type: steric/donor/acceptor.

- void my_label_radii (Molecule *mol)

  Assign standard VdW radii to every atom in a molecule.

- int exists (int n, int *arr, int size)

  Check for existence of a number within an array.

- Complex * my_read_molecule_file (char *path, char *chainID, char *m, char *scaleM, int stripH)

  Read a molecule file.

- Complex * my_read_pdb_file (char *path, char *chainID, char *m, char *scaleM, int stripH, int ligand)

  Read a pdb file.

- void write_complex_mol2 (Complex *comp, int trim, char *filename)

  Write out the complex in mol2 format.

- void freeComplex (Complex *comp)

  Free the memory associated with the Complex.

- void printComplexInfo (Complex *complex)

  Print out info regarding the complex.

- void mergeComplexes (Complex *to, Complex *from)

  Merge two complexes.

- void sampleHydroxylRotamer (char *file, int atom)

  Given a specific hydroxyl H, sample 360 rotamers, outputting each.

- void coerceMol (Complex *comp, int bond, int order, char *filename)

  Force a bond to have a certain bond order.

- void trimProtein (Complex *complex, Vector3 pt, double radius)

  Given a pt and radius, trim protein atoms not within the defined sphere of interest.

- void getProteinCentroid (Complex *comp, Vector3 *pt)

  Find centroid for a complex.

- void getLigandCentroid (Complex *comp, Vector3 *pt, char *ligFile)

  Find the appropriate ligand centroid.

- double findMinDist (Molecule *a, Molecule *b)

  Return the minimum distance between two molecules.

- void getMolMatrix (Molecule *a, Molecule *b, char *matrixName)

  Given 2 monomers in different alignments, retrieve the matrix which transforms a into b.

- void optimizeProtons (Molecule *protein, Molecule **ligand, Molecule **ligand_noH)

  Optimize the interactions of flexible protons of the protein-ligand complex.

- void computeSame (Molecule *mol1, Molecule *mol2)

  Compute the rms similarity of 2 molecules if they are of the same constituency.

- void centroidDist (Molecule *mol1, Molecule *mol2)

  Calculate the distance between the centroids of two molecules.

- Molecule * my_make_molecule (int natoms, int nbonds)

  Make a molecule with the given amount of storage.

- Molecule * my_free_molecule (Molecule *mol)

  Free memory allocated to a Molecule.

Variables

- int _copyElement

  Maintain element id parsed from file.

- char _crazyAtom [8]

  Report pdb element ids not currently captured.

- int _crazyFlag = 0

  True if we've parsed a pdb element id never seen before.

- int _opt = 0

  Begin testing for hypothetical protons.

- Molecule * _optProt

  Spare pointer to protein, don't use if _opt is off.

4.5.2 Define Documentation

4.5.2.1 #define AROMATIC 2

Sentinel value defines bond as aromatic type.

4.5.2.2 #define ATO "ATOM"

4.5.2.3 #define BON "BOND"

4.5.2.4 #define CC_DOUBLE_BOND_LEN 1.34

4.5.2.5 #define CC_SINGLE_BOND_LEN 1.54

4.5.2.6 #define CC_TRIPLE_BOND_LEN 1.20

4.5.2.7 #define CN_DOUBLE_BOND_LEN 1.29

4.5.2.8 #define CN_SINGLE_BOND_LEN 1.47

4.5.2.9 #define CN_TRIPLE_BOND_LEN 1.175

CN pdb: 1e55.

4.5.2.10 #define CO_DOUBLE_BOND_LEN 1.20

4.5.2.11 #define CO_SINGLE_BOND_LEN 1.43

4.5.2.12 #define CO_TRIPLE_BOND_LEN 1.13

4.5.2.13 #define CS_DOUBLE_BOND_LEN 1.67

CS pdb:1epp.

4.5.2.14 #define CS_SINGLE_BOND_LEN 1.82

4.5.2.15 #define DOUBLE -2

Sentinel value defines bond as double type.

4.5.2.16 #define DOUBLE_BOND_SCORE -1.665

4.5.2.17 #define GRP "GROUP"

4.5.2.18 #define LONGEST_HBOND 4.0

Only atoms closer than this threshold can be considered for hbonds.

4.5.2.19 #define MAX_BOND_LEN -1.25

Only atoms closer than this threshold can be considered covalently bonded.

4.5.2.20 #define MAX_LIG_ATOMS 300

4.5.2.21 #define MOL "MOLECULE"

4.5.2.22 #define N_DOUBLE_BOND_LEN 1.26

4.5.2.23 #define N_RESONANT_BOND_LEN 1.35

4.5.2.24 #define NN_DOUBLE_BOND_LEN 1.265

4.5.2.25 #define NN_SINGLE_BOND_LEN 1.45

4.5.2.26 #define NN_TRIPLE_BOND_LEN 1.10

4.5.2.27 #define NO_DOUBLE_BOND_LEN 1.21

4.5.2.28 #define NO_SINGLE_BOND_LEN 1.40

4.5.2.29 #define P_DOUBLE_BOND_LEN 1.50

pdb: 1TLP

4.5.2.30 #define P_SINGLE_BOND_LEN 1.63

4.5.2.31 #define PO_DOUBLE_BOND_LEN 1.50

4.5.2.32 #define PO_SINGLE_BOND_LEN 1.63

4.5.2.33 #define PS_DOUBLE_BOND_LEN 1.86

4.5.2.34 #define RES "RESIDUE"

4.5.2.35 #define S_DOUBLE_BOND_LEN 1.55

4.5.2.36 #define SO_DOUBLE_BOND_LEN 1.595

pdb: 1f4f S=O (1.43)

4.5.2.37 #define SS_DOUBLE_BOND_LEN 1.49

4.5.2.38 #define SS_SINGLE_BOND_LEN 2.05

4.5.2.39 #define TRIPLE_BOND_SCORE -2.00

4.5.2.40 #define V_DOUBLE_BOND_LEN 1.76

pdb: 6RSA (Vandalate)

4.5.3 Function Documentation

4.5.3.1 int acceptorNearProton (Molecule * *mol1*, int *acceptor*, Molecule * *mol2*, int *proton*)

Check that the acceptor-proton distance is within the threshold for h-bonds Mirror image top protonNearAcceptor().

See also:

protonNearAcceptor(). If no proton atom is given, cycle through all atoms of the mol2 to find those proton atoms close to the given acceptor in mol1.

Optional arguments may be passed NULL.

Parameters:

mol1 A Molecule that may be nearby to our proton

acceptor Acceptor index into mol1->atoms[] to test, pass -1 to test all atoms in mol1

mol2 A Molecule with our proton to test

proton Index into mol2->atoms[] of a proton

Returns:

True if passes test

4.5.3.2 void add_sp2_co (Molecule * *mol*, int *at*)

Add an O to an sp2 carbon.

Exactly like add_sp2_ch, but adds an O instead.

Parameters:

mol A Molecule.

at Carbon atom index into molecule->atoms[]

4.5.3.3 void add_sp3_sh_6 (Molecule * *mol*, int *at*)

Add a hydrogen to an S with valency 6 that needs it.

Parameters:

mol A Molecule

at Sulfur index into molecule->atoms[]

4.5.3.4 void addWater (Molecule * *water*, Molecule * *new*)

Add the well-formed water molecule to our global water structure.

Parameters:

water Molecule that holds all the water molecule

new A new water molecule to add

4.5.3.5 int amideN (Molecule * *mol*, int *atom*)

Motif finder: amide from N.

```
     O     R
     ||   /
R - C - N   <--- from N perpective
          \
           R
```

Parameters:

mol A Molecule

atom Index of N atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.6 Molecule * applyTransform (Molecule * *pdbMol*, Matrix4x4 * *xform*, Matrix4x4 * *o2f*)

Apply matrix tranformation to the molecule.

- Copy 2 pdbMol's into new
- Transform the first copy

Called thru processMatrix.

Parameters:

pdbMol A Molecule

xform Transformation matrix

o2f Scale matrix (orthagonal to crystallographic coordinates)

Returns:

Transformed Molecule

4.5.3.7 int assignAromBond (Molecule * *mol*, int *bond*, int *attemptOrder*, int *priority*)

Recursively assigns alternating pattern of double-single bonds in aromatic rings.

Parameters:

mol A Molecule

bond Index of bond into molecule->connections[] to assign

attemptOrder If > 0, Try double bond first

priority If true, will attempt to assign double bond despite valence restriction

Returns:

2 indicates double bond assigned
1 indicates single bond assigned
0 indicates no bond assigned
-1 indicates abort recursion

4.5.3.8 void assignAromBondPlanarPref (Molecule * *mol*, int * *arBonds*, int *numBonds*)

Assign aromatic bonds with a preference for planar carbons.

Parameters:

mol A Molecule

arBonds Array of aromatic bond indices into molecule->connections[]

numBonds Number of bonds in array

4.5.3.9 int atomNeedsBond (Molecule * *mol*, int *atom*, int *numBonds*)

Test if atom has satisfied valency.

ASSUME: atom connectivity already established. ASKS: can I change the bond order of a neighbor to numBonds.

If orbital shell is not satisfied with number of bonds made, return true.

Can now check how many bonds this atom needs.

Parameters:

mol A Molecule

atom Index into molecule->atoms[]

numBonds Additional bonds to add to atom

Returns:

True if passes test.

4.5.3.10 int bond_in_arom_ring (Molecule * *mol*, int *bnum*)

Check if a bond is part of an aromatic ring.

Parameters:

mol A Molecule

bnum Index of bond in molecule->connections[]

Returns:

True if passes test

4.5.3.11 void bondOrderAA (Molecule * *mol*, int *bondNum*)

Assign bond order based on residue.

Amino acids have standard bond connectivities.

Parameters:

mol A protein Molecule

bondNum Index into mol->connections[] to process

4.5.3.12 void bondOrderAromatic (Molecule * *mol*, int * *arBonds*, int *numBonds*)

Infer ring bond order from bonds labelled aromatic.

Systematically attempts to find a solution for the ring system by propagating alternating single and double bonds. Arbitrarily starts with a double bond.

Parameters:

mol A Molecule

arBonds Array of aromatic bond indices into molecule->connections[]

numBonds Number of bonds in array

4.5.3.13 Vector3 calc_sp2_nh (Molecule * *mol*, int *at*)

Calculate location of a hydrogen to connect to an sp2 N.

Parameters:

mol A Molecule

at Index into molecule->atoms[]

Returns:

Location of hydrogen

4.5.3.14 void calcRMS (Molecule * *mol1*, Molecule * *mol2*, int ** *match*, int *nmatch*, double * *min_rms*, double * *min_hrms*)

Given the match array, calculate the RMS deviation b/w the two molecules.

Match array generated from isoDFS()

See also:

isoDFS().

Print results to stderr.

Parameters:

mol1 A Molecule

mol2 Another Molecule

match 2xN array of atom matches for both molecules where N is the number of atoms

nmatch Number of matches

min_rms Storage for the minimum RMS

min_hrms Storage for the mininmum heavy atom RMS

Returns:

True if passed test

4.5.3.15 int carbonylAmidMotif (Molecule * *mol*, int *c*, int *passThresh*)

Motif finder for carbonylAmid:.

```
Recognize this:

    R
    |
O - C - N - R
    ^
    checks from here
as

    R
    |
O = C - N - R

So do not assign double bonds to C, N, R from the ring
```

Parameters:

mol A Molecule

c Index of C atom into molecule->atoms[]

passThresh If < 1: ignore bump rule, else if $== 1$: test that we also pass dist thresholds

Returns:

True if passes test

4.5.3.16 int carboxylateO (Molecule * *mol*, int *atom*)

Motif finder: carboxylate from oxygen atom.

Have partial double bonds on oxygen. Shared negative overall charge.

```
example:
      O -0.5
     //
R - C
     \
      O -0.5
```

Parameters:

mol A Molecule

atom Index of O atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.17 void center_conformer (Conformer * *conf*)

Center the conformer around its centroid.

Find centroid. Subtract it from every point.

Parameters:

conf A Conformer

4.5.3.18 void centroidDist (Molecule * *mol1*, Molecule * *mol2*)

Calculate the distance between the centroids of two molecules.

Parameters:

mol1 Molecule A

mol2 Molecule B

4.5.3.19 int checkAdjDoubleBond (Molecule * *mol*, int *at*)

Swap out the adjacent amid double bond to a single bond.

Special case: there is an adjacent double bond from an amid which makes it impossible to place a C=O correctly.

```
     R
     |
HO - C = N - R

Turns into:

    R
    |
O = C - N - R
        |
        R
```

Parameters:

mol A Molecule

at Atom index into molecule->atoms[]

Returns:

True if swap occurred

4.5.3.20 char checkAmbigAtom (char * *str*, Molecule * *mol*, int *atom*)

Handle atoms with ambiguous crystallographic positions.

Those atoms labelled 'A' are ambiguous crystallographically We have to guess atom identity if we know the residue.

The str is what we've parsed as the atom name from the ATOM record. We are currently parsing molecule, index = atom.

This function will change the str to hold the appropriate pdbatom description for the ambiguous atom. ASSUMES: string has space for 3 chars!

This function returns the unambiguated atom (GLN/ASN only).

Parameters:

str PDB id atom string

mol A Molecule

atom Index into mol->atoms[]

Returns:

Disambiguated element for GLN/ASN only, otherwise returns an empty string

4.5.3.21 int checkCarbonDoubleDouble (Molecule * *mol*, int *a1*, int *a2*)

Verify collinearity of R = C = R.

Parameters:

mol A Molecule

a1 Atom index into molecule->atoms[]

a2 Atom index into molecule->atoms[]

Returns:

True if passes test

4.5.3.22 void checkCarbonyl (Molecule * *mol*, int *a1*, int *a2*)

Suspected carbonyl bond - break adjacent bonds in a ring that prevent the C=O from forming.

Maybe redundant: carbonyl check in aromatic bond processing.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

4.5.3.23 int checkCarbonylAmidMotif (Molecule * *mol*, int *a1*, int *a2*)

Atom checks for carbonyl amid motif.

Do not let C=N formed over C=O for this motif.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

Returns:

True if passes test

4.5.3.24 int checkCollinear (Molecule * *mol*, int *a1*, int *a2*)

General check for collinear atoms.

A-a1-a2-B are all collinear.

Parameters:

mol A Molecule

a1 Atom index into molecule->atoms[]

a2 Atom index into molecule->atoms[]

Returns:

True if passes test

4.5.3.25 int checkDoubleNitrogen (Molecule * *mol*, int *a1*, int *a2*)

4.5.3.26 int checkHypoProton (Molecule * *mol*, int *a1*, int *a2*)

Test for possible protonation of a primary amine.

Create a hypothetical proton off primary amine, if there exists a good interaction with the protein, do not let a double bond be formed to this nitrogen so it will be protonated later. Tests turned on by _opt global. This function call useful from bond tests.

Parameters:

mol A Molecule

a1 Atom index into molecule->atoms[], bond partner to a2

a2 Atom index into molecule->atoms[], bond partner to a1

Returns:

True if passes test

4.5.3.27 int checkNearbyCarbonyl (Molecule * *mol*, int *a1*, int *a2*)

Look for a nearby carbonyl.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

Returns:

True if passes test

4.5.3.28 int checkOptProton (Molecule * *mol*, int *atom*)

Test a hypothetical proton to see if protonating this atom makes sense.

For sp2 N atoms only, score its hypothetical proton. If score > HYPO_PROTON_-THRESH, do not assign this N a double bond. Verify that existing bond lengths to this atom are greater than double.

Parameters:

mol A Molecule

atom Index into molecule->atoms[] of N atom

Returns:

True if test is passed

4.5.3.29 int checkPlanarCarbon (Molecule * *mol*, int *a1*, int *a2*)

Special carbon planarity test for use in isValidBond() only.

C=O special cased.

Parameters:

mol A Molecule

a1 carbon atom index into molecule->atoms[]

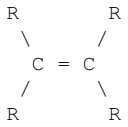
a2 Index into molecule->atoms[] of another atom connected to a1

Returns:

True if passes test

4.5.3.30 int checkPlanarSystem (Molecule * *mol*, int *a1*, int *a2*)

Verify substituents of 2 doubly bonded C are all coplanar.



Parameters:

mol A Molecule

a1 carbon1 atom index into molecule->atoms[]

a2 carbon2 atom index into molecule->atoms[]

Returns:

True if passes test

if (!isPlanarSP2Carbon(mol, a1) || !isPlanarSP2Carbon(mol, a2)) return 0;

4.5.3.31 void clear_marks_x (Molecule * *mol*, int *mark*)

Reset all atoms with a given mark.

Parameters:

mol A Molecule.

mark Mark to give to connected atoms

4.5.3.32 void coerceMol (Complex * *comp*, int *bond*, int *order*, char * *filename*)

Force a bond to have a certain bond order.

Parameters:

comp A Complex (this function operates on the protein)

bond Index of bond in molecule->connections[]

order Bond order desired

filename Output name for new complex

4.5.3.33 double compute_atom_bump (Molecule * *atomMol*, int *atom*, Molecule * *bumpMol*)

Calculate interpenetration score of one atom against all atoms of another molecule.

Parameters:

atomMol A Molecule that contains our atom

atom Index of atom into mol1->atoms[]

bumpMol A 2nd Molecule against which we'll score our atom

Returns:

Bump score

4.5.3.34 double compute_hypo_atom_bump (Molecule * *atomMol*, Vector3 * *atom*, double *atomRadius*, int *atomType*, int *central*, Molecule * *bumpMol*)

Calculate interpenetration score of a hypothetical atom against all atoms of another molecule.

Parameters:

atomMol A Molecule that will contain the hypothetical atom

atom Hypothetical position of an atom

atomRadius Radius of the hypothetical atom

atomType Type of hypothetical atom (DONOR/ACCEPTOR)

central Index of the hypothetical atom's parent atom into mol2->atoms[]

bumpMol A 2nd Molecule against which we'll score our proton

Returns:

Score

4.5.3.35 void computeSame (Molecule * *mol1*, Molecule * *mol2*)

Compute the rms similarity of 2 molecules if they are of the same constituency.

Particular emphasis is on proton composition/positioning as this is primarily a check to see that our bond ordering assignment and opt_protons command are working.

Print results to stderr.

Parameters:

mol1 Molecule A

mol2 Molecule B

4.5.3.36 Molecule * copy_molecule_basic (Molecule * *old_mol*, int *atoms_factor*, int *bonds_factor*)

Does shallow copy of molecule into newly allocated molecule.

Basic information copied over only:

- numAtoms * atoms_factor
- numBonds * bonds_factor
- numRes

Parameters:

old_mol Source molecule to copy

atoms_factor Factor size increase to number of atoms in old molecule

bonds_factor Factor size increase to number of bonds in old molecule

Returns:

Newly allocated Molecule copy

4.5.3.37 Molecule * copy_molecule_deep (Molecule * *source*, int *atoms_factor*, int *bonds_factor*)

Perform a deep copy of a molecule.

Allocates a new molecule 'target', then deep copies the atom and bond information from 'source' => target.

Parameters:

source Source molecule to copy

atoms_factor Factor size increase to number of atoms in old molecule

bonds_factor Factor size increase to number of bonds in old molecule

Returns:

Newly allocated Molecule copy

4.5.3.38 Molecule * copy_molecule_without_bond (Molecule * *source*, int *bond*)

Copy a molecule minus a specific bond.

Helper to coerceMol()

See also:

coerceMol() Allocates a new molecule 'target', then copies the atom and bond information from 'source' to target, minus the given bond.

Parameters:

source Molecule to copy

bond Index of bond in molecule->connections[] to omit

Returns:

Newly allocated copy of molecule minus the bond

4.5.3.39 void copyAtom (Conformer * *target*, int *toAtom*, Conformer * *from*, int *fromAtom*, int *resetBonds*)

Deep copy of atom contents.

resetBonds lets copyBond do the right thing

Parameters:

target A target Conformer

toAtom Target atom index into target->atoms[]

from A source Conformer

fromAtom Source atom index into from->atoms[]

resetBonds Set to 1 if copyBonds() will be used directly after this function

4.5.3.40 void copyBond (Molecule * *toMol*, int *toBond*, Molecule * *fromMol*, int *fromBond*, int * *atom_map*)

Deep copy of bond information.

Atom map argument allows us to remap atom indexes to start from zero. Useful if taking a subset of larger molecule.

Optional arguments may be passed NULL.

Parameters:

toMol Target Molecule

toBond Target bond index into molecule->connections[]

fromMol Source Molecule

fromBond Source bond index into molecule->connections[]

atom_map Optional - map of old atom indices to the new indices

4.5.3.41 int countAtomInRes (char * *elem*, Molecule * *mol*, int *atom*)

Count the number of elems in the residue to which a given atom belongs.

Parameters:

elem An element

mol A Molecule

atom Index into molecule->atoms, member of the residue we're checking

Returns:

Number of elements in the residue

4.5.3.42 void countRes (Molecule * *mol*)

Count the number of residues in a molecule.

Store in mol->miscdata->nres the current residue count. Assumes atoms are arranged in residue order.

Parameters:

mol A Molecule.

4.5.3.43 int cutLocalLooseEnds (Molecule * *mol*, int * *atoms*, int *nAtom*)

Mark "loose ends" atoms that are connected only to 1 atom (thus not part of local ring).

All bad atoms be marked -1.

Intersection atoms (mark > 2) will also be marked -1 so that they can be incorporated into adjoining future rings.

- non-intersection atoms in local ring marked untouchable: close[2] = 1

Parameters:

mol A Molecule

atoms Array of potential local ring atoms; after this call will contain only local ring atoms

nAtom Number of atoms in array

Returns:

Number of local ring atoms

4.5.3.44 void descendPlanar (Molecule * *mol*, int *at*, int * *set*, int *index*)

Recursive call does all the work of markPlanarAtoms.

See also:

markPlanarAtoms()

Parameters:

mol A Molecule
at Atom index into molecule->atoms[] from which we'll begin our descent
set Growing set of planar atoms as we search
index Current size of set

4.5.3.45 double det (Matrix4x4 * *m*)

Return the determinant of the top, left 3x3 corner of the 4x4 matrix.

Ignores translation, w vector.

Based on Rasmol code written by: Herbert J. Bernstein, yaya@bernstein-plus-sons.com, 6 March 1998

Parameters:

m A Matrix4x4

Returns:

Determinant

4.5.3.46 int doubleCNBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for C=N.

Check for amide where carbonyl gets preference. Resonant C-N awarded double iff its C's shortest bond.

Parameters:

mol A Molecule
a1 Atom1 index into molecule->atoms[]
a2 Atom2 index into molecule->atoms[]
dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.47 int doubleCOBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for C=O.

Recognize and correctly assign bond order to these motifs:

```
R      O1 - R ester
 \    /
  C
  ||
  O2

     O1
    /
R - C carboxyl
    \\
     O2

     R
    /
R - C  hydroxyl
    \\
     O
```

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.48 int doubleNNBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for N=N.

Must be at least resonant length. Must be coplanar.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.49 int doubleNOBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for N=O.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.50 int doubleSNBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for thiodiimine N=S=N.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.51 int doubleSOBond (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for S=O.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.52 int equivBondOrder (int *bo1*, int *bo2*)

Test for bond order equivalence.

Handles double/aromatic case.

Parameters:

bo1 Bond order of bond1

bo2 Bond order of bond2

Returns:

True if passes test

4.5.3.53 int equivEnviro (Molecule * *mol1*, int *h1*, Molecule * *mol2*, int *h2*)

Test that two atoms are in equivalent environments.

Given a heavy atom from mol1 and mol2, determine if they are have same bond members.

ASSUMES: the 2 molecules to be perfectly aligned!! Does a test that atoms match roughly by location in space as well. Can be called on hydrogens as well.

Parameters:

mol1 A Molecule

h1 Atom index into mol1->atoms[]

mol2 Another Molecule

h2 Atom index into mol2->atoms[]

Returns:

True if passed test

4.5.3.54 int exists (int *n*, int * *arr*, int *size*)

Check for existence of a number within an array.

Parameters:

n Integer number

arr Array of ints

size Size of array

Returns:

True if n exists in arr

4.5.3.55 void find_nmers (Complex * *comp*)

Look for large subunits of a multimer structure.

Those components that are sufficiently large in size relative to the protein are most likely n-mers of the protein and should not be outputted as ligand.

Parameters:

comp A Complex

4.5.3.56 int findAtom (Molecule * *mol1*, int *atom*, Molecule * *mol2*)

Find the atom with identical coordinates.

Parameters:

mol1 A Molecule

atom Index into mol1->atoms[]

mol2 Another Molecule in which we'll search for an equivalently located atom

Returns:

Index into mol2->atoms[] with identical location to our atom

4.5.3.57 int findLocalRing (Molecule * *mol*, int *atom*, int * *localAtoms*)

Given a list of ring atoms from a 'ringSet' which may consist of 1+ rings, extract the local ring that this atom is a part of.

Local ring = smallest ring containing this atom.

Need to call to clear_marks_x(-1) prior to this call. Remembers parent thru atom->close[1].

Parameters:

mol A Molecule

atom Index of atom into molecule->atoms[]

localAtoms Array of atoms that are a part of this local ring

Returns:

Number of atoms in localAtoms

4.5.3.58 double findMinDist (Molecule * *a*, Molecule * *b*)

Return the minimum distance between two molecules.

Parameters:

a Molecule A

b Molecule B

Returns:

Distance

4.5.3.59 void findMismatchedHeavy (Molecule * *mol1*, Molecule * *mol2*)

Simple test for matched number of heavy atoms in two molecules.

Prints to stderr test results

Parameters:

mol1 A Molecule

mol2 Another Molecule

4.5.3.60 int fixTerminalCarboxyl (Molecule * *oldmol*, Molecule * *newmol*, int *c*)

Fix terminal carboxyl group of a protein.

Some pdb files leave the terminal carboxyl group with a missing O. Note: Correctness of this fix is up for debate.

Parameters:

oldmol A Molecule

newmol Identical molecule that will contain the fixed group

c Index into molecule->atoms[] of the carbon of the carboxyl group

Returns:

True if fix applied

4.5.3.61 void freeComplex (Complex * *comp*)

Free the memory associated with the Complex.

Parameters:

comp A Complex

4.5.3.62 void get_trimmed_protein (Complex * *comp*, int * *numAtom*, int * *numBond*, int * *numRes*)

Get number of remaining atoms, bonds, residues after trimming.

Run a fake pass through write_bond_mol2() to establish number of trimmed atoms and bonds that will be written to file.

Number of trimmed atoms already stored in each molecule

- mol->conformer->data[9] (hack)
- Earlier done in trimMol()
- Same goes for number of res

Parameters:

comp A Complex

numAtom Storage for number of atoms

numBond Storage for number of bonds

numRes Storage for number of residues

4.5.3.63 int getAtomDoubleBondIndex (Molecule * *mol*, int *at*)

Get the first bond partner that participates in a double bond.

Parameters:

mol A Molecule

at Atom index into molecule->atoms[]

Returns:

If found, returns partner index into at.connected_atoms[], else returns -1

4.5.3.64 int getBoundAtomIndex (Molecule * *mol*, int *fromAtom*, int *toAtom*)

Find the index of a bond partner in an atom->connected_atoms[].

Parameters:

mol A Molecule

fromAtom Atom1 index into molecule->atoms[], bond partner to a2

toAtom Atom2 index into molecule->atoms[], bond partner to a1

4.5.3.65 double getHypoProtonScore (Molecule * *mol1*, Vector3 * *proton*, int *central*, Molecule * *mol2*, double * *bump*)

Score a hypothetical proton that has not yet added to the molecule.

Exactly like

See also:

getProtonScore().

Parameters:

mol1 A Molecule with protons to score

proton Hypothetical position of a proton

mol2 A 2nd Molecule against which we'll score our proton

central Proton parent atom index into molecule->atoms[]

bump Storage for bump score (steric clash) of this proton

Returns:

Score

4.5.3.66 void getLigandCentroid (Complex * *comp*, Vector3 * *pt*, char * *ligFile*)

Find the appropriate ligand centroid.

If ligand not given, default to protein centroid.

Optional arguments may be passed NULL.

Parameters:

comp A Complex

pt Storage for the found centroid

ligFile Optional - file containing the ligand

4.5.3.67 int getMolBond (Molecule * *mol*, int *a1*, int *a2*)

Find the bond.

Return the index in the Molecule connections array of the bond between a1 & a2.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

Returns:

Index into molecule->connections[]

4.5.3.68 void getMolMatrix (Molecule * *a*, Molecule * *b*, char * *matrixName*)

Given 2 monomers in different alignments, retrieve the matrix which transforms a into b.

Parameters:

a Molecule A

b Molecule B

matrixName Matrix filename to write out

4.5.3.69 int getNumAromBonds (Molecule * *mol*, int * *atoms*, int *natom*)

Traverse the local ring atoms counting the number of aromatic bonds.

Parameters:

mol A Molecule

atoms Array of potential local ring atoms

natom Number of atoms in array

Returns:

Number of aromatic bonds

4.5.3.70 int getNumHeavy (Molecule * *mol*, int * *nProton*)

Return the number of heavy atoms in molecule.

Also, give the exact number of protons.

Parameters:

mol A Molecule

nProton Storage for number of protons found

Returns:

Number of heavy atoms

4.5.3.71 void getProteinCentroid (Complex * *comp*, Vector3 * *pt*)

Find centroid for a complex.

Assumes all n-mers of the complex are approx same size.

Parameters:

comp A Complex

pt Storage for the found centroid

4.5.3.72 double getProtonScore (Molecule * *mol1*, int *proton*, Molecule * *mol2*, int *movement*, double * *bump*)

Score given proton against all nearby acceptor atoms of another molecule.

Return only the highest score found.

Parameters:

mol1 A Molecule with protons to score

proton Index into molecule->atoms[] of a proton

mol2 A 2nd Molecule against which we'll score our proton

movement If true, protons recently moved, make sure to score both HIS tautomers

bump Storage for bump score (steric clash) of this proton

Returns:

Score

4.5.3.73 void getResidue (char * *line*, char * *residue*, char * *chain*, int * *resNum*, char * *altLoc*)

Retrieve residue information from ATOM/HETATOM line in a PDB file.

Based on PDB file specification:

http://www.wwpdb.org/documentation/format23/sect9.html

Parameters:

line ATOM line

residue Allocated string buffer to store residue name

chain Allocated string buffer to store chain id

resNum Storage for the residue number of this atom

altLoc Allocated string buffer to store alternate locations for the atom, often a letter placed directly adjacent to the element

4.5.3.74 void getRingSetAtoms (Molecule * *mol*, int *at*, int * *ringAtoms*, int * *len*)

Gather up the atoms connected to this ring.

Traverse all ring bonds connected to the given one, saving ring atoms along the way in the passed in array.

Parameters:

mol A Molecule

at Index of atom into molecule->atoms[] to start ring set search

ringAtoms Growing array of ring atoms as we search

len Current length of ringAtoms

4.5.3.75 double getShortestBond (Molecule * *mol*, int *atom*, int * *bondAtom*)

Find the length of an atom's shortest bond.

Parameters:

mol A Molecule

atom Atom index into molecule->atoms[]

bondAtom Storage for the bond partner index into molecule->atoms[]

Returns:

Distance, if this atom doesn't have any bond partners return 100000

4.5.3.76 void grabAtomInfo (Molecule * *mol*, Molecule * *pdbMol*, int *compID*, int * *atom_map*)

Assign atoms with the given compID from pdbMol to mol.

Parameters:

mol Target molecule

pdbMol Source molecule

compID Unique component ID

atom_map Atom map necessary to find the appropriate bonds in pdbMol for copying

4.5.3.77 void grabBondInfo (Molecule * *mol*, Molecule * *pdbMol*, int *compID*, int * *atom_map*)

Recreate the bond structure for this component.

See also:

grabAtomInfo

Parameters:

mol Target molecule

pdbMol Source molecule

compID Unique component ID

atom_map Atom map necessary to find the appropriate bonds in pdbMol for copying

4.5.3.78 int grabProtons (Molecule * *mol1*, Molecule * *mol2*, int * *protons*)

Grab interesting flexible protons.

Save in a pre-allocated array the atom indices which represent flexible protons on mol1 that are close to acceptors on mol2.

Parameters:

mol1 A Molecule that contains protons

mol2 A 2nd Molecule against which we'll score our protons

protons Allocated array to store interesting protons

Returns:

Number of protons found

4.5.3.79 void inferBondConnectivity (Molecule * *mol*)

Find atoms close enough to form bonds.

If atoms are close enough, they share a bond.

Parameters:

mol Molecule that needs bond detection and creation

4.5.3.80 void inferLigandBondOrder (Molecule * *mol*)

Assign bond order based on distance.

When assigning bond order:

- both atoms BondOrder()
- bond itself MolBondOrder()

Parameters:

mol A ligand Molecule

4.5.3.81 void inferProtBondOrder (Molecule * *mol*)

Predict the correct bond order for known residues.

Assumes:

- atoms in molecule are already connected
- pdb file uses standard 20 names for amino acids

Parameters:

mol A protein Molecule

4.5.3.82 void initComp (Complex * *complex*, int * *natoms_comp*, int *numDistinct*, int *max_id*, char * *path*, Molecule * *pdbMol*)

From the now distinctly connected components, initialize each component and store it appropriately.

Called from separateComp()

Parameters:

complex A Complex of different components

natoms_comp Array containing the number of atoms in each component

numDistinct Number of distinct components

max_id ID of the largest component

path Name to prefix to protein and ligands

pdbMol Molecule containing all atoms

4.5.3.83 void initProtons (Molecule * *mol1*, int * *protons*, int *numP*, Molecule * *mol2*)

Initialize proton scores to small number.

Save scores from initial conformation.

Parameters:

mol1 A Molecule with protons to score

protons Array of proton indices in molecule->atoms[]

numP Number of protons in array

mol2 A 2nd Molecule against which we'll score our protons

4.5.3.84 Molecule * initStructures (FILE * *file*)

Parse everything from file into one single molecule.

Counts total atoms for entire PDB file and creates one big molecule. Parsing individual structures relies too much on PDB file correctness.

Parameters:

file File to parse

Returns:

Newly allocated Molecule

4.5.3.85 int inv_xform (Matrix4x4 * *m*, Matrix4x4 * *inv*)

Calculate the inverse of the transformation matrix.

(Works on only 3x3 upper left corner of the 4x4 matrix). Returns 0 inverse calculation fails.

Based on Rasmol code written by: Herbert J. Bernstein, yaya@bernstein-plus-sons.com, 6 March 1998

Parameters:

m A Matrix4x4

inv Storage for the calculated inverse matrix

Returns:

True if successful inverse matrix found

4.5.3.86 int is_heavy_metal_atom (Molecule * *mol*, int *at*)

Test if an atom is a heavy metal.

Parameters:

mol A Molecule

at Index into molecule->atoms[]

Returns:

True if passes test

4.5.3.87 int IS_HIS (Molecule * *mol*, int *atom*, int *H*)

Find motif: histidine.

```
        C
      /   \\
H - N1     N2 (- H*) <--- calculates coords of this as well
    |      |
    R      R

Test can be performed on several atoms:

If atom = -1, test on IS_HIS(H), already protonated - just check residue name
     H = -2, test on IS_HIS(N2), return farProton if found, -1 if not

Else test for IS_HIS(N1) with adj proton
```

Saves various HIS info

See also:

setHISInfo(). *

Parameters:

mol A Molecule with our atom to test

atom Index of N atom into molecule->atoms[], pass -1 if testing H

H Index of H atom into molecule->atoms[], pass -2 if testing N2

Returns:

True if passes test; will also return farProton index if found, -1 if not

4.5.3.88 int is_planar_N (Molecule * *mol*, int *at*)

4.5.3.89 int is_resonant_N (Molecule * *mol*, int *at*)

Test that C-N bond is resonant.

Check that the central C is planar. Check that the C-N bond lengths are resonant length.

Parameters:

mol A Molecule

at Index into molecule->atoms[]

Returns:

True if passes test.

4.5.3.90 int isAA (Molecule * *mol*, int *at*)

Test that an atom is part of a residue with standard amino acid names.

Parameters:

mol A Molecule

at Index into molecule->atoms[]

Returns:

True if passes test.

4.5.3.91 int isAmidine (Molecule * *mol*, int *n*, int *doubleN*)

Motif finder for amidine:.

```
          N <- doubleN
         //
-> N - C
  /     \
 R       N <-
```

doubleN argument allows this function to be called before and after bonds are assigned.

Parameters:

mol A Molecule

n Index of N atom into molecule->atoms[]

doubleN If true, bonds have already been assigned

Returns:

True if passes test

4.5.3.92 int isAmidine_p (Molecule * *mol*, int *atom*)

Motif finder for amidine when it's protonated:.

```
     H    H
      \  /
       N   <--- this atom
      //
R - C
      \
       N   <--- this atom
      /  \
     H    H
```

Parameters:

mol A Molecule

atom Index of N atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.93 int isAtomClose (Molecule * *mol1*, int *atom*, Vector3 * *v*, Molecule * *mol2*)

Test for out atom proximity to any atom of another molecule.

mol1 atom must be within LONGEST_HBOND distance of any atom in mol2.

Parameters:

mol1 A Molecule with our atom to test

atom Atom index into molecule->atoms[] to test, pass -1 if location test desired

v Potential atom location

mol2 A Molecule that may be nearby to our atom

Returns:

True if passes test

4.5.3.94 int isCofactorCation (Molecule * *mol*, int *atom*)

Test for cofactor cation.

Parameters:

mol A Molecule

atom Index into molecule->atoms[]

Returns:

True if passes test.

4.5.3.95 int isElement (char * *id*)

Check that given string is a recognized element.

Does some cleaning up too. Used in conjunction with my_clean_atom_type().

See also:

my_clean_atom_type().

Parameters:

id PDB id

Returns:

True if passes test.

4.5.3.96 int isFlexibleProton (Molecule * *mol*, int *atom*)

Test for identification of flexible protons.

Accept OH, SH, Histidine for now. If histidine, does further processing.

See also:

IS_HIS()

Parameters:

mol A Molecule

atom Atom index into molecule->atoms[] to test

Returns:

True if passes test

4.5.3.97 int isGraphISO (Molecule * *mol1*, Molecule * *mol2*, double * *min_rms*, double * *min_hrms*, int ** *match*, int * *nmatch*)

Test to see if two molecules are graph isomorphisms.

Parameters:

mol1 A Molecule

mol2 Another Molecule

min_rms Storage for the minimum RMS

min_hrms Storage for the mininmum heavy atom RMS

match 2xN array of atom matches for both molecules where N is the number of atoms

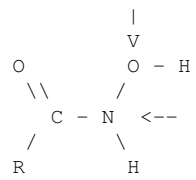
nmatch Number of matches

Returns:

True if passed test

4.5.3.98 int isHydroxamicAcid (Molecule * *mol*, int *n*, int *o*)

Motif finder for hydroxamic acid:.



Parameters:

mol A Molecule

n Index of N atom into molecule->atoms[]

o Index of O atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.99 int isNitro (Molecule * *mol*, int *N*)

Recognize this:.

```
      R
      |
      N+
    /  \\ <-- resonates like carboxyl
  O     O
 -.5   -.5
```

Parameters:

mol A Molecule

N Index of nitrogen atom into molecule->atoms[]

Returns:

-1 if not nitro
index of shortest bonded O

4.5.3.100 int isoDFS (Molecule * *mol1*, int *a1*, Molecule * *mol2*, int *a2*, int ** *match*, int * *nmatch*, int *mark*, double * *min_rms*, double * *min_hrms*)

Recursion entry point for graph isomorphism search.

Depth first search. Emphasizes protons.

Parameters:

mol1 A Molecule

a1 Atom index into mol1->atoms[]

mol2 Another Molecule

a2 Atom index into mol2->atoms[]

match 2xN array of atom matches for both molecules where N is the number of atoms

nmatch Number of matches

mark Mark used for this level of recursion (start with 1)

min_rms Storage for the minimum RMS

min_hrms Storage for the mininmum heavy atom RMS

Returns:

True if passed test

4.5.3.101 int isPlanarAtom (Molecule * *mol*, int *at*)

Atom planarity test.

Parameters:

mol A Molecule

at Index of atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.102 int isPlanarSP2Carbon (Molecule * *mol*, int *atom*)

Tests if given atom is a planar sp2 carbon in need of a double bond.

Parameters:

mol A Molecule

atom Index of carbon atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.103 int isResonantBond (Molecule * *mol*, int *a1*, int *a2*, int *order*, double *dist*)

Bond length is in between order and order-1.

Resonance dist must be less than SINGLE_BOND_LEN - fudge (0.06).

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

order Bond order to try (integer value)

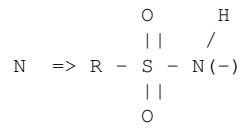
dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes resonance test

4.5.3.104 int isSulfonamide (Molecule * *mol*, int *atom*, Molecule * *newMol*)

Motif finder for sulonamide:.



Protonated as on right (usually metal chelating). Label atoms DONOR/ACCEPTOR if we're in midst of protonating a molecule.

Optional arguments may be passed NULL.

Parameters:

mol A Molecule

atom Index of S atom into molecule->atoms[]

newMol Optional - Molecule identical to mol that's in process of being protonated

Returns:

True if passes test

4.5.3.105 int isThiodiimine (Molecule * *mol*, int *s*)

Motif finder for thiodiimine:.

Parameters:

mol A Molecule

s Index of S atom into molecule->atoms[]

Returns:

True if passes test

4.5.3.106 int isValidAABond (Molecule * *mol*, int *a1*, int *a2*)

Determine if the AA bond we're looking at is valid.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[], bond partner to a2

a2 Atom2 index into molecule->atoms[], bond partner to a1

Returns:

True if passes test.

4.5.3.107 int isValidBond (Molecule * *mol*, int *a1*, int *a2*, int *bondOrder*)

Check special cases for bond validity.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

bondOrder Specific bond order

Returns:

True if passes test.

4.5.3.108 int isWater (Molecule * *mol*, int *atom*, int *molTest*)

Residue name test for water.

Parameters:

mol A Molecule

atom Index into molecule->atoms[]

molTest If true, test the mol has less than 3 atoms

Returns:

True if passes test.

4.5.3.109 void label_rings (Molecule * *mol*, int *aromatic*)

Mark bonds that are part of rings.

Parameters:

mol A Molecule

aromatic If true, mark the atoms of aromatic rings in the overloaded score field

4.5.3.110 void labelAromAtoms (Molecule * *mol*, int *bond*)

Given a single bond, locate connected atoms that are part of the aromatic system.

From the given bond: 1. Find all other atoms in the ring

2. Verify coplanarity

- True: label each ring atom as aromatic
- False: label each ring atom as regular

  3. Label all ring atoms as 'processed' by marking them AROMATIC

Parameters:

mol A Molecule

bond Index of bond into molecule->connections[] where we start our atom labelling

*

4.5.3.111 Complex * makeComplex (int *numLigands*, int *numCofactors*, int *numWaters*)

Allocate the Complex structure.

Parameters:

numLigands Number of ligands

numCofactors Number of cofactors

numWaters Number of waters

Returns:

Newly allocated Complex

4.5.3.112 void mark_arom_cycle (Molecule * *mol*, int *at*)

Mark atoms that are part of an aromatic cycle.

Copy of mark_connected_atoms with additional test that atoms must be unmarked and aromatic in nature before being marked.

Parameters:

mol A Molecule.

at Index of atom into molecule->atoms[] to start connectivity search

4.5.3.113 int markBumps (Molecule * *mol1*, Molecule * *mol2*, Molecule * *markMol*)

Mark the atoms that suffer large bump scores.

For every proton, calculate self bump and vs mol2. Mark their owners if bump < BUMP_THRESH. Makes use of the atom.close[0] field in Atom_struct to mark the parent of the offending proton.

Parameters:

mol1 A Molecule with bumps we'd like to mark

mol2 A Molecule that's causing the bumps

markMol A copy of mol1 where we'll mark the offending parents of bumpy atoms

4.5.3.114 void markLocalRing (Molecule * *mol*, int * *atoms*, int *nAtom*)

Mark atoms that are being traversed as we detect local rings.

Intersection points can be in multiple rings

- mark -1 so can be processed in future

Non-intersection pts are marked 'untouchable'

- atom[n].close[2] = 1

Parameters:

mol A Molecule

atoms Array of potential local ring atoms

nAtom Number of atoms in array

4.5.3.115 void markPlanarAtoms (Molecule * *mol*, int * *ringAtoms*, int *numAtom*)

Grab sets of 5 interconnected atoms and test for planarity.

If planar, mark atoms as so. Do this for all atoms in the array.

Parameters:

mol A Molecule

ringAtoms Array of atom indices into molecule->atoms[]

numAtom Number of atoms in ringAtoms

4.5.3.116 void markResidue (Molecule * *mol*, int *atom*, char * *res*, int *resNum*, int * *val*)

Mark the atom with a specific value if it belongs to the given residue.

Recursively mark this atom's neighbors as well. Mark starts with value and ascends consecutively with distance from the start atom.

Parameters:

mol A Molecule

atom Index into molecule->atoms[]

res Residue name

resNum Residue number

val Mark to give atoms; represents recursion level/distance from start atom

4.5.3.117 Molecule* merge_molecules (Molecule * *mol1*, Molecule * *mol2*)

4.5.3.118 void mergeComplexes (Complex * *to*, Complex * *from*)

Merge two complexes.

Given comp1 and comp2, merge the contents of comp2 into comp1. comp2 pointers to everything but protein & water are still valid. comp1 pointers now point to merged complex elements.

Parameters:

to A Complex; this structure will be updated to contain to the merged complex

from Another complex; protein and water pointers will be invalid

4.5.3.119 void moveProtons (Molecule * *mol1*, Molecule * *mol2*, int * *protons*, int *numP*, Molecule * *testmol*, int *currIter*)

Randomly move the protons.

We do random moves because we don't know what the best protonation network will be. Usually called multiple times.

Parameters:

mol1 A Molecule that contains our proton

mol2 A 2nd Molecule against which we'll score our proton

protons Array of proton indices into mol1->atoms[]

numP Number of protons in array

testmol Optional - allocated Molecule structure to hold sampled proton positions

currIter Current iteration of proton movement

4.5.3.120 void my_add_atom (Molecule * *mol*, int *at*, Vector3 * *v*, char * *el*)

Add an atom to the given molecule.

Parameters:

mol A Molecule

at Index into molecule->atoms[]

v Location of atom

el Element id of atom

4.5.3.121 int my_atoms_in_ring (Molecule * *mol*, int *a1*, int *a2*, int *aromatic*)

Check if two atoms are part of a ring.

Parameters:

mol A Molecule

a1 Atom index into molecule->atoms[]

a2 Atom index into molecule->atoms[]

aromatic If true, atoms may already be processed so skip it

Returns:

True if passes test

4.5.3.122 void my_clean_atom_type (char *id*[ ])

Clean the pdb id leaving only the element information.

Does this in place. Return true if element is recognized. Can be used as a test isElement().

Parameters:

id PDB id string

4.5.3.123 Molecule * my_clean_molecule_deprot_acid (Molecule * *old_mol*)

Remove hydrogens from a specific motif.

```
Remove H off of:
     /
H-O=C
     \
```

Parameters:

old_mol A Molecule

Returns:

Newly allocated Molecule with alterations

4.5.3.124 Molecule* my_free_molecule (Molecule * *mol*)

Free memory allocated to a Molecule.

Handles pdbgrind specific data structures.

Parameters:

mol A Molecule

Returns:

If this molecule is part of a linked list, return the next molecule. Otherwise, return NULL.

4.5.3.125 int my_is_sp2_atom (Molecule * *mol*, int *at*)

Test for sp2 atom.

Before protonation, determine if this atom sp2:

- has aromatic bond
- C w/ double bond
- Any atom with multiple bonds, 1 being double

Parameters:

mol A Molecule

at Index into molecule->atoms[]

Returns:

True if passes test

4.5.3.126 void my_label_atoms (Molecule * *mol*)

Assign all atoms a type: steric/donor/acceptor.

Does the special cases for metal chelates

Parameters:

mol A Molecule

4.5.3.127 void my_label_radii (Molecule * *mol*)

Assign standard VdW radii to every atom in a molecule.

Parameters:

mol A Molecule

4.5.3.128 Molecule* my_make_molecule (int *natoms*, int *nbonds*)

Make a molecule with the given amount of storage.

Handles pdbgrind specific data structures.

Parameters:

natoms Number of atoms.
nbonds Number of bonds.

Returns:

Newly allocated Molecule

4.5.3.129 void my_mark_connected_atoms_n (Molecule * *mol*, int *at*, int *nmark*)

Mark connected atoms with a given integer.

Sentinel for unmarked atoms = -1.

Parameters:

mol A Molecule.
at Index of atom into molecule->atoms[] to start connectivity search
nmark Mark to give to connected atoms

4.5.3.130 Molecule * my_merge_molecules (Molecule * *mol1*, Molecule * *mol2*)

Given 2 molecules, merge them into 1 molecule.

The two given molecules are NOT freed.

Parameters:

mol1 One molecule
mol2 A second molecule

Returns:

Merged molecule

4.5.3.131 Molecule * my_protonate_molecule (Molecule * *old_mol*)

Protonate a molecule.

Checks atom environment and available valency before adding protons.

Parameters:

old_mol An unprotonated Molecule

Returns:

A protonated Molecule

4.5.3.132 Molecule * my_read_mdl_file (char * *path*, int *stripH*, int *copyElement*)

Handle reading of mdl files.

Parameters:

path Full pathname to file to be parsed

stripH If true, remove all hydrogens from molecule parsed

copyElement If true, retain element id from mdl file, else clean it as usual

See also:

clean_atom_type().

Returns:

Newly allocated Molecule

4.5.3.133 Molecule * my_read_mol2_file (char * *path*, char * *chainID*, int *stripH*)

Read a mol2 file.

Optional arguments may be passed NULL.

Parameters:

path Full pathname of the file to be read in

chainID Optional - specific chain identifier to assign to the complex

stripH If true, remove all H's from complex

Returns:

Newly allocated Molecule

4.5.3.134 Complex* my_read_molecule_file (char * *path*, char * *chainID*, char * *m*, char * *scaleM*, int *stripH*)

Read a molecule file.

Input can be .pdb, .mol, .mol2. Optional arguments may be passed NULL.

Parameters:

path Full pathname to input file

chainID Optional - assign a specific chainID to the complex

m Optional - transform the read in complex by the Matrix4x4 m

scaleM Optional - Preprocess complex using scale matrix

See also:

processMatrix

Parameters:

stripH If true, remove H from the complex

Returns:

Newly allocated Complex

4.5.3.135 Complex* my_read_pdb_file (char * *path*, char * *chainID*, char * *m*, char * *scaleM*, int *stripH*, int *ligand*)

Read a pdb file.

http://www.wwpdb.org/documentation/format23/sect9.html

See the "ATOM" section.

Placement of parsed text is HARDCODED, will need to be updated as file format changes (expansion of columns in atom serial #, for instance).

Optional arguments may be passed NULL.

Parameters:

path Full pathname to input file

chainID Optional - assign a specific chainID to the complex

m Optional - transform the read in complex by the Matrix4x4 m

scaleM Optional - preprocess complex using scale matrix

See also:

processMatrix

Parameters:

stripH If true, remove all H's from the complex

ligand If true, treat complex like a ligand (infer bond connectivity by distance vs by residue)

Returns:

Newly allocated Complex

4.5.3.136 void my_write_mol2_file (char * *path*, Conformer * *conf*, FILE * *fd*)

Write a molecule in mol2 format.

Optional arguments may be passed NULL.

Parameters:

path Filename to output

conf A Conformer

fd Optional - open file pointer to write

4.5.3.137 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.5.3.138 int nearbyCarbonyl (Molecule * *mol*, int *c*)

Test for a nearby carbonyl during processing of aromatic bonds.

```
R
 \
  C - O  where the C-O bond is of ~ C=O length
 /
R
```

Parameters:

mol A Molecule

c Index into molecule->atoms[] of a carbon atom suspected of participating in a carbonyl

Returns:

If passes nearby carbonyl test

4.5.3.139 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.5.3.140 void optimizeProtons (Molecule * *protein*, Molecule ** *ligand*, Molecule ** *ligand_noH*)

Optimize the interactions of flexible protons of the protein-ligand complex.

- Smooth bumps caused by misplaced protons
- Maximize their proton interaction with their acceptors by sampling rotamers/tautomers

Parameters:

protein Protein Molecule

ligand Pointer to ligand Molecule

ligand_noH Pointer to an identical ligand Molecule that's been stripped of hydrogens

4.5.3.141 void parseAtomInfo (FILE * *file*, Molecule * *pdbMol*, char * *chainID*, int *stripH*)

Parse all atom info from a file into a single molecule.

- Element
- Residue info
- XYZ

Optional arguments may be passed NULL.

Parameters:

file File to parse
pdbMol Allocated Molecule to store atom information
chainID Optional - specific chain identifier to assign to the complex
stripH If true, remove all H's from complex

4.5.3.142 Matrix4x4 * parseMatrix (FILE * *file*, char * *matrixFile*, int *xform*)

Parse a matrix from a file.

Reads:

- SCALE matrix from pdb file OR
- matrix from user-supplied file Format:

```
  rotation matrix
  __|__
[ a b c x ]
[ d e f y ]  User-defined matrix
[ g h i z ]
        |____translation vector
```

  Rows separated by newlines; columns separated by whitespace.

Optional arguments may be passed NULL.

Parameters:

file Optional - Open file pointer to a PDB file where we can find a scale matrix
matrixFile Full pathname to a file containing a matrix
xform If true, we're parsing a xform matrix (vs a scale matrix)

Returns:

Newly allocated Matrix4x4

4.5.3.143 void parsePDBAtom (char * *the_line*, Conformer * *conf*, int *atom*)

Parse and clean up the pdb atom name field.

Parameters:

the_line ATOM line of a PDB file
conf Conformer that contains this atom
atom Index number of this atom in conformer->atom[]

4.5.3.144 int passBondThresh (Molecule * *mol*, int *a1*, int *a2*, int *order*, double *dist*)

Given two atoms and distance between them, determine if the appropriate bond threshold is passed.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

order Bond order to try (integer value)

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes bond thresh

4.5.3.145 int passDoubleBondThresh (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Thresholding code specific to double bonds.

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.146 void plotPoint (Molecule * *mol*, Vector3 * *pt*, char * *name*)

Add a "point" to a molecule.

Used for testing - points can be seen when molecule is visualized.

Parameters:

mol A Molcule

pt A point

name Output file name

4.5.3.147 int possibleBond (Molecule * *mol*, int *a1*, int *a2*, int *bondOrder*)

Test for potential bond of a given order between two atoms.

Determines:

- Bond atoms can accept the bond order
- That bond is a valid one

Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

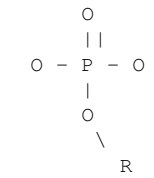
bondOrder Specific bond order

Returns:

True if passes test

4.5.3.148 int potentialPhosphoester (Molecule * *mol*, int *a1*, int *a2*, double *dist*)

Special case test for potentialPhosphoester ——————.



Parameters:

mol A Molecule

a1 Atom1 index into molecule->atoms[]

a2 Atom2 index into molecule->atoms[]

dist Distance between a1 and a2, pass -1 if not precomputed

Returns:

True if passes double bond test

4.5.3.149 void printComplexInfo (Complex * *complex*)

Print out info regarding the complex.

- Protein number of atoms, bonds.
- Number of ligands
- Number of cofactors
- Number of waters

Parameters:

complex A Complex

4.5.3.150 void printContents (char * *key*, void * *data*)

Print to stderr hash contents for a given key.

A look at what's inside hash. Function useful for iterating over a hash.

See also:

HashEnumerate()

Parameters:

key Hash key

data A Molecule

4.5.3.151 void printProtonScores (Molecule * *mol*, int * *protons*, int *numP*)

Informational print out of current proton scores.

Parameters:

mol A Molecule

protons Array of proton indices into molecule->atoms[]

numP Number of protons

4.5.3.152 void processLigands (Complex * *comp*)

Process the ligand information.

See also:

processProtein()

Parameters:

comp Complex containing ligands

4.5.3.153 Molecule * processMatrix (FILE * *file*, Molecule * *pdbMol*, char * *matrixFile*, char * *scaleM*)

Transform the molecule by the given matrices.

Before the transformation matrix can be applied, the molecule must first be converted from orthogonal to crystallographic coordinates using a scale matrix. This scale matrix is usually included in the pdb file itself, but this function also allows it to be passed in. As a convenience, any time a pdb file is processed by pdbgrind, it writes the scale matrix, if found. If the molecule is already in crystallographic coordinates, then the optional scale matrix arguments may be ignored.

For matrix file format,

See also:

parseMatrix()

Optional arguments may be passed NULL.

Parameters:

file Optional - open pointer to PDB file from which we can parse the scale matrix

pdbMol A Molecule parsed from a PDB file

matrixFile Full path to a transformation matrix

scaleM Optional - full path to a scale matrix

Returns:

Transformed Molecule

4.5.3.154 void processPlanarCarbons (Molecule * *mol*)

Verify planarity of all carbons.

Requirements:

- 3 current substituents
- planar carbon
- should NOT get protonated

Parameters:

mol A Molecule

4.5.3.155 void processProtein (Complex * *complex*, int *ligand*)

Process the protein information.

- infer bond order
- protonate
- label atoms

Parameters:

complex Complex containing protein

ligand If true, process bond connectivity as a ligand ignoring residue information

4.5.3.156 void processUnboundAtom (Complex * *complex*, Molecule * *pdbMol*, int *atomNum*, int * *numCofactors*)

Process any unbound atoms leftover as either water or cofactors.

Parameters:

complex A Complex where the water and cofactors will be stored

pdbMol Molecule that contains all the atoms

atomNum Index into pdbMol->atoms[] of atom being processed

numCofactors Storage for number of cofactors found

4.5.3.157 int propagateAromBond (Molecule * *mol*, int *atom*, int *order*, int *priority*)

Continue propagating aromatic bonds from the given atom.

Parameters:

mol A Molecule

atom Index of atom into molecule->atoms[] from which we'll propagate the bond assignment

order If > 0, double bond

priority If true, will attempt to assign double bond despite valence restriction

Returns:

True if successful

4.5.3.158 int protonNearAcceptor (Molecule * *mol1*, int *proton*, Vector3 * *protLoc*, Molecule * *mol2*, int *acceptor*)

Check that the proton-acceptor distance is within the threshold for h-bonds.

If no acceptor atom is given (acceptor = -1), cycle through all atoms of the mol2 to find those acceptor atoms close to the given proton in mol1. This is used only when initializing protons.

Optional arguments may be passed NULL.

Parameters:

mol1 A Molecule with our proton to test

proton Proton index into mol1->atoms[] to test

protLoc Optional - test this potential atom location

mol2 A Molecule that may be nearby to our atom

acceptor Index into mol2->atoms[] of an acceptor to test, pass -1 to test all atoms in mol2

Returns:

True if passes test

4.5.3.159 double real_total_bonds (Molecule * *mol*, int *at*)

4.5.3.160 void removeBoundAtom (Molecule * *mol*, int *a1*, int *a2*)

Remove the bound atom a2 from the connected_atoms[] of a1.

Helper to copy_molecule_without_bond().

See also:

copy_molecule_without_bond()

Parameters:

mol A Molecule

a1 An atom index into molecule->atoms[], former bond partner of a2

a2 Index into molecule->atoms[] of an atom whose bond with a1 we will break

4.5.3.161 void removeH (Molecule * *mol*, int *atom*)

Remove protons bonded to this atom.

Parameters:

mol A Molecule

atom Index into molecule->atoms[] of an atom from whom we're stripping bound H's

4.5.3.162 void resetAllAromBonds (Molecule * *mol*)

Reset all the bond assignments to single.

Parameters:

mol A Molecule

4.5.3.163 void resetAromBond (Molecule * *mol*, int *atom*)

Recursively traverses a ring cycle, resetting bond orders to single and aromatic atoms to unprocessed.

Parameters:

mol A Molecule

atom Index into molecule->atoms[] to start recursion

4.5.3.164 void resetAromRing (Molecule * *mol*, int *bond*)

Recursion entry point for resetting the aromatic assignment of a ring.

Parameters:

mol A Molecule

bond Index of bond into molecule->connections[] to begin resetting

4.5.3.165 int retryAssignAromBond (Molecule * *mol*, int *bond*, int *attemptOrder*, int *bondOrder*, int *priority*)

Retry aromatic bond assignment but with reversed bond order.

N is tricky case: sometimes foil aromatic bond assignment.

- Case 1: N=R bond, try N-R then repropagate
- Case 2: N-R bond, try N=R then repropagate

(Failure defined in propagateAromBond() - aromatic sp2 C not given double bond)

See also:

propagateAromBond()

Parameters:

mol A Molecule

bond Index of bond into molecule->connections[] to assign

attemptOrder If > 0, Try double bond first

bondOrder The bond order previously tried

priority If true, will attempt to assign double bond despite valence restriction

Returns:

0 Failure
1 Single bond assigned
2 Double bond assigned

4.5.3.166 int ringIsCoplanar (Molecule * *mol*, int * *ringAtoms*, int *len*)

Test for ring coplanarity.

Parameters:

mol A Molecule

ringAtoms Array of atom indices into molecule->atoms[]

len Number of atoms in ringAtoms

Returns:

True if passes test

4.5.3.167 void rotateProton (Molecule * *mol*, int *proton*, int *click*)

Rotate a proton by a certain number of clicks.

Click = degrees of rotation about v1 axis.

```
           S
       N  /
norm   | / v2
       |/      / \
       C --- A |.|
rvec  /  v1   \\ /
     /   out   H
    R

C = central atom
S = sibling
A = adjacent
H = hydrogen
```

Parameters:

mol A Molecule

proton Index of proton in molecule->atoms[]

click Degrees to rotate anticlockwise

4.5.3.168 void sampleHydroxylRotamer (char * *file*, int *atom*)

Given a specific hydroxyl H, sample 360 rotamers, outputting each.

Also output the entire sampling.

Parameters:

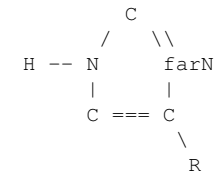
file Full pathname to a mol2 file

atom Index of hydroxyl H

4.5.3.169 void score_HIS_tautomer (Molecule * *mol1*, int *proton*, Molecule * *mol2*, double * *hi*)

Processing a HIS tautomer, score the farN acceptor.

Searching for the optimum proton assignment for a HIS tautomer. There may not be a viable acceptor for our current proton location, but if there's a viable donor for the farN, that's reason enough to maintain the proton position.



Parameters:

mol1 A Molecule that contains our proton

proton Index of HIS proton into mol1->atoms[]

mol2 A 2nd Molecule against which we'll score our proton

hi Pointer to the current hi score for this HIS

4.5.3.170 double scoreHypoPolarPair (Molecule * *mol1*, Vector3 * *proton*, double *atomRadius*, int *atomType*, Molecule * *mol2*, int *central*, int *otherAtom*)

Score a hypothetical proton.

Exactly same as scoreProton except this takes a hypothetical atom that has yet to be added to the molecule and scores its interaction vs. another atom using distance and directionality. Alternatively, bump can also be scored for an atom of arbitrary radius.

Parameters:

mol1 A Molecule that will contain the hypothetical atom

proton Hypothetical position of an atom

atomRadius Radius of the hypothetical atom

atomType Type of hypothetical atom (DONOR/ACCEPTOR)

mol2 A 2nd Molecule against which we'll score our proton

central Index of the hypothetical atom's parent atom into mol2->atoms[]

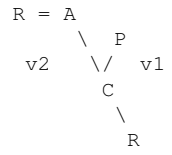
otherAtom Acceptor atom index into mol2->atoms[]; if -1 then retrieve only bump score (steric crash)

Returns:

Score

4.5.3.171 double scoreProton (Molecule * *mol1*, int *protAtom*, Molecule * *mol2*, int *acceptAtom*)

Score the interaction between a proton and an acceptor based on distance and orientation.




```
R -  R Group
A -  Acceptor atom
P -  Proton
C -  Central atom
v2 - Vector CA
v1 - Vecotr CP
```

Parameters:

mol1 A Molecule with protons to score

protAtom Index into mol1->atoms[] of a proton

mol2 A 2nd Molecule against which we'll score our proton

acceptAtom Acceptor atom index into mol2->atoms[]; if -1 then retrieve only bump score (steric crash)

Returns:

Score

4.5.3.172 void scoreProtonArray (Molecule * *mol1*, Molecule * *mol2*, int * *protons*, int *numP*)

Get scores for all protons in the array.

Wrapper function for

See also:

updateProtonScore()

Parameters:

mol1 A Molecule with protons to score

mol2 A 2nd Molecule against which we'll score our protons

protons Array of proton indices in molecule->atoms[]

numP Number of protons in array

4.5.3.173 Complex * separateComp (Molecule * *pdbMol*, char * *path*)

Separate the different unconnected components contained within the given Molecule.

- Mark connected atoms with a unique component ID
- Largest graphs = protein (n-mers)
- Everything else = ligand
- Waters & cofactors treated separately

Parameters:

pdbMol Molecule containing all atoms
path Name to prefix to protein and ligands

Returns:

Newly allocated Complex

4.5.3.174 void setBondOrderConnectedAtoms (Molecule * *mol*, int *bondNum*)

Change the bond order of a given bond.

Given a bond, set the bond order of the connected atoms to be that of the given bond.

Parameters:

mol A Molecule
bondNum Index of bond in molecule->connections[]

4.5.3.175 void setHISInfo (Molecule * *mol*, int *H*, int *adjN*, int *C*, int *farN*)

Set the information about HIS that's needed later in swapHISProton().

See also:

swapHISProton().
AtomMiscData::histinfo.

Parameters:

mol A Molecule
H Donor proton index into molecule->atoms[]
adjN Index into molecule->atoms[] of N atom adjacent to the donor proton
C Index into molecule->atoms[] of Carbon connected to adjN and farN
farN Index into molecule->atoms[] of far N atom

4.5.3.176 void smoothBumps (Complex * *comp*, Molecule ** *ligand_noH*, Molecule ** *ligand*)

Redo ligand bond ordering if there are obvious steric clashes due to current protonation state.

Mark the owners of clashing protons. Do not allow them to be protonated. Bumps marked in atom->close[0].

Parameters:

comp A Complex

ligand_noH A ligand stripped of all hydrogens

ligand A ligand with bumps to smooth

4.5.3.177 int startAromBondRecursion (Molecule * *mol*, int * *bonds*, int *nbonds*, int *startBond*)

Entry point into recursive aromatic bond assignment.

Tries four times with different initial order assignment to the first bond. Propagate our alternating assignment of single and double bonds until we find a solution for our aromatic system.

Parameters:

mol A Molecule

bonds Array of bond indices into molecule->connections[]

nbonds Number of bonds

startBond Index into bonds argument where we'll start propgation

Returns:

True if solution is found

4.5.3.178 int sulfonS (Molecule * *mol*, int *atom*)

Return number of double-bonded O's in a sulfonyl.

Parameters:

mol A Molecule

atom Index of S atom into molecule->atoms[]

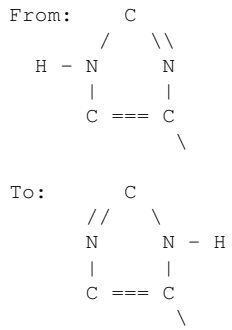
Returns:

Number of S=O found

4.5.3.179 void swapHISProton (Molecule * *mol*, int *H*)

Swap the H positions on the tautomer by moving the double bond.

The H will then be assigned appropriately later during protonation of the molecule.



Parameters:

mol A Molecule

H Index of proton in molecule->atoms[]

4.5.3.180 void sybylAtom (Molecule * *mol*, int *atom*, char * *sybyl*)

Assign correct sybyl atom type to the given atom.

Careful: sybyl char[] len = 8!

http://www.tripos.com/mol2/atom_types.html#11465

Parameters:

mol A Molecule.

atom Index of atom into molecule->atoms[]

sybyl Allocated string buffer to store processed sybyl name

4.5.3.181 void sybylC (Molecule * *mol*, int *atom*, char * *sybyl*)

Sybyl atom types for carbon.

Parameters:

mol A Molecule.

atom Index of atom into molecule->atoms[]
sybyl Allocated string buffer to store processed sybyl name

4.5.3.182 void sybylN (Molecule * *mol*, int *atom*, char * *sybyl*)

Sybyl atom types for nitrogen.

Parameters:

mol A Molecule.
atom Index of atom into molecule->atoms[]
sybyl Allocated string buffer to store processed sybyl name

4.5.3.183 void sybylO (Molecule * *mol*, int *atom*, char * *sybyl*)

Sybyl atom types for oxygen.

Parameters:

mol A Molecule.
atom Index of atom into molecule->atoms[]
sybyl Allocated string buffer to store processed sybyl name

4.5.3.184 void sybylP (Molecule * *mol*, int *atom*, char * *sybyl*)

Sybyl atom types for phosphorus.

Parameters:

mol A Molecule
atom Index of atom into molecule->atoms[]
sybyl Allocated string buffer to store processed sybyl name

4.5.3.185 void sybylS (Molecule * *mol*, int *atom*, char * *sybyl*)

Sybyl atom types for sulfur.

Parameters:

mol A Molecule.
atom Index of atom into molecule->atoms[]
sybyl Allocated string buffer to store processed sybyl name

4.5.3.186 int total_bonds (Molecule * *mol*, int *at*)

4.5.3.187 void trimMol (Molecule * *mol*, Vector3 *pt*, double *radius*)

Given a pt and radius, trim any atoms not within the defined sphere of interest.

Atoms that survive trimming are marked. Keeps all residues intact. Prints out the number of residues left.

Used in conjunction with other functions:

See also:

trimProtein()
write_trimmed_protein_mol2()

Parameters:

mol A Molecule

pt Center of sphere of interest

radius Radius of sphere of interest

4.5.3.188 void trimProtein (Complex * *complex*, Vector3 *pt*, double *radius*)

Given a pt and radius, trim protein atoms not within the defined sphere of interest.

Wrapper for trimMol().

See also:

trimMol()

Parameters:

complex A Complex

pt Center of sphere of interest

radius Radius of sphere of interest

4.5.3.189 int updateProtonScore (Molecule * *mol1*, int *proton*, Molecule * *mol2*, int *movement*)

Rescore all protons keeping improved scores.

Overloads atoms[N].charge to report bump scores.

Parameters:

mol1 A Molecule with protons to score

proton Index into molecule->atoms[] of a proton

mol2 A 2nd Molecule against which we'll score our proton

movement If true, protons recently moved, make sure to score both HIS tautomers

Returns:

True if any proton score improved

4.5.3.190 int V3AllPlanar (Vector3 ** *vectors*, int *len*)

Determine if an arbitrary number of points are all coplanar.

Requirements:

- First 3 points determine the plane
- Point-plane distance for all other points should be close to zero

Parameters:

vectors Array of points to test

len Number of points

Returns:

True if passes test

4.5.3.191 int V3Collinear (Vector3 * *v1*, Vector3 * *v2*, Vector3 * *v3*)

Determine if the given points are collinear.

Should be true:

- d = 0 in the point-line distance
- d = |(x2-x1) x (x1-x3)| == 0

http://mathworld.wolfram.com/Collinear.html

Parameters:

v1 Point 1

v2 Point 2

v3 Point 3

Returns:

True if passes test

4.5.3.192 int V3Planar_sp3 (Vector3 * *a*, Vector3 * *b*, Vector3 * *c*, Vector3 * *d*)

Test four points for planarity.

Parameters:

a Point A

b Point B

c Point C

d Point D

Returns:

True if passes test

4.5.3.193 int verifyMatrix (Matrix4x4 * *m*)

Verify validity of transformation matrix.

Parameters:

m A Matrix4x4

Returns:

True if passes test

4.5.3.194 int weirdElement (char * *string*, Molecule * *mol*, int *atom*)

Process idiosyncrasies of pdb atom names.

First 2+ letters of atom name that are clearly not elements. Cleans in place.

Parameters:

string PDB atom id

mol A Molecule

atom Index into molecule->atoms[]

Returns:

True if processed succesfully

4.5.3.195 int write_atom_mol2 (FILE * *file*, Molecule * *mol*, int *currAtom*, int *trim*)

Write out an atom in mol2 format.

Decomp the my_write_mol2_file a bit. Writes out all of the molecule's atoms, starting with a specific atom index.

Parameters:

file An open file pointer

mol A Molecule

currAtom Atom index into molecule->atoms[] to start writing

trim If true, write only those atoms that have not been trimmed return Number of written atoms

4.5.3.196 int write_bond_mol2 (FILE * *file*, Molecule * *mol*, int *currBond*, int *trim*)

Write out bond in mol2 format.

Decomp the my_write_mol2_file a bit. Writes out all of the molecule's bonds, starting with a specific bond index.

Parameters:

file An open file pointer

mol A Molecule

currBond Bond index into molecule->connections[] to start writing

trim If true, write only those bonds that have not been trimmed return Number of written bonds

4.5.3.197 void write_complex_mol2 (Complex * *comp*, int *trim*, char * *filename*)

Write out the complex in mol2 format.

Each of the following will have its own mol2 file:

- protein + n-mers + cofactors
- ligands
- water

Optional arguments may be passed NULL.

Parameters:

comp Complex to write out

trim If true, write out only those atoms marked as trimmed (

See also:

trimMol)

Parameters:

filename Optional - use this name as the file prefix for output

4.5.3.198 void write_protein_mol2 (char * *path*, Complex * *comp*, FILE * *fd*, int *trim*, int *knowNumAtoms*, int *knowNumBonds*, int *knowNumRes*)

Write out the protein to mol2 format.

Optional arguments may be passed NULL.

Parameters:

path Filename to output

comp Complex containing protein

fd Optional - open file pointer

trim If true, output only those atoms marked as trimmed

See also:

trimMol()

Parameters:

knowNumAtoms Optional - Number of atoms (pass -1 if unknown)

knowNumBonds Optional - Number of bonds (pass -1 if unknown)

knowNumRes Optional - Number of residues (pass -1 if unknown)

*

4.5.3.199 void write_substructure_mol2 (FILE * *file*, Molecule * *mol*, int *trim*)

Write the substructure info (mostly just chainID).

Assumes the pdb file is organized by residues (i.e. as we read atoms, atoms are ordered consecutively by residue groups).

Parameters:

file An open file pointer

mol A Molecule

trim Write out substructure only for residues that have not been marked as trimmed

4.5.3.200 void write_trimmed_protein_mol2 (char * *path*, Complex * *comp*)

Function to call for writing out trimmed proteins.

Need to call trimMol first.

See also:

trimMol()

Parameters:

path Filename to output

comp A Complex

4.5.3.201 void writeMatrix (Matrix4x4 * *m*, FILE * *file*, char * *name*)

Write a matrix to file.

For file format,

See also:

parseMatrix()

Parameters:

m A Matrix4x4

file Optional - allows writing of matrix to stderr

name Output filename

4.5.4 Variable Documentation

4.5.4.1 int _copyElement

Maintain element id parsed from file.

4.5.4.2 char _crazyAtom[8]

Report pdb element ids not currently captured.

4.5.4.3 int _crazyFlag = 0

True if we've parsed a pdb element id never seen before.

4.5.4.4 int _opt = 0

Begin testing for hypothetical protons.

4.5.4.5 Molecule* _optProt

Spare pointer to protein, don't use if _opt is off.

4.6 pdbgrind.h File Reference

4.6.1 Detailed Description

pdbgrind public interface.

```
#include "../sflib/surflex-public.h"
#include "pdbgrind-types.h"
```

Defines

- #define PDBName(m, a) m → atoms[a].pdbatom

  Atom name as given in PDB file.

- #define RingAtom(m, a) m → atoms[a].ring_p

  If true, this atom is part of a ring.

- #define AtomMark(m, a) m → atoms[a].mark

  Atom marking utility.

- #define ResNum(c, a) c → molecule → atoms[a].resnum

  The number of the residue to which this atom belongs.

- #define AromAtom(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → aromatic

  If true, this atom is part of an aromatic bond.

- #define AtomNum(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → num

  The index for this atom into molecule->atoms[] or conformer->atom[].

- #define AtomScore(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → score

  Atom score, useful in optimizing protons.

- #define Chain(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → chain

  The monomer chain to which this atom belongs.

- #define Alternate(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → alternate

  An alternate position for this atom.

- #define HistInfo(m, h, n) ((AtomMiscData∗)(m → atoms[h].miscdata)) → histinfo[n]

Access hist information array.

- #define NRes(m) ((MoleculeMiscData∗)(m → miscdata)) → nres

  Number of residues contained by this molecule.

Functions

- Complex ∗ my_read_molecule_file (char ∗path, char ∗chainID, char ∗m, char ∗scaleM, int stripH)

  Read a molecule file.

- Complex ∗ my_read_pdb_file (char ∗path, char ∗chainID, char ∗m, char ∗scaleM, int stripH, int ligand)

  Read a pdb file.

- Molecule ∗ my_make_molecule (int natoms, int nbonds)

  Make a molecule with the given amount of storage.

- Molecule ∗ my_free_molecule (Molecule ∗mol)

  Free memory allocated to a Molecule.

- void write_complex_mol2 (Complex ∗comp, int trim, char ∗prefix)

  Write out the complex in mol2 format.

- void freeComplex (Complex ∗comp)

  Free the memory associated with the Complex.

- void printComplexInfo (Complex ∗complex)

  Print out info regarding the complex.

- void getMolMatrix (Molecule ∗a, Molecule ∗b, char ∗matrixName)

  Given 2 monomers in different alignments, retrieve the matrix which transforms a into b.

- void sampleHydroxylRotamer (char ∗file, int atom)

  Given a specific hydroxyl H, sample 360 rotamers, outputting each.

- void coerceMol (Complex ∗comp, int bond, int order, char ∗filename)

  Force a bond to have a certain bond order.

- void computeSame (Molecule ∗mol1, Molecule ∗mol2)

Compute the rms similarity of 2 molecules if they are of the same constituency.

- void centroidDist (Molecule *mol1, Molecule *mol2)

  Calculate the distance between the centroids of two molecules.

- int hasSulfonamide (Molecule *protein, Molecule *ligand)
- void getProteinCentroid (Complex *comp, Vector3 *pt)

  Find centroid for a complex.

- void getLigandCentroid (Complex *comp, Vector3 *pt, char *ligFile)

  Find the appropriate ligand centroid.

- void trimProtein (Complex *complex, Vector3 pt, double radius)

  Given a pt and radius, trim protein atoms not within the defined sphere of interest.

- void optimizeProtons (Molecule *protein, Molecule **ligand, Molecule **ligand_noH)

  Optimize the interactions of flexible protons of the protein-ligand complex.

- double findMinDist (Molecule *a, Molecule *b)

  Return the minimum distance between two molecules.

- void mergeComplexes (Complex *to, Complex *from)

  Merge two complexes.

4.6.2 Define Documentation

4.6.2.1 #define Alternate(m, a) ((AtomMiscData*)(m → atoms[a].miscdata)) → alternate

An alternate position for this atom.

4.6.2.2 #define AromAtom(m, a) ((AtomMiscData*)(m → atoms[a].miscdata)) → aromatic

If true, this atom is part of an aromatic bond.

4.6.2.3 #define AtomMark(m, a) m → atoms[a].mark

Atom marking utility.

4.6.2.4 #define AtomNum(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → num

The index for this atom into molecule->atoms[] or conformer->atom[].

4.6.2.5 #define AtomScore(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → score

Atom score, useful in optimizing protons.

4.6.2.6 #define Chain(m, a) ((AtomMiscData∗)(m → atoms[a].miscdata)) → chain

The monomer chain to which this atom belongs.

4.6.2.7 #define HistInfo(m, h, n) ((AtomMiscData∗)(m → atoms[h].miscdata)) → histinfo[n]

Access hist information array.

4.6.2.8 #define NRes(m) ((MoleculeMiscData∗)(m → miscdata)) → nres

Number of residues contained by this molecule.

4.6.2.9 #define PDBName(m, a) m → atoms[a].pdbatom

Atom name as given in PDB file.

4.6.2.10 #define ResNum(c, a) c → molecule → atoms[a].resnum

The number of the residue to which this atom belongs.

4.6.2.11 #define RingAtom(m, a) m → atoms[a].ring_p

If true, this atom is part of a ring.

4.6.3 Function Documentation

4.6.3.1 void centroidDist (Molecule * *mol1*, Molecule * *mol2*)

Calculate the distance between the centroids of two molecules.

Parameters:

mol1 Molecule A

mol2 Molecule B

4.6.3.2 void coerceMol (Complex * *comp*, int *bond*, int *order*, char * *filename*)

Force a bond to have a certain bond order.

Parameters:

comp A Complex (this function operates on the protein)

bond Index of bond in molecule->connections[]

order Bond order desired

filename Output name for new complex

4.6.3.3 void computeSame (Molecule * *mol1*, Molecule * *mol2*)

Compute the rms similarity of 2 molecules if they are of the same constituency.

Particular emphasis is on proton composition/positioning as this is primarily a check to see that our bond ordering assignment and opt_protons command are working.

Print results to stderr.

Parameters:

mol1 Molecule A

mol2 Molecule B

4.6.3.4 double findMinDist (Molecule * *a*, Molecule * *b*)

Return the minimum distance between two molecules.

Parameters:

a Molecule A

b Molecule B

Returns:

Distance

4.6.3.5 void freeComplex (Complex * *comp*)

Free the memory associated with the Complex.

Parameters:

comp A Complex

4.6.3.6 void getLigandCentroid (Complex * *comp*, Vector3 * *pt*, char * *ligFile*)

Find the appropriate ligand centroid.

If ligand not given, default to protein centroid.

Optional arguments may be passed NULL.

Parameters:

comp A Complex

pt Storage for the found centroid

ligFile Optional - file containing the ligand

4.6.3.7 void getMolMatrix (Molecule * *a*, Molecule * *b*, char * *matrixName*)

Given 2 monomers in different alignments, retrieve the matrix which transforms a into b.

Parameters:

a Molecule A

b Molecule B

matrixName Matrix filename to write out

4.6.3.8 void getProteinCentroid (Complex * *comp*, Vector3 * *pt*)

Find centroid for a complex.

Assumes all n-mers of the complex are approx same size.

Parameters:

comp A Complex

pt Storage for the found centroid

4.6.3.9 int hasSulfonamide (Molecule * *protein*, Molecule * *ligand*)

4.6.3.10 void mergeComplexes (Complex * *to*, Complex * *from*)

Merge two complexes.

Given comp1 and comp2, merge the contents of comp2 into comp1. comp2 pointers to everything but protein & water are still valid. comp1 pointers now point to merged complex elements.

Parameters:

to A Complex; this structure will be updated to contain to the merged complex

from Another complex; protein and water pointers will be invalid

4.6.3.11 Molecule* my_free_molecule (Molecule * *mol*)

Free memory allocated to a Molecule.

Handles pdbgrind specific data structures.

Parameters:

mol A Molecule

Returns:

If this molecule is part of a linked list, return the next molecule. Otherwise, return NULL.

4.6.3.12 Molecule* my_make_molecule (int *natoms*, int *nbonds*)

Make a molecule with the given amount of storage.

Handles pdbgrind specific data structures.

Parameters:

natoms Number of atoms.

nbonds Number of bonds.

Returns:

Newly allocated Molecule

4.6.3.13 Complex* my_read_molecule_file (char * *path*, char * *chainID*, char * *m*, char * *scaleM*, int *stripH*)

Read a molecule file.

Input can be .pdb, .mol, .mol2. Optional arguments may be passed NULL.

Parameters:

path Full pathname to input file

chainID Optional - assign a specific chainID to the complex

m Optional - transform the read in complex by the Matrix4x4 m

scaleM Optional - Preprocess complex using scale matrix

See also:

processMatrix

Parameters:

stripH If true, remove H from the complex

Returns:

Newly allocated Complex

4.6.3.14 Complex* my_read_pdb_file (char * *path*, char * *chainID*, char * *m*, char * *scaleM*, int *stripH*, int *ligand*)

Read a pdb file.

http://www.wwpdb.org/documentation/format23/sect9.html

See the "ATOM" section.

Placement of parsed text is HARDCODED, will need to be updated as file format changes (expansion of columns in atom serial #, for instance).

Optional arguments may be passed NULL.

Parameters:

path Full pathname to input file

chainID Optional - assign a specific chainID to the complex

m Optional - transform the read in complex by the Matrix4x4 m

scaleM Optional - preprocess complex using scale matrix

See also:

processMatrix

Parameters:

stripH If true, remove all H's from the complex

ligand If true, treat complex like a ligand (infer bond connectivity by distance vs by residue)

Returns:

Newly allocated Complex

4.6.3.15 void optimizeProtons (Molecule * *protein*, Molecule ** *ligand*, Molecule ** *ligand_noH*)

Optimize the interactions of flexible protons of the protein-ligand complex.

- Smooth bumps caused by misplaced protons
- Maximize their proton interaction with their acceptors by sampling rotamers/tautomers

Parameters:

protein Protein Molecule

ligand Pointer to ligand Molecule

ligand_noH Pointer to an identical ligand Molecule that's been stripped of hydrogens

4.6.3.16 void printComplexInfo (Complex * *complex*)

Print out info regarding the complex.

- Protein number of atoms, bonds.

- Number of ligands
- Number of cofactors
- Number of waters

Parameters:

complex A Complex

4.6.3.17 void sampleHydroxylRotamer (char * *file*, int *atom*)

Given a specific hydroxyl H, sample 360 rotamers, outputting each.

Also output the entire sampling.

Parameters:

file Full pathname to a mol2 file

atom Index of hydroxyl H

4.6.3.18 void trimProtein (Complex * *complex*, Vector3 *pt*, double *radius*)

Given a pt and radius, trim protein atoms not within the defined sphere of interest.

Wrapper for trimMol().

See also:

trimMol()

Parameters:

complex A Complex

pt Center of sphere of interest

radius Radius of sphere of interest

4.6.3.19 void write_complex_mol2 (Complex * *comp*, int *trim*, char * *filename*)

Write out the complex in mol2 format.

Each of the following will have its own mol2 file:

- protein + n-mers + cofactors

- ligands
- water

Optional arguments may be passed NULL.

Parameters:

comp Complex to write out

trim If true, write out only those atoms marked as trimmed (

See also:

trimMol)

Parameters:

filename Optional - use this name as the file prefix for output

4.7 utils.c File Reference

4.7.1 Detailed Description

Utility functions code.

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

Functions

- void exitError (char *msg, int code)
  Exit with error msg and code.
- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)

Copy source file to target.

- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.
- double myRound (double num, int power)
  round the number to the given power of 10.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.

Variables

- int crlf_p
  Newline status: if newlines are "\\r\\n" then true.

4.7.2 Function Documentation

4.7.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.7.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message
code Error code

4.7.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.7.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.7.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file
src Source file to copy

4.7.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open
mode fopen mode

Returns:

Newly opened file pointer if successful

4.7.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.7.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.7.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.7.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) > maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.7.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.7.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.7.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.7.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.7.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.7.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");

- }

3. And after loop completes for pretty printing:

   - fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

4.7.3 Variable Documentation

4.7.3.1 int crlf_p

Newline status: if newlines are "\\r\\n" then true.

4.8 utils.h File Reference

4.8.1 Detailed Description

Utility functions public interface.

```
#include "stdio.h"
```

Functions

- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- void exitError (char *msg, int code)
  Exit with error msg and code.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- double myRound (double num, int power)
  round the number to the given power of 10.
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.

- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)
  Copy source file to target.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.

4.8.2 Function Documentation

4.8.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.8.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message

code Error code

4.8.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.8.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.8.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file

src Source file to copy

4.8.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open

mode fopen mode

Returns:

Newly opened file pointer if successful

4.8.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.8.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.8.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.8.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) $>$ maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.8.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.8.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.8.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.8.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.8.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.8.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");
- }

3. And after loop completes for pretty printing:

- fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

Appendix B. Enhanced Protomols

B.1.1. Usage

This section will detail the usage of surflex-proto on the command line. Its format will be as follows:

Brief command description

General usage: **surflex-proto <command>** `args`

Example command

Generate enhanced protomols. The `sitemol` *is used to define the important proximal residues in the active site of a* `protein`*. The* `protomol` *is a prefix assigned to any protomol files generated.*

```
surflex-proto -fancy proto sitemol protein protomol
surflex-proto -fancy proto ligand.mol2 protein.mol2 p1
```

Generate solvation protomols composed solely of water molecules. The arguments are identical to that of the above enhanced protomol command.

```
surflex-proto -solvation proto sitemol protein protomol
surflex-proto -solvation proto ligand.mol2 protein.mol2 p1
```

B.1.2. Code Documentation

Contents

Chapter 1

Surflex-Protomol Data Structure Index

1.1 Surflex-Protomol Data Structures

Here are the data structures with brief descriptions:

Chapter 2

Surflex-Protomol File Index

2.1 Surflex-Protomol File List

Here is a list of all files with brief descriptions:

Chapter 3

Surflex-Protomol Data Structure Documentation

3.1 bucket Struct Reference

`#include <hash.h>`

3.1.1 Detailed Description

A hash table consists of an array of these buckets.

Each bucket holds a copy of the key, a pointer to the data associated with the key, and a pointer to the next bucket that collided with this one, if there was one.

Data Fields

- char * key

  Key that hashes to this bucket.

- void * data

  Data paired with this key.

- struct bucket * next

  Linked list of collisions on this key.

3.1.2 Field Documentation

3.1.2.1 char* bucket::key

Key that hashes to this bucket.

3.1.2.2 void* bucket::data

Data paired with this key.

3.1.2.3 struct bucket* bucket::next [read]

Linked list of collisions on this key.

The documentation for this struct was generated from the following file:

- hash.h

3.2 HashIter Struct Reference

`#include <hash.h>`

3.2.1 Detailed Description

Iterator data structure for traversing the hashtable.

Initialize using HashNewIterator(). Useful in while loops with HashIterateNext().

Data Fields

- bucket * next
  Next bucket.
- unsigned int index
  Current position in bucket[] of HashTable.
- HashTable * ht
  Iterator initialized to this HashTable.
- char * key
  Current key in iteration.
- void * val
  Current value in iteration.

3.2.2 Field Documentation

3.2.2.1 bucket* HashIter::next

Next bucket.

3.2.2.2 unsigned int HashIter::index

Current position in bucket[] of HashTable.

3.2.2.3 HashTable* HashIter::ht

Iterator initialized to this HashTable.

3.2.2.4 char* HashIter::key

Current key in iteration.

3.2.2.5 void* HashIter::val

Current value in iteration.

The documentation for this struct was generated from the following file:

- hash.h

3.3 HashTable Struct Reference

`#include <hash.h>`

3.3.1 Detailed Description

Stores information related to a hash table.

This is what you actually declare an instance of to create a table. You then call 'construct_table' with the address of this structure, and a guess at the size of the table. Note that more nodes than this can be inserted in the table, but performance degrades as this happens. Performance should still be quite adequate until 2 or 3 times as many nodes have been inserted as the table was created with.

Data Fields

- size_t size

  Initial guess of HashTable size in number of buckets.

- int currSize

  Current size of HashTable in number of buckets.

- bucket ** table

  Array of buckets.

3.3.2 Field Documentation

3.3.2.1 size_t HashTable::size

Initial guess of HashTable size in number of buckets.

3.3.2.2 int HashTable::currSize

Current size of HashTable in number of buckets.

3.3.2.3 bucket** HashTable::table

Array of buckets.

The documentation for this struct was generated from the following file:

- hash.h

3.4 Probe Struct Reference

`#include <protomol-types.h>`

3.4.1 Detailed Description

Stores information pertaining to a probe.

Adds 3 fields to the conformer data structure.

See also:

Conformer

Data Fields

- Conformer * conf
  Probe conformation.
- int * protInter
  Array of protein atoms interacting with this probe.
- double * protScore
  Array of protein atom interaction scores.
- int type
  Defines the probe type.
- double alive
  If positive, probe is alive. If negative, probe is dead.
- int id
  Unique probe ID assigned by sanityCheckProbes().

3.4.2 Field Documentation

3.4.2.1 Conformer* Probe::conf

Probe conformation.

3.4.2.2 int* Probe::protInter

Array of protein atoms interacting with this probe.

Contains indices into molecule->atoms[]. End of array is marked by sentinel value of -1.

3.4.2.3 double* Probe::protScore

Array of protein atom interaction scores.

Has identical ordering as in protInter[].

3.4.2.4 int Probe::type

Defines the probe type.

See also:

SMALL_PROBE
BIG_PROBE.

3.4.2.5 double Probe::alive

If positive, probe is alive. If negative, probe is dead.

3.4.2.6 int Probe::id

Unique probe ID assigned by sanityCheckProbes().

See also:

sanityCheckProbes().

The documentation for this struct was generated from the following file:

- protomol-types.h

3.5 ProbeSet Struct Reference

`#include <protomol-types.h>`

3.5.1 Detailed Description

Store probes that are talking to a given set of protein atoms.

Useful for arbitrating protein-centric elimination of redundant probes.

Data Fields

- int * pas
  Array of protein atoms of interest (sorted indices into molecule->atom[]).
- int numPAs
  Number of elements in pas[].
- Probe * probes [MAX_INTER]
  Array of distinct probes that talk to our protein atoms.
- int currSize
  Number of elements in probes[].
- Probe * bestProbe
  Best probe in this set, NULL if not defined.

3.5.2 Field Documentation

3.5.2.1 int* ProbeSet::pas

Array of protein atoms of interest (sorted indices into molecule->atom[]).

3.5.2.2 int ProbeSet::numPAs

Number of elements in pas[].

3.5.2.3 Probe* ProbeSet::probes[MAX_INTER]

Array of distinct probes that talk to our protein atoms.

3.5.2.4 int ProbeSet::currSize

Number of elements in probes[].

3.5.2.5 Probe* ProbeSet::bestProbe

Best probe in this set, NULL if not defined.

The documentation for this struct was generated from the following file:

- protomol-types.h

Chapter 4

Surflex-Protomol File Documentation

4.1 dock-main-proto.c File Reference

4.1.1 Detailed Description

Command line entry point into surflex-proto.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "../sflib/surflex-public.h"
#include "protomol.h"
```

Functions

- int main (int argc, char **argv)

  Command line argument handler fdaf.

Variables

- int _FANCY_PROTO_P = 0

  If on, construct enhanced protomols.

- int _SOLV_PROTO_P = 0

  If on, construct solvation (water-based) protomols.

4.1.2 Function Documentation

4.1.2.1 int main (int *argc*, char ** *argv*)

Command line argument handler fdaf.

Parameters:

argc Number of arguments

argv Array of arguments

Returns:

Exit code

4.1.3 Variable Documentation

4.1.3.1 int _FANCY_PROTO_P = 0

If on, construct enhanced protomols.

Toggles on fancy protomol generation.

4.1.3.2 int _SOLV_PROTO_P = 0

If on, construct solvation (water-based) protomols.

Toggles on solvation protomol generation.

4.2 hash.c File Reference

4.2.1 Detailed Description

HashTable code.

Public domain code by Jerry Coffin, with improvements by HenkJan Wolthuis.

```
#include <string.h>
#include <stdlib.h>
#include "hash.h"
```

Functions

- HashTable * HashConstructTable (HashTable *table, size_t size)

  Initialize the HashTable to the size asked for.

- static unsigned hash (const char *ptr)
- void * HashInsert (char *key, void *data, HashTable *table)

  Insert 'key' into hash table.

- void * HashLookup (char *key, HashTable *table)

  Returns a pointer to the data associated with a key.

- void * HashDel (char *key, HashTable *table)

  Deletes an entry from the table.

- void HashFreeTable (HashTable *table, void(*func)(void *))

  Frees a hash table.

- void HashEnumerate (HashTable *table, void(*func)(char *, void *))

  Goes through a hash table and calls the function passed to it for each node that has been inserted.

- char ** HashGetKeys (HashTable *table)

  Enumerates through all keys in the hashtable, returning an array of char's (keys), of size table->currSize.*

- HashIter * HashNewIterator (HashTable *ht)

  HashIter constructor.

- void * HashIterateNext (HashIter *hi)

  Method for traversing to the next key->value pair in the hashtable.

- double HashDoublePlusPlus (HashTable *ht, char *key)

  Increment by one the int value stored for the given key.

- double HashDoubleMinusMinus (HashTable *ht, char *key)

  Decrement by one the double value stored for the given key.

- int HashPlusPlus (HashTable *ht, char *key)

  Add one to the int value stored for the given key.

- HashTable * LoadHashDouble (char *filename)

  Create a hashtable from the given file.

4.2.2 Function Documentation

4.2.2.1 static unsigned hash (const char * *ptr*) `[static]`

4.2.2.2 HashTable* HashConstructTable (HashTable * *table*, size_t *size*)

Initialize the HashTable to the size asked for.

This is used to construct the table.

Allocates space for the correct number of pointers and sets them to NULL. If it can't allocate sufficient memory, signals error by setting the size of the table to 0.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

4.2.2.3 void* HashDel (char * *key*, struct HashTable * *table*)

Deletes an entry from the table.

Returns a pointer to the data that was associated with the key so the calling code can dispose of it properly.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.2.2.4 double HashDoubleMinusMinus (HashTable * *ht*, char * *key*)

Decrement by one the double value stored for the given key.

If no value exists for this key, initialize it to -1.

Parameters:

ht
key

Returns:

The newly incremented value

4.2.2.5 double HashDoublePlusPlus (HashTable * *ht*, char * *key*)

Increment by one the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht
key

Returns:

The newly incremented value

4.2.2.6 void HashEnumerate (struct HashTable * *table*, void(*)(char *, void *) *func*)

Goes through a hash table and calls the function passed to it for each node that has been inserted.

The function is passed a pointer to the key, and a pointer to the data associated with it.

Parameters:

table HashTable
func Function to call on user data

4.2.2.7 void HashFreeTable (HashTable * *table*, void(*)(void *) *func*)

Frees a hash table.

For each node that was inserted in the table, it calls the function whose address it was passed, with a pointer to the data that was in the table. The function is expected to free the data. Typical usage would be: free_table(&table, free); if the data placed in the table was dynamically allocated, or: free_table(&table, NULL); if not. (If the parameter passed is NULL, it knows not to call any function with the data.)

Parameters:

table HashTable

func Function to free user data

4.2.2.8 char** HashGetKeys (HashTable * *table*)

Enumerates through all keys in the hashtable, returning an array of char*'s (keys), of size table->currSize.

Parameters:

table Hashtable

Returns:

Newly allocated array of key strings

4.2.2.9 void* HashInsert (char * *key*, void * *data*, HashTable * *table*)

Insert 'key' into hash table.

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Returns pointer to old data associated with the key, if any, or NULL if the key wasn't in the table previously.

Parameters:

key Key string

data User data

table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

4.2.2.10 void* HashIterateNext (HashIter * *hi*)

Method for traversing to the next key->value pair in the hashtable.

Parameters:

hi A HashIter

Returns:

User data or NULL if all data has been returned by this HashIter.

4.2.2.11 void* HashLookup (char * *key*, struct HashTable * *table*)

Returns a pointer to the data associated with a key.

If the key has not been inserted in the table, returns NULL.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.2.2.12 HashIter* HashNewIterator (HashTable * *ht*)

HashIter constructor.

Parameters:

ht HashTable

Returns:

A newly allocated HashIter

4.2.2.13 int HashPlusPlus (HashTable * *ht*, char * *key*)

Add one to the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.2.2.14 HashTable* LoadHashDouble (char * *filename*)

Create a hashtable from the given file.

The file contents are a set of key->value pairs, one on each row in the following format: key[tab]value[newline]

Parameters:

filename A data file

Returns:

Newly allocated HashTable

4.3 hash.h File Reference

4.3.1 Detailed Description

HashTable public interface.

```
#include <stddef.h>
#include <stdio.h>
```

Data Structures

- struct bucket

  A hash table consists of an array of these buckets.

- struct HashTable

  Stores information related to a hash table.

- struct HashIter

  Iterator data structure for traversing the hashtable.

Functions

- HashTable * HashConstructTable (HashTable *table, size_t size)

  This is used to construct the table.

- void * HashInsert (char *key, void *data, struct HashTable *table)

  Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

- void * HashLookup (char *key, struct HashTable *table)

  Returns a pointer to the data associated with a key.

- void * HashDel (char *key, struct HashTable *table)

  Deletes an entry from the table.

- void HashEnumerate (struct HashTable *table, void(*func)(char *, void *))

  Goes through a hash table and calls the function passed to it for each node that has been inserted.

- char ** HashGetKeys (HashTable *table)

  Enumerates through all keys in the hashtable, returning an array of char's (keys), of size table->currSize.*

- void HashFreeTable (HashTable *table, void(*func)(void *))

  Frees a hash table.

- HashIter * HashNewIterator (HashTable *ht)

  HashIter constructor.

- void * HashIterateNext (HashIter *hi)

  Method for traversing to the next key->value pair in the hashtable.

- int HashPlusPlus (HashTable *ht, char *key)

  Add one to the int value stored for the given key.

- double HashDoublePlusPlus (HashTable *ht, char *key)

  Increment by one the int value stored for the given key.

- double HashDoubleMinusMinus (HashTable *ht, char *key)

  Decrement by one the double value stored for the given key.

- HashTable * LoadHashDouble (char *filename)

  Create a hashtable from the given file.

4.3.2 Function Documentation

4.3.2.1 HashTable* HashConstructTable (HashTable * *table*, size_t *size*)

This is used to construct the table.

If it doesn't succeed, it sets the table's size to 0, and the pointer to the table to NULL.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

This is used to construct the table.

Allocates space for the correct number of pointers and sets them to NULL. If it can't allocate sufficient memory, signals error by setting the size of the table to 0.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

4.3.2.2 void* HashDel (char * *key*, struct HashTable * *table*)

Deletes an entry from the table.

Returns a pointer to the data that was associated with the key so the calling code can dispose of it properly.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.3.2.3 double HashDoubleMinusMinus (HashTable * *ht*, char * *key*)

Decrement by one the double value stored for the given key.

If no value exists for this key, initialize it to -1.

Parameters:

ht

key

Returns:

The newly incremented value

4.3.2.4 double HashDoublePlusPlus (HashTable * *ht*, char * *key*)

Increment by one the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.3.2.5 void HashEnumerate (struct HashTable * *table*, void(*)(char *, void *) *func*)

Goes through a hash table and calls the function passed to it for each node that has been inserted.

The function is passed a pointer to the key, and a pointer to the data associated with it.

Parameters:

table HashTable

func Function to call on user data

4.3.2.6 void HashFreeTable (HashTable * *table*, void(*)(void *) *func*)

Frees a hash table.

For each node that was inserted in the table, it calls the function whose address it was passed, with a pointer to the data that was in the table. The function is expected to free the data. Typical usage would be: free_table(&table, free); if the data placed in the table was dynamically allocated, or: free_table(&table, NULL); if not. (If the parameter passed is NULL, it knows not to call any function with the data.)

Parameters:

table HashTable

func Function to free user data

4.3.2.7 char** HashGetKeys (HashTable * *table*)

Enumerates through all keys in the hashtable, returning an array of char*'s (keys), of size table->currSize.

Parameters:

table Hashtable

Returns:

Newly allocated array of key strings

4.3.2.8 void* HashInsert (char * *key*, void * *data*, HashTable * *table*)

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Note that this makes a copy of the key, but NOT of the associated data.

Parameters:

key Key string

data User data

table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Returns pointer to old data associated with the key, if any, or NULL if the key wasn't in the table previously.

Parameters:

key Key string

data User data

table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

4.3.2.9 void* HashIterateNext (HashIter * *hi*)

Method for traversing to the next key->value pair in the hashtable.

Parameters:

hi A HashIter

Returns:

User data or NULL if all data has been returned by this HashIter.

4.3.2.10 void* HashLookup (char * *key*, struct HashTable * *table*)

Returns a pointer to the data associated with a key.

If the key has not been inserted in the table, returns NULL.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.3.2.11 HashIter* HashNewIterator (HashTable * *ht*)

HashIter constructor.

Parameters:

ht HashTable

Returns:

A newly allocated HashIter

4.3.2.12 int HashPlusPlus (HashTable * *ht*, char * *key*)

Add one to the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.3.2.13 HashTable* LoadHashDouble (char * *filename*)

Create a hashtable from the given file.

The file contents are a set of key->value pairs, one on each row in the following format: key[tab]value[newline]

Parameters:

filename A data file

Returns:

Newly allocated HashTable

4.4 protomol-types.h File Reference

4.4.1 Detailed Description

Enhanced protomol data structures.

Data Structures

- struct Probe

  Stores information pertaining to a probe.

- struct ProbeSet

  Store probes that are talking to a given set of protein atoms.

Defines

- #define MAX_INTER 1000

  Maximum number of interactions saved in a ProbeSet.

4.4.2 Define Documentation

4.4.2.1 #define MAX_INTER 1000

Maximum number of interactions saved in a ProbeSet.

4.5 protomol.c File Reference

4.5.1 Detailed Description

Enhanced protomol code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <math.h>
#include "../sflib/surflex-public.h"
#include "hash.h"
#include "protomol.h"
#include "utils.h"
```

Functions

- int createProbe (Probe *probe, Conformer *protein, Grid *grid, Vector3 v, Probe **probeGrp, double initThresh, double polarThresh, double bumpThresh, double *zsumInit, double *zsumPolar, double *zsumBump, double *count, Probe **bestRMS, int *numRMS, int samplingFlag, HashTable *probeSets, HashTable *bigSets)

  Try to make a probe in the given point in space.

- Probe * make_probe_steric ()

  Make a steric probe (CH3) suitable for use with createProbe().

- Probe * make_probe_acceptor ()

  Make an acceptor probe (C=O) suitable for use with createProbe().

- Probe * make_probe_donor ()

  Make a donor probe (N-H) suitable for use with createProbe().

- Probe * make_probe_methanol ()

  Make a methanol probe (CH3-OH) suitable for use with createProbe().

- Probe * make_probe_carboxylicAcid ()

  Make a carboxyl probe (CH3-COO) suitable for use with createProbe().

- Probe * make_probe_amidine ()

  Make an amidine probe (CH3-C-(NH2)2) suitable for use with createProbe().

- Probe * make_probe_amnT ()

  Make an amnT probe (C-(NH)2) suitable for use with createProbe().

- Probe * make_probe_amnY ()

  Make an amnY probe (CH3-C(NH2)2) suitable for use with createProbe().

- Probe * make_probe_water ()

  Make a water probe (H2O) suitable for use with createProbe().

- Probe ** getBestProbes (HashTable *probeSets, HashTable *bigSets, int *numBest)

  Retrieve the bestProbes stored in each set in the hashtable.

- void findBestProbeInSet (char *paStr, void *data)

  Assigns set->bestProbe field.

- void byScore (ProbeSet *set)

  Choose probe with greatest scores.

- void byPolarScore (ProbeSet *set)

  Choose probe with greatest polar score.

- void byProtInterSum (ProbeSet *set)

  Choose probe with greatest sum of interaction scores (probe->protScore).

- int collect_interact_prots (Probe *probe, Conformer *protein, int find_close_p, Grid *grid, HashTable *probeSets, HashTable *bigSets)

  Score the polar interaction b/w the given probe & protein.

- int isGoodBigProbe (Conformer *probe, HashTable *uniquePA, HashTable *uniqueLA)

  Test that this big probe is worthy of being saved.

- int addProbePASet (HashTable *probeSets, HashTable *bigSets, HashTable *uniquePA, Probe *probe)

  Track which probes and protein atoms (paSet) are talking to one another.

- void sanityCheckProbes (Probe **probes, int numProbes, int doSort)

  qsorts list of probes, assigns to each probe id a ranking number, pretty print out probe information.

- void proteinCentricElim (Probe **probes, int numProbes, Conformer *protein, HashTable *probeSets)

  Eliminate redundant probes from protein's perspective.

- void rmsElim (Probe **probes, int numProbes)

  Given array of probes, remove those that are similar to each other by rms.

- double rms_probe (Conformer *conf1, Conformer *conf2)

  Calculate RMSD between two conformers.

- int writeProbes (Probe **probes, int numProbes, char *prefix)

  Write probes out to disk.

- int mergeProbes (Probe **probes, int numProbes, char *prefix, Molecule **protomol)

  Merges probes on disk into protomol.

- void mergeProbesInMemory (Probe **probes, int numProbes, char *prefix, Molecule **protomol, int allProbes)

  Merges probes in memory into protomol.

- void deleteProbes (Probe **probes, int numProbes, char *prefix)

  Delete probe files on disk.

- void freeProbeSet (void *data)

  Free a ProbeSet and its contents.

- void freeProbes (Probe **probes, int numProbes)

  Free an array of probes.

- void freeProbe (Probe *p)

  Free a probe and its contents.

- void verifyProtInterStorage (Probe **probes, int numProbes, char *prefix)

  Print out basic information for each probe in the array.

- void printProbeStats (char *name, double sumInit, double initCt, double sumPolar, double sumBump, double polarCt)

  Helper function prints out mean stats.

- void printContents (char *key, void *data)

  Printer function for use with HashEnumerate().

- int compProbeByScore (const void *probe1, const void *probe2)

  qsort compare function for probes.

- int compProbePA (const void *paStr1, const void *paStr2)

  qsort compare function for paStr created using collapseStr().

- void markProbeAlive (Probe *p, double value)

  Mark this probe as alive.

- void markProbeDead (Probe *p, double value)

  Mark this probe as dead.

- int isProbeAlive (Probe *p)

  Check if this probe is still alive.

- double getProbeMark (Probe *p)

  Get this probe's mark.

- void collapseStr (char **strArr, int arrSize, char *str, int buffSize, int *intArr)

  Given a char array, collapse all contents of the array into one long string.

- char * intToStr (int num)

  Convert int to str (MAX=5 chars).

- void make_protomol_wrapper (char *ligpath, char *protpath, char *name)

  Wrapper function for making protomols.

- void make_solvmol (Conformer *protein, Grid *grid, char *name, Conformer *ligand)

  Ligand + residue-based protomol generator using only water probes.

- void make_fancy_protomol (Conformer *protein, Grid *grid, char *name, Conformer *ligand)

  Ligand + residue-based protomol generator using a variety of small and big probes.

Variables

- int _FANCY_PROTO_P

  Toggles on fancy protomol generation.

- int _SOLV_PROTO_P

  Toggles on solvation protomol generation.

- double sf_poz

  Surflex parameter: Polar Gaussian attraction scale factor.

- double sf_stz

  Surflex parameter: Steric Gaussian attraction scale factor.

- double sf_pom

  Surflex parameter: Polar Gaussian location.

- double sf_pos

  Surflex parameter: Polar Gaussian spread.

- double sf_srm

  Surflex parameter: Polar sigmoid inflection point.

- double sf_por

  Surflex parameter: Polar sigmoid repulsion scale factor.

- double polar_bump_thresh

  Surflex parameter: VdW allowance for hard clashing (polar).

- Contact lig_contacts [1000]

  Cached nearby protein contacts to the ligand.

4.5.2 Function Documentation

4.5.2.1 int addProbePASet (HashTable * *probeSets*, HashTable * *bigSets*, HashTable * *uniquePA*, Probe * *probe*)

Track which probes and protein atoms (paSet) are talking to one another.

- Protein-centric: Add this probe's paSet to the collective probeSets hash that takes care of overlapping set matching.
- Probe-centric: add this paSet to the HashTable bigSets if it interacts with more than 1 protein atom.

Parameters:

probeSets protein-centric hash tracks which paSet->probes.

bigSets probe-centric hash tracks protein atoms that are talking to big probes.

uniquePA HashTable contains all protein atoms this probe talks to.

probe A probe of interest.

Returns:

True if we've encountered a unique paSet

4.5.2.2 void byPolarScore (ProbeSet * *set*)

Choose probe with greatest polar score.

Helper function to findBestProbeInSet(). ASSUMES: all probes in set are of same type!!.

Currently only coded for probes of same type to compete (big vs big / small vs small). Need to think about how small competes with big...

Parameters:

set A probe set.

4.5.2.3 void byProtInterSum (ProbeSet * *set*)

Choose probe with greatest sum of interaction scores (probe->protScore).

Helper function to findBestProbeInSet().

Parameters:

set A probe set.

4.5.2.4 void byScore (ProbeSet * *set*)

Choose probe with greatest scores.

Helper function to findBestProbeInSet().

Parameters:

set A probe set.

4.5.2.5 void collapseStr (char ** *strArr*, int *arrSize*, char * *str*, int *buffSize*, int * *intArr*)

Given a char array, collapse all contents of the array into one long string.

Parameters:

strArr String to collapse.

arrSize String length.

str An allocated buffer in which to store the collapsed string.

buffSize The maximum size of the buffer str.

intArr An allocated int[] in which to store the parsed contents of strArr. Assumes the strArr contains only integers.

4.5.2.6 int collect_interact_prots (Probe * *probe*, Conformer * *protein*, int *find_close_p*, Grid * *grid*, HashTable * *probeSets*, HashTable * *bigSets*)

Score the polar interaction b/w the given probe & protein.

Those which score well are deemed good interactions. Store good interactions in two ways:

- Protein-centric: which probes are talking to which protein atoms?
- Probe-centric: which protein atoms are talking to which big probes?

Parameters:

probe A docked probe whose current conformation will be checked for good interactions with the receptor site.

protein A protein receptor site.

find_close_p To simplify computation, toggles scoring only those protein atoms close to the probe.

grid A grid caching all interesting contacts within the protein binding pocket.

probeSets HashTable tracking which probes are talking to which protein atoms.

bigSets HashTable tracking which protein atoms are talking to big probes.

Returns:

Number of good interactions found for this probe conformation.

4.5.2.7 int compProbeByScore (const void * *probe1*, const void * *probe2*)

qsort compare function for probes.

Parameters:

probe1 A probe.

probe2 A probe.

Returns:

-1 if probe1 > probe2, 0 if probe1 = probe2, 1 if probe1 < probe2.

4.5.2.8 int compProbePA (const void * *paStr1*, const void * *paStr2*)

qsort compare function for paStr created using collapseStr().

Parameters:

paStr1 A paStr.

paStr2 A paStr.

Returns:

-1 if probe1 > probe2, 0 if probe1 = probe2, 1 if probe1 < probe2.

4.5.2.9 int createProbe (Probe * *probe*, Conformer * *protein*, Grid * *grid*, Vector3 *v*, Probe ** *probeGrp*, double *initThresh*, double *polarThresh*, double *bumpThresh*, double * *zsumInit*, double * *zsumPolar*, double * *zsumBump*, double * *count*, Probe ** *bestRMS*, int * *numRMS*, int *samplingFlag*, HashTable * *probeSets*, HashTable * *bigSets*)

Try to make a probe in the given point in space.

A probe is placed at the given voxel and sampled

Parameters:

probe A probe to rotationally sample at this voxel.

protein A protein receptor site.

grid A grid caching all interesting contacts within the protein binding pocket.

v Vector3 point in space where we will attempt to place the probe.

probeGrp Array to store our new probe if successful.

initThresh Minimum score threshold for viable probe.
polarThresh Minimum polar score threshold for viable probe.
bumpThresh Minimum bump score threshold for viable probe.
zsumInit Tracking this probe group's avg score.
zsumPolar Tracking this probe group's avg polar score.
zsumBump Tracking this probe group's avg bump score.
count Number of probes tried.
bestRMS Probe array with best RMSD to a target probe (minConf).
numRMS Number of probes in bestRMS[].
samplingFlag Output all sampled probe conformations for this point.
probeSets HashTable tracks all ProbeSets talking to which protein atoms.
bigSets HashTable tracks which protein atoms belong to big probes.

Returns:

Number of good probes found.

4.5.2.10 void deleteProbes (Probe ∗∗ *probes*, int *numProbes*, char ∗ *prefix*)

Delete probe files on disk.

Parameters:

probes Array of probes to delete.
numProbes Number of probes in array.
prefix Prefix string to probe names.

4.5.2.11 void findBestProbeInSet (char ∗ *paStr*, void ∗ *data*)

Assigns set->bestProbe field.

Used with HashEnumerate() to traverse a probeSets HashTable in order to locate the best probe for the given set of protein atoms.

Parameters:

paStr A collapsed string of protein atoms.

See also:

collapseStr().

Parameters:

data A probeSet.

4.5.2.12 void freeProbe (Probe * *p*)

Free a probe and its contents.

Parameters:

p A probe.

4.5.2.13 void freeProbes (Probe ** *probes*, int *numProbes*)

Free an array of probes.

Does not free the array itself.

Parameters:

probes Array of Probe pointers.

numProbes Number of probes in array.

4.5.2.14 void freeProbeSet (void * *data*)

Free a ProbeSet and its contents.

Parameters:

data A probeSet. Given type void * so this function can be passed to HashFreeTable().

4.5.2.15 Probe ** getBestProbes (HashTable * *probeSets*, HashTable * *bigSets*, int * *numBest*)

Retrieve the bestProbes stored in each set in the hashtable.

If we're looking at a small probe, verify that its set does not overlap any other.

Parameters:

probeSets protein-centric hash tracks which paSet->probes.

bigSets probe-centric hash tracks protein atoms that are talking to big probes.

numBest number of probes returned in array.

Returns:

Array of all the best probes.

4.5.2.16 double getProbeMark (Probe * *p*)

Get this probe's mark.

Parameters:

p A probe.

Returns:

The value stored as this probe's mark.

4.5.2.17 char * intToStr (int *num*)

Convert int to str (MAX=5 chars).

Parameters:

num Integer to convert to string.

Returns:

Allocated string representation of num.

4.5.2.18 int isGoodBigProbe (Conformer * *probe*, HashTable * *uniquePA*, HashTable * *uniqueLA*)

Test that this big probe is worthy of being saved.

This probe must satisfy the following:

- Interaction with 2+ unique protein atoms.
- Interaction with 2+ unique probe atoms.
- Amidine probes must interact with 3+ unique protein atoms.
- 2 interactions must be "good" as defined by collect_inter_prot().

Returns:

True if the probe is good.

4.5.2.19 int isProbeAlive (Probe * *p*)

Check if this probe is still alive.

Parameters:

p A probe.

Returns:

1 if probe is alive; 0 if probe is dead.

4.5.2.20 void make_fancy_protomol (Conformer * *protein*, Grid * *grid*, char * *name*, Conformer * *ligand*)

Ligand + residue-based protomol generator using a variety of small and big probes.

Fancy protomol generator. Does redundancy elimination. Protomols generated from several probe types:

- Small probes

  See also:

  – make_probe_steric()

  See also:

  – make_probe_donor()

  See also:

  – make_probe_acceptor()

- Large probes

  See also:

  – make_probe_carboxylicAcid()

  See also:

  – make_probe_amnT()

  See also:

  – make_probe_amnY()

- Optional probes

  See also:

  – make_probe_methanol()

  See also:

  – make_probe_amidine()

4.5.2.21 Probe * make_probe_acceptor ()

Make an acceptor probe (C=O) suitable for use with createProbe().

Used to sample potential hydrogen bond donors within the protein pocket.

Returns:

An acceptor probe.

4.5.2.22 Probe * make_probe_amidine ()

Make an amidine probe (CH3-C-(NH2)2) suitable for use with createProbe().

Used to sample interactions available to a ligand's amidine functional group, particularly multiple (2+) interactions.

Returns:

A amidine probe.

4.5.2.23 Probe * make_probe_amnT ()

Make an amnT probe (C-(NH)2) suitable for use with createProbe().

Used to sample specific interactions available to a ligand's amidine functional group, particularly bidentate interactions.

Returns:

An amnT probe.

4.5.2.24 Probe * make_probe_amnY ()

Make an amnY probe (CH3-C(NH2)2) suitable for use with createProbe().

Used to sample specific interactions available to a ligand's amidine functional group, particularly multiple bidentate interactions.

Returns:

An amnY probe.

4.5.2.25 Probe * make_probe_carboxylicAcid ()

Make a carboxyl probe (CH3-COO) suitable for use with createProbe().

Used to sample interactions available to a ligand's carboxyl functional group, particularly bidentate interactions.

Returns:

A carboxyl probe.

4.5.2.26 Probe * make_probe_donor ()

Make a donor probe (N-H) suitable for use with createProbe().

Used to sample potential hydrogen bond acceptors within the protein pocket.

Returns:

A donor probe.

4.5.2.27 Probe * make_probe_methanol ()

Make a methanol probe (CH3-OH) suitable for use with createProbe().

Used to sample interactions available to a ligand's hydroxyl functional group.

Returns:

A methanol probe.

4.5.2.28 Probe * make_probe_steric ()

Make a steric probe (CH3) suitable for use with createProbe().

Used to sample the shape of a binding pocket.

Returns:

A steric probe.

4.5.2.29 Probe * make_probe_water ()

Make a water probe (H2O) suitable for use with createProbe().

Used to create a solvation protomol that models possible interactions with water in the protein pocket.

Returns:

A water probe.

4.5.2.30 void make_protomol_wrapper (char * *ligpath*, char * *protpath*, char * *name*)

Wrapper function for making protomols.

Performs preprocessing necessary before calling actual protomol making functions.

- Read in the protein and ligand.
- Setup the grid
- Mark the relevant protein atoms

Parameters:

ligpath a ligand file path
protpath a protein file path
name output protomol's name

4.5.2.31 void make_solvmol (Conformer * *protein*, Grid * *grid*, char * *name*, Conformer * *ligand*)

Ligand + residue-based protomol generator using only water probes.

Creates a protomol solely from water probes. Uses redundancy elimination.

Parameters:

protein a protein conformer

grid A grid caching all interesting contacts within the protein binding pocket.

name the output protomol's name

ligand a model ligand conformation whose interactions with the binding site will serve as a model for protomol generation

4.5.2.32 void markProbeAlive (Probe * *p*, double *value*)

Mark this probe as alive.

Any non-negative, double value can be used to mark this probe.

Parameters:

p A probe.

value Some positive, sentinel value.

4.5.2.33 void markProbeDead (Probe * *p*, double *value*)

Mark this probe as dead.

Any negative, double value can be used to mark this probe. Overrides the conformer->data[9] field to store the mark.

See also:

getProbeMark().

Parameters:

p A probe.

value Some negative, sentinel value.

4.5.2.34 int mergeProbes (Probe ** *probes*, int *numProbes*, char * *prefix*, Molecule ** *protomol*)

Merges probes on disk into protomol.

Should combo writeProbes + mergeProbes to do this in memory.

See also:

mergeProbesInMemory()

Parameters:

probes An array of probes.

numProbes Number of probes in array.

prefix Prefix string to probe names.

protomol Location to place newly created protomol from merged probes.

Returns:

Number of probes merged

4.5.2.35 void mergeProbesInMemory (Probe ∗∗ *probes*, int *numProbes*, char ∗ *prefix*, Molecule ∗∗ *protomol*, int *allProbes*)

Merges probes in memory into protomol.

Parameters:

probes An array of probes.

numProbes Number of probes in array.

prefix Prefix string for these probes.

protomol Location to place newly created protomol from merged probes.

allProbes Toggles on merging of all probes in array into protomol. Usually, probes are filtered so as not to include those marked for death.

Returns:

Number of probes merged

4.5.2.36 void printContents (char ∗ *key*, void ∗ *data*)

Printer function for use with HashEnumerate().

Assumes data is int∗

Parameters:

key Hashed key.

data Value hashed to key.

4.5.2.37 void printProbeStats (char ∗ *name*, double *sumInit*, double *initCt*, double *sumPolar*, double *sumBump*, double *polarCt*)

Helper function prints out mean stats.

Parameters:

name Name of object whose stats we're outputting

sumInit Sum of scores

initCt Number of scores to average

sumPolar Sum of polar scores

sumBump Sum of bump scores

polarCt Number of polar/bump scores

4.5.2.38 void proteinCentricElim (Probe ** *probes*, int *numProbes*, Conformer * *protein*, HashTable * *probeSets*)

Eliminate redundant probes from protein's perspective.

Each protein atom accepts 1 probe, best by total score. Essential that this comes after seeding best_probes array.

Parameters:

probes An array of probes.

numProbes Number of probes in array.

protein A protein receptor site.

probeSets protein-centric hash tracks which paSet->probes.

4.5.2.39 double rms_probe (Conformer * *conf1*, Conformer * *conf2*)

Calculate RMSD between two conformers.

Parameters:

conf1 A conformer.

conf2 A conformer.

Returns:

RMSD.

4.5.2.40 void rmsElim (Probe ** *probes*, int *numProbes*)

Given array of probes, remove those that are similar to each other by rms.

Always keep higher scoring probe.

Parameters:

probes An array of probes.

numProbes Number of probes in array.

4.5.2.41 void sanityCheckProbes (Probe ** *probes*, int *numProbes*, int *doSort*)

qsorts list of probes, assigns to each probe id a ranking number, pretty print out probe information.

Probe id is useful for writing probes to disk with unique names.

Parameters:

probes An array of probes.

numProbes Number of probes in array.

doSort Toggles on qsort of probe list in place.

4.5.2.42 void verifyProtInterStorage (Probe ** *probes*, int *numProbes*, char * *prefix*)

Print out basic information for each probe in the array.

Parameters:

probes An array of probes.

numProbes Number of probes in array.

prefix Prefix string to probe names.

4.5.2.43 int writeProbes (Probe ** *probes*, int *numProbes*, char * *prefix*)

Write probes out to disk.

IMPORTANT: each probe should have a distinct # stored in probes[i]->id.

See also:

sanityCheckProbes().

Parameters:

probes An array of probes.

numProbes Number of probes in array.

prefix Prefix string to probe names.

Returns:

Number of probes written.

4.5.3 Variable Documentation

4.5.3.1 int _FANCY_PROTO_P

Toggles on fancy protomol generation.

4.5.3.2 int _SOLV_PROTO_P

Toggles on solvation protomol generation.

4.5.3.3 Contact lig_contacts[1000]

Cached nearby protein contacts to the ligand.

4.5.3.4 double polar_bump_thresh

Surflex parameter: VdW allowance for hard clashing (polar).

4.5.3.5 double sf_pom

Surflex parameter: Polar Gaussian location.

4.5.3.6 double sf_por

Surflex parameter: Polar sigmoid repulsion scale factor.

4.5.3.7 double sf_pos

Surflex parameter: Polar Gaussian spread.

4.5.3.8 double sf_poz

Surflex parameter: Polar Gaussian attraction scale factor.

4.5.3.9 double sf_srm

Surflex parameter: Polar sigmoid inflection point.

4.5.3.10 double sf_stz

Surflex parameter: Steric Gaussian attraction scale factor.

4.6 protomol.h File Reference

4.6.1 Detailed Description

Enhanced protomol public interface.

```
#include "protomol-types.h"
```

Defines

- #define PROBE_ROTATIONAL_SAMPLING 4.0

  Samping rate for each rotational degree of freedom. (6.0 for small probes).

- #define PROTEIN_INTERACT_THRESH 0.50

  Minimum polar interaction for a probe.

- #define GOOD_POLAR_CONTACT_THRESH 0.90

  Minimum polar interaction for a protein atom.

- #define GOOD_POLAR_CONTACT 1

  Sentinel value identifies good contacts within the uniquePA hashtable.

- #define MAX_PROBE_RMS 0.75

  Maximum rmsd for two probes to be considered identical.

- #define MAX_PROTATOM_INTER 100

  Maximum number of interactions saved for a single protein atom.

- #define MAX_INTER 1000

  Maximum number of interactions saved in a ProbeSet.

- #define MAX_CONTACTS 200

  Maximum number of interesting contacts saved for a single atom.

- #define SMALL_PROBE 1

  Identifies probe->type as small.

- #define BIG_PROBE 2

  Identifies probe->type as big.

- #define STERIC 0

  Identifies molecule->atom->type as steric.

- #define ACCEPTOR 1

  Identifies molecule->atom->type as acceptor.

- #define DONOR 2

  Identifies molecule->atom->type as donor.

- #define SYMM 7

  Identifies molecule->bond->type as symmetric.

- #define USE_SMALL_PROBES 1

  Toggles on use of small probes in make_fancy_protomol().

- #define USE_BIG_PROBES 1

  Toggles on use of big probes in make_fancy_protomol().

- #define BIG 1000000

  Sentinel value for a big number (used for finding minimums).

- #define SMALL -1000000

  Sentinel value for a small number (used for finding maximums).

- #define PI 3.14159265

  Pi to the 8th decimal place.

- #define TWO_PI 6.28

  2Pi to the 2nd decimal place

- #define STT 10.0

  Scoring function parameter: Sigmoid steepness.

Functions

- void make_protomol_wrapper (char *ligpath, char *protpath, char *name)

  Wrapper function for making protomols.

- void make_solvmol (Conformer *protein, Grid *grid, char *name, Conformer *ligand)

  Ligand + residue-based protomol generator using only water probes.

- void make_fancy_protomol (Conformer *protein, Grid *grid, char *name, Conformer *ligand)

  Ligand + residue-based protomol generator using a variety of small and big probes.

4.6.2 Define Documentation

4.6.2.1 #define ACCEPTOR 1

Identifies molecule->atom->type as acceptor.

4.6.2.2 #define BIG 1000000

Sentinel value for a big number (used for finding minimums).

4.6.2.3 #define BIG_PROBE 2

Identifies probe->type as big.

4.6.2.4 #define DONOR 2

Identifies molecule->atom->type as donor.

4.6.2.5 #define GOOD_POLAR_CONTACT 1

Sentinel value identifies good contacts within the uniquePA hashtable.

4.6.2.6 #define GOOD_POLAR_CONTACT_THRESH 0.90

Minimum polar interaction for a protein atom.

4.6.2.7 #define MAX_CONTACTS 200

Maximum number of interesting contacts saved for a single atom.

See also:

Contact.

4.6.2.8 #define MAX_INTER 1000

Maximum number of interactions saved in a ProbeSet.

4.6.2.9 #define MAX_PROBE_RMS 0.75

Maximum rmsd for two probes to be considered identical.

4.6.2.10 #define MAX_PROTATOM_INTER 100

Maximum number of interactions saved for a single protein atom.

4.6.2.11 #define PI 3.14159265

Pi to the 8th decimal place.

4.6.2.12 #define PROBE_ROTATIONAL_SAMPLING 4.0

Samping rate for each rotational degree of freedom. (6.0 for small probes).

4.6.2.13 #define PROTEIN_INTERACT_THRESH 0.50

Minimum polar interaction for a probe.

4.6.2.14 #define SMALL -1000000

Sentinel value for a small number (used for finding maximums).

4.6.2.15 #define SMALL_PROBE 1

Identifies probe->type as small.

4.6.2.16 #define STERIC 0

Identifies molecule->atom->type as steric.

4.6.2.17 #define STT 10.0

Scoring function parameter: Sigmoid steepness.

4.6.2.18 #define SYMM 7

Identifies molecule->bond->type as symmetric.

4.6.2.19 #define TWO_PI 6.28

2Pi to the 2nd decimal place

4.6.2.20 #define USE_BIG_PROBES 1

Toggles on use of big probes in make_fancy_protomol().

4.6.2.21 #define USE_SMALL_PROBES 1

Toggles on use of small probes in make_fancy_protomol().

4.6.3 Function Documentation

4.6.3.1 void make_fancy_protomol (Conformer * *protein*, Grid * *grid*, char * *name*, Conformer * *ligand*)

Ligand + residue-based protomol generator using a variety of small and big probes.

Fancy protomol generator. Does redundancy elimination. Protomols generated from several probe types:

- Small probes

  See also:

  – make_probe_steric()

  See also:

  – make_probe_donor()

  See also:

  – make_probe_acceptor()

- Large probes

  See also:

  – make_probe_carboxylicAcid()

  See also:

  – make_probe_amnT()

See also:

- make_probe_amnY()

- Optional probes

  See also:

  - make_probe_methanol()

  See also:

  - make_probe_amidine()

4.6.3.2 void make_protomol_wrapper (char * *ligpath*, char * *protpath*, char * *name*)

Wrapper function for making protomols.

Performs preprocessing necessary before calling actual protomol making functions.

- Read in the protein and ligand.
- Setup the grid
- Mark the relevant protein atoms

Parameters:

ligpath a ligand file path
protpath a protein file path
name output protomol's name

4.6.3.3 void make_solvmol (Conformer * *protein*, Grid * *grid*, char * *name*, Conformer * *ligand*)

Ligand + residue-based protomol generator using only water probes.

Creates a protomol solely from water probes. Uses redundancy elimination.

Parameters:

protein a protein conformer

grid A grid caching all interesting contacts within the protein binding pocket.

name the output protomol's name

ligand a model ligand conformation whose interactions with the binding site will serve as a model for protomol generation

4.7 utils.c File Reference

4.7.1 Detailed Description

Utility functions code.

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

Functions

- void exitError (char *msg, int code)
  Exit with error msg and code.
- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)

Copy source file to target.

- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.
- double myRound (double num, int power)
  round the number to the given power of 10.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.

Variables

- int crlf_p
  Newline status: if newlines are "\\r\\n" then true.

4.7.2 Function Documentation

4.7.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.7.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message
code Error code

4.7.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.7.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.7.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file
src Source file to copy

4.7.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open
mode fopen mode

Returns:

Newly opened file pointer if successful

4.7.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.7.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.7.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.7.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) > maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.7.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.7.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.7.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.7.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.7.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.7.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");

- }

3. And after loop completes for pretty printing:

   - fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

4.7.3 Variable Documentation

4.7.3.1 int crlf_p

Newline status: if newlines are "\\r\\n" then true.

4.8 utils.h File Reference

4.8.1 Detailed Description

Utility functions public interface.

```
#include "stdio.h"
```

Functions

- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- void exitError (char *msg, int code)
  Exit with error msg and code.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- double myRound (double num, int power)
  round the number to the given power of 10.
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.

- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)
  Copy source file to target.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.

4.8.2 Function Documentation

4.8.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.8.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message

code Error code

4.8.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.8.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.8.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file

src Source file to copy

4.8.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open

mode fopen mode

Returns:

Newly opened file pointer if successful

4.8.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.8.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.8.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.8.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) > maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.8.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.8.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.8.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.8.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.8.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.8.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");
- }

3. And after loop completes for pretty printing:

- fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

Appendix C. Scoring Function Optimization

C.1.1. Usage

This section will detail the usage of surflex-opt on the command line. All option arguments shown are their default values.

Brief command / option description

General usage: **surflex-opt <options> <command>** args

Example command

OPTIMIZATION COMMANDS

Optimize the Surflex scoring function using the constraints *and initial* parameters. *All optimization output will be prefixed with* optID.

```
surflex-opt optimize optID constraints param
surflex-opt optimize hivpr screen.constraint default.param
```

OPTIMIZATION OPTIONS: Search Strategy

Use only line search (systematic perturbation).

```
surflex-opt -lineopt optimize optID constraints param
surflex-opt –lineopt optimize hivpr scrn.constr dft.prm
```

Use only random walk.

```
surflex-opt -randwalk optimize optID constraints param
surflex-opt –randwalk optimize hivpr scr.constr dft.prm
```

Use only bounded random search.

```
surflex-opt -randsearch optimize optID constraints param
surflex-opt –randsearch optimize hivpr scr.constr dft.prm
```

Set the parameter step size used with line search via the option argument N*:*
*Parameter step size = 1/*N ** initial param value*

```
surflex-opt -stepsize N optimize optID constraints param
surflex-opt -stepsize 100 optimize hivpr scr.constr dft.prm
```

Set the parameter step size used with random walk via the option argument N*:*
*Parameter step size = 1/*N ** initial param value*

```
surflex-opt -rw_stepsize N optimize optID constraints param
surflex-opt -rw_stepsize 10 optimize hivpr scr.constr dft.prm
```

Set the bounded range of random search via the option argument N*. Used in conjunction with the range assigned to a specific parameter. See Parameter file format.*

RangeType	*Lower bound*	*Upper bound*
None	*-½ ** N ** initial param value*	*½ ** N ** initial param value*]
Positive	*0*	N ** initial param value*
Negative	N ** initial param value*	*0*
> 1	*1*	N ** initial param value*
0 < p < 1	*0*	*1*

```
surflex-opt -paramrange N optimize optID constraints param
surflex-opt -paramrange 5 optimize hivpr scr.constr dft.prm
```

OPTIMIZATION OPTIONS: Stopping Conditions

Set the maximum number of optimization epochs.

```
surflex-opt -maxepochs N optimize optID constraints param
surflex-opt -maxepochs 100000 optimize hivpr scr.constr dft.prm
```

Set the maximum number of epochs of random walk without improvement.

```
surflex-opt -rw_maxepochs N optimize optID constraints param
surflex-opt -rw_maxepochs 200 optimize hivpr scr.constr dft.prm
```

Set the minimum mean squared error (MSE) goal for optimization performance.

```
surflex-opt -minmse N optimize optID constraints param
surflex-opt -minmse 0.0001 optimize hivpr scr.constr dft.prm
```

Set the maximum number of search cycles. One cycle consists of a random walk search followed by line search (systematic parameter perturbation).

```
surflex-opt -ncycle N optimize optID constraints param
surflex-opt -ncycle 2 optimize hivpr scr.constr dft.prm
```

OPTIMIZATION OPTIONS: Pose cache

Optimize all poses stored in the ligand pose cache after N *epochs of improvement.*

```
surflex-opt -ngood2optpose N optimize optID constraints param
surflex-opt -ngood2optpose 5 optimize hivpr scr.constr dft.prm
```

Set the number N *of poses to cache for each ligand during optimization.*

```
surflex-opt -posecachesize N optimize optID constraints param
surflex-opt -posecachesize 5 optimize hivpr scr.constr dft.prm
```

OPTIMIZATION OPTIONS: Miscellaneous

Score the data with the given `parameters` *and exit. No parameter optimization takes place. Quick method for doing static scoring of the current ligand poses in the context of the given* `constraints`.

```
surflex-opt -score optimize optID constraints param
surflex-opt -score optimize hivpr scr.constr dft.prm
```

After optimization is complete, output the optimized pose stored in each ligand's pose cache. This pose gave rise to the score found during optimization.

```
surflex-opt -exportposes optimize optID constraints param
surflex-opt -exportposes optimize hivpr scr.constr dft.prm
```

Repeat the optimization `N` *times using the* `constraints` *and initial* `parameters`. *Each optimization run is prefixed by a unique, incremented version of* `optID`. *Outputs the best parameters found over all runs. This command is equivalent to calling the* `surflex-opt optimize` *command* `N` *times.*

```
surflex-opt -repeat N optimize optID constraints param
surflex-opt -repeat 5 optimize hivpr scr.constr dft.prm
```

The first parameter steps taken are from param.log. Only progressive steps which lower the MSE are considered. Optimization then proceeds as usual. This command is useful for recapitulating a previous optimization run or finishing a paused optimization run.

```
surflex-opt -useparam log optimize optID constraints param
surflex-opt -useparam param.log optimize hivpr scr.constr dft.prm
```

DOCKING AND SCORING COMMANDS

Dock a `ligand` *to a* `protein` *using a* `protomol` *and the given scoring function* `parameters`. *Outputs the top ten scoring poses.*

```
surflex-opt dock ligand protomol protein param
surflex-opt dock ligand.mol2 proto.mol2 protein.mol2 opt.param
```

Statically score the given ligand pose. Uses the identical arguments as the `dock` *command above.*

```
surflex-opt score ligand protomol protein param
surflex-opt score ligand.mol2 proto.mol2 protein.mol2 opt.param
```

Optimize the given ligand pose. Uses the identical arguments as the `dock` *command above. No protomol is necessary. Outputs the optimized ligand pose.*

```
surflex-opt opt ligand protein param
surflex-opt opt ligand.mol2 protein.mol2 opt.param
```

Statically score the poses found in `ligArchive` *against a* `protein` *using a* `protomol` *and the given scoring function* `parameters`. *The scoring log is output with the given* `logname`.

```
surflex-opt score_list ligArchive protomol protein logname param
surflex-opt score_list tps.mol2 prto.mol2 p.mol2 tp.scores opt.prm
```

This command is useful for performing virtual screening experiments. Dock the active ligand poses found in the `tp` *mol2 archive and the decoy poses found in* `fp` *mol2 archive against a* `protein` *using a* `protomol` *and the given scoring function* `parameters`*. Then compute ROC statistics illustrating screening enrichment. The performance log is output with the given* `logname`*. If the* `tp` *or* `fp` *arguments are passed '0' (zero), that archive is ignored.*

```
surflex-opt dock_list_tpfp tp fp protomol protein logname param
surflex-opt dock_list_tpfp t.mol2 f.mol2 o.mol2 p.mol2 roc opt.prm
```

DOCKING AND SCORING OPTIONS

Perform a pre-minimization of the unbound ligand prior to docking. Also do an all-atom re-minimization during the docking process. This option helps eliminate hidden ligand strain energy as well as lessen interpenetration between protein and ligand atoms. This option may be used by any of the docking and scoring commands of the previous section.

```
surflex-opt -pscreen dock ligand protomol protein param
surflex-opt -pscreen dock lig.mol2 prto.mol2 prtn.mol2 opt.param
```

MISCELLANEOUS PROCESSING COMMANDS

Strip the protons off all ligands in the `archive`*. Then, re-protonate and minimize all ligands, saving them to* `newArchiveName`*.*

```
surflex-opt min archive newArchiveName
surflex-opt min ligands.mol2 ligands_min.mol2
```

C.1.2. Constraint File Format

This file lists constraints that govern the objective function of the Surflex scoring function optimizer. Constraints may be given in any order, one per line. All files to be read in require full path information. Empty lines and those preceded by '#' are ignored.

General file format is as follows:

```
<groups>
[group1] constraint1
[group1] constraint2

[group2] constraint3
[group2] constraint4
...
[groupN] constraintN

<weights>
group1 weight1
group2 weight2
group3 weight3
...
groupN weightN
```

Groups

Note that constraints can be organized into groups. During optimization, each constraint group has an equal amount of influence over the objective function. This group frequency arbitration is guaranteed inter-group only. All constraints must belong to some group.

Weights

Individual groups may also be given additional weight by specifying a positive, non-zero integer in the weights section. The weight acts as a group multiplier; thus giving a group a weight of 3 effectively lists that group 3 times in the constraint file. By adjusting weights, one can change the ratio by which training data is presented for optimization. Group weights are listed by group name in the weights section of the file after all constraints have been listed.

Score Constraints

Score Equal constraint: predicted score remains near a target score. This is useful for optimizing scoring accuracy.

```
group protein proto ligand = score
group1 protein.mol2 proto.mol2 ligand.mol2 = 6.0
```

Score Lesser constraint: predicted score must be less than a target score. This is useful for depressing the scores of negative (FP) ligands.

```
group protein proto ligand < score
group1 protein.mol2 proto.mol2 ligand.mol2 < 4.0
```

Score Greater constraint: predicted score must be greater than a target score. This is useful for promoting the scores of positive (TP) ligands.

```
group protein proto ligand > score
group1 protein.mol2 proto.mol2 ligand.mol2 > 5.0
```

Screening Constraints

Maximize the separation between true-positive scores and false-positive scores. This is useful for optimizing screening for a particular system of interest.

```
group protein proto roc tp.archive fp.archive
group2 protein.mol2 proto.mol2 roc actives.mol2 decoys.mol2
```

Geometric Constraints

Maintain that the bogus poses of a particular ligand always score less than the highest scoring good pose for that ligand. This is useful for optimizing docking accuracy.

```
group protein proto geometric_decoy good.poses bad.poses
group3 protein.mol2 proto.mol2 geometric_decoy good.mol2 bad.mol2
```

Being example constraint file here:

```
#
# An example constraint file.
#


<groups> # BEGIN GROUP SECTION #

# This is an ROC constraint; it belongs to group1.
# This constraint will try to maximize enrichment for the ligands in
# tp.dud.20.archive against the screening background provided by
# ZincDrugLike-v2.archive. Note that the following is actually
# one single line. The following looks like two lines due to the
# margins and word wrapping.

group1 hivpr/protein_opt.mol2 hivpr/p1-protomol.mol2 roc
hivpr/tp.dud.20.archive.mol2 hivpr/ZincDrugLike-v2.archive.mol2


# This is a score constraint; it belongs to group2.
# This constraint wants to keep the cognate1 score close to 4.6

group2 hivpr/protein.mol2 hivpr/p1-protomol.mol2 cognate1.mol2 = 4.6

# This is a score constraint; it also belongs to group2.
# This constraint will push the decoy1 score below 3.0

group2 hivpr/protein.mol2 hivpr/p1-protomol.mol2 decoy1.mol2 < 3.0


# This is a geometric decoy constraint; it belongs to group3.
# This constraint will push the scores of poses found in
# lig1.bad.poses.archive such that they remain less than the scores of
# the poses found in lig1.good.poses.archive.

group3 hivpr/protein.mol2 hivpr/p1-protomol.mol2 geometric_decoy
hivpr/lig1.good.poses.archive.mol2 hivpr/lig1.bad.poses.archive.mol2


# We want the scoring function to pay particular attention to group3,
# thus we double its weight here in the following weights section. As a
# result, the objective function listens to this group's constraints
# twice as much as any other group with the standard weight of 1.

<weights> # BEGIN WEIGHT SECTION #
group1 1
group2 1
group3 2
```

End of example constraint file.

C.1.3. Parameter File Format

This file contains the parameters values of the Surflex scoring function. The order in which the parameters appear in this file is critical and should not be changed. Empty lines and those preceded by '#' are ignored. The beginning of the file may contain an optional line containing optimization error information.

The general file format is as follows:

```
Final Error: MSE Epochs: NUM_EPOCHS

ID    Name        Range       Optimized   Initial     Comments
0     paramName0  range0      x.xxx       y.yyy       comments0
...
N     paramNameN  rangeN      a.aaa       b.bbb       commentsN
```

Parameter columns:

ID Order of parameter in the file

Name Parameter name

Range

Value	Parameter Restrictions
1	None
1	Must be positive
-1	Must be negative
11	One or greater
01	Between zero and one

Optimized Optimized parameter value
If NO_OPT, this parameter will not be optimized

Initial Initial parameter value

Comments User comments for the given parameter

Begin example parameter file:

```
## This is the default parameters for the scoring function
## within Surflex-Dock. This is a typical .param file that
## one can pass as an argument to Surflex.
##
## Changing these parameters allows one to customize their
## scoring function. By default, Surflex will load parameters
## from the 'Optimized' column.
##
## On an optimization run, if the 'Optimized' column contains
## the value 'NO_OPT', that parameter will not be optimized.
## The parameter will then default to the value in the 'Initial'
## column.
##
## Do not change the ordering of these parameters.

Final Error:      10000 Epochs: -1

#          Parameter     Rg    Optimized    Initial    Comments
 0             sf_stz      1      0.08980          0.08980    l0-steric
 1             sf_str     -1     -0.08410    -0.08410    l1
 2             sf_sts      1      0.62130    0.62130    n0
 3             sf_stm      1      0.13390    0.13390    n1
 4             sf_srm      1      0.48800    0.48800    n2
 5             sf_hrd     -1     -0.94500    -0.94500    interpenetration
 6             sf_poz      1      1.23880    1.23880    l2-polar
 7             sf_por     -1     -0.17960    -0.17960    l3
 8             sf_pos      1      0.32340    0.32340    n3
 9             sf_pom      1      0.63130    0.63130    n4
10             sf_hpl      1      0.61390    0.61390    n5
11             sf_csf      1      0.50000    0.50000    n6
12             sf_pr2     -1     -2.52000    -2.52000    l5-chg
13             sf_prm      1      0.50100    0.50100    n7
14              sf_ms      1      0.50000    0.50000    n8
15             sf_ent     -1     -0.21370    -0.21370    l7-entropy
16             sf_con     -1     -1.04060    -1.04060    l8
```

End example parameter file.

C.1.4. Code Documentation

Contents

Chapter 1

surflex-opt Data Structure Index

1.1 surflex-opt Data Structures

Here are the data structures with brief descriptions:

Chapter 2

surflex-opt File Index

2.1 surflex-opt File List

Here is a list of all files with brief descriptions:

Chapter 3

surflex-opt Data Structure Documentation

3.1 bucket Struct Reference

`#include <hash.h>`

3.1.1 Detailed Description

A hash table consists of an array of these buckets.

Each bucket holds a copy of the key, a pointer to the data associated with the key, and a pointer to the next bucket that collided with this one, if there was one.

Data Fields

- char * key

  Key that hashes to this bucket.

- void * data

  Data paired with this key.

- struct bucket * next

  Linked list of collisions on this key.

3.1.2 Field Documentation

3.1.2.1 char* bucket::key

Key that hashes to this bucket.

3.1.2.2 void* bucket::data

Data paired with this key.

3.1.2.3 struct bucket* bucket::next [read]

Linked list of collisions on this key.

The documentation for this struct was generated from the following file:

- hash.h

3.2 ConstraintData_struct Struct Reference

`#include <optimize-types.h>`

3.2.1 Detailed Description

Stores information relevant to a constraint.

Data Fields

- int type
  Constraint type.
- char group [MAX_BUFFER_SIZE]
  Name of the group of which this constraint is a part.
- int weight
  Weight given to this constraint.
- unsigned int nResample
  Number of times to resample this constraint (useful for weight arbitration).
- Protein * protein
  Input protein.
- LigData ** ligand
  Array of TP ligands.
- unsigned int nlig
  Number of TP ligands.
- LigData ** decoy
  Array of FP ligands.
- unsigned int ndecoy
  Number of FP ligands.
- LigData ** temp
  Storage allocated to hold all ligands during ROC computation.
- double targetScore

Target score of a scoring constraint.

- double error
  Constraint error.
- struct ConstraintData_struct * next
  Points to next ConstraintData in the linked list that represents a Constraint group.

3.2.2 Field Documentation

3.2.2.1 int ConstraintData_struct::type

Constraint type.

3.2.2.2 char ConstraintData_struct::group[MAX_BUFFER_SIZE]

Name of the group of which this constraint is a part.

3.2.2.3 int ConstraintData_struct::weight

Weight given to this constraint.

3.2.2.4 unsigned int ConstraintData_struct::nResample

Number of times to resample this constraint (useful for weight arbitration).

3.2.2.5 Protein* ConstraintData_struct::protein

Input protein.

3.2.2.6 LigData** ConstraintData_struct::ligand

Array of TP ligands.

3.2.2.7 unsigned int ConstraintData_struct::nlig

Number of TP ligands.

3.2.2.8 LigData** ConstraintData_struct::decoy

Array of FP ligands.

3.2.2.9 unsigned int ConstraintData_struct::ndecoy

Number of FP ligands.

3.2.2.10 LigData** ConstraintData_struct::temp

Storage allocated to hold all ligands during ROC computation.

3.2.2.11 double ConstraintData_struct::targetScore

Target score of a scoring constraint.

3.2.2.12 double ConstraintData_struct::error

Constraint error.

3.2.2.13 struct ConstraintData_struct* ConstraintData_struct::next [read]

Points to next ConstraintData in the linked list that represents a Constraint group.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.3 HashIter Struct Reference

`#include <hash.h>`

3.3.1 Detailed Description

Iterator data structure for traversing the hashtable.

Initialize using HashNewIterator(). Useful in while loops with HashIterateNext().

Data Fields

- bucket * next
  Next bucket.
- unsigned int index
  Current position in bucket[] of HashTable.
- HashTable * ht
  Iterator initialized to this HashTable.
- char * key
  Current key in iteration.
- void * val
  Current value in iteration.

3.3.2 Field Documentation

3.3.2.1 bucket* HashIter::next

Next bucket.

3.3.2.2 unsigned int HashIter::index

Current position in bucket[] of HashTable.

3.3.2.3 HashTable* HashIter::ht

Iterator initialized to this HashTable.

3.3.2.4 char* HashIter::key

Current key in iteration.

3.3.2.5 void* HashIter::val

Current value in iteration.

The documentation for this struct was generated from the following file:

- hash.h

3.4 HashTable Struct Reference

`#include <hash.h>`

3.4.1 Detailed Description

Stores information related to a hash table.

This is what you actually declare an instance of to create a table. You then call 'construct_table' with the address of this structure, and a guess at the size of the table. Note that more nodes than this can be inserted in the table, but performance degrades as this happens. Performance should still be quite adequate until 2 or 3 times as many nodes have been inserted as the table was created with.

Data Fields

- size_t size

  Initial guess of HashTable size in number of buckets.

- int currSize

  Current size of HashTable in number of buckets.

- bucket ** table

  Array of buckets.

3.4.2 Field Documentation

3.4.2.1 size_t HashTable::size

Initial guess of HashTable size in number of buckets.

3.4.2.2 int HashTable::currSize

Current size of HashTable in number of buckets.

3.4.2.3 bucket** HashTable::table

Array of buckets.

The documentation for this struct was generated from the following file:

- hash.h

3.5 LigData Struct Reference

`#include <optimize-types.h>`

3.5.1 Detailed Description

Stores information relevant to ligands during optimization.

Data Fields

- Molecule * mol
  Ligand Molecule.
- PoseCache * cache
  Ligand PoseCache.
- char filename [MAX_BUFFER_SIZE]
  Full pathname to ligand file/archive.
- char name [MAX_BUFFER_SIZE]
  Ligand name read from file.
- int type
  Type: POS/NEG.
- char protein [MAX_BUFFER_SIZE]
  Protein name.
- double initscore
  Initial score parsed from ligand file.
- double score
  Current ligand score.
- double error
  Current ligand error (if part of a score constraint).
- unsigned int initrank
  Initial ligand rank parsed from ligand file.
- unsigned int rank
  Current ligand rank.

3.5.2 Field Documentation

3.5.2.1 Molecule* LigData::mol

Ligand Molecule.

3.5.2.2 PoseCache* LigData::cache

Ligand PoseCache.

3.5.2.3 char LigData::filename[MAX_BUFFER_SIZE]

Full pathname to ligand file/archive.

3.5.2.4 char LigData::name[MAX_BUFFER_SIZE]

Ligand name read from file.

3.5.2.5 int LigData::type

Type: POS/NEG.

3.5.2.6 char LigData::protein[MAX_BUFFER_SIZE]

Protein name.

3.5.2.7 double LigData::initscore

Initial score parsed from ligand file.

3.5.2.8 double LigData::score

Current ligand score.

3.5.2.9 double LigData::error

Current ligand error (if part of a score constraint).

3.5.2.10 unsigned int LigData::initrank

Initial ligand rank parsed from ligand file.

3.5.2.11 unsigned int LigData::rank

Current ligand rank.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.6 OptData Struct Reference

`#include <optimize-types.h>`

3.6.1 Detailed Description

Stores information relevant to optimization.

Data Fields

- ParamSet * param

  Parameter Set data.

- void ** input

  Deprecated: Array of input examples (ligands).

- unsigned int numIn

  Deprecated: Number of input examples.

- void * extra

  Deprecated: Additional optimization data (was Proteins, now extra pose cache).

- ConstraintData ** groupHead

  Array of constraint group heads (first node in a linked list representing a constraint group).

- unsigned int ngroups

  Number of constraint groups.

- double minError

  Current MSE.

- double maxEpochs

  Maximum number of epochs for this optimization run.

- double maxEpochsNoImprove

  Maximum number of epochs without improvement for this optimization run.

- double stepSize

  Parameter step size; dependent on search strategy.

- double nreopt

  Number of times the poses in the pose cache were optimized.

3.6.2 Field Documentation

3.6.2.1 ParamSet* OptData::param

Parameter Set data.

3.6.2.2 void** OptData::input

Deprecated: Array of input examples (ligands).

3.6.2.3 unsigned int OptData::numIn

Deprecated: Number of input examples.

3.6.2.4 void* OptData::extra

Deprecated: Additional optimization data (was Proteins, now extra pose cache).

3.6.2.5 ConstraintData** OptData::groupHead

Array of constraint group heads (first node in a linked list representing a constraint group).

3.6.2.6 unsigned int OptData::ngroups

Number of constraint groups.

3.6.2.7 double OptData::minError

Current MSE.

3.6.2.8 double OptData::maxEpochs

Maximum number of epochs for this optimization run.

3.6.2.9 double OptData::maxEpochsNoImprove

Maximum number of epochs without improvement for this optimization run.

3.6.2.10 double OptData::stepSize

Parameter step size; dependent on search strategy.

3.6.2.11 double OptData::nreopt

Number of times the poses in the pose cache were optimized.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.7 ParamLog Struct Reference

`#include <optimize-types.h>`

3.7.1 Detailed Description

Stores information pertinent to a parameter log.

Useful for recapitulating optimization runs from a param.log file.

Data Fields

- unsigned int currStep
  Index into step[] of the current step.
- unsigned int nsteps
  Number of parameter steps.
- ParamStep * step
  Array of ParamStep.
- int nparam
  Number of parameters.

3.7.2 Field Documentation

3.7.2.1 unsigned int ParamLog::currStep

Index into step[] of the current step.

3.7.2.2 unsigned int ParamLog::nsteps

Number of parameter steps.

3.7.2.3 ParamStep* ParamLog::step

Array of ParamStep.

3.7.2.4 int ParamLog::nparam

Number of parameters.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.8 ParamSet Struct Reference

`#include <optimize-types.h>`

3.8.1 Detailed Description

Stores data pertinent to a parameter set.

Data Fields

- int n

  Number of params.

- char type [MAX_BUFFER_SIZE]

  Parameter data source.

- double * set

  Parameter values.

- char ** name

  Parameter names.

- char ** comments

  Additional parameter comments.

- int * range

  Parameter ranges: [NONE, POS, NEG, ONE_OR_GREATER, BETWEEN_ZERO_-ONE].

- double * incr

  Parameter increment sizes.

- double * init

  Initial parameter values.

- int * opt

  Array of parameter indices in set[] that are to be optimized.

- int nopt

  Number of optimized parameters.

- int optIndex

  Index into opt[] of the current parameter being optimized (useful to line search).

- int direction

  Direction of parameter perturbation (+/-) in line search.

- int index

  Index into set[] of the current parameter being optimized.

3.8.2 Field Documentation

3.8.2.1 int ParamSet::n

Number of params.

3.8.2.2 char ParamSet::type[MAX_BUFFER_SIZE]

Parameter data source.

3.8.2.3 double* ParamSet::set

Parameter values.

3.8.2.4 char** ParamSet::name

Parameter names.

3.8.2.5 char** ParamSet::comments

Additional parameter comments.

3.8.2.6 int* ParamSet::range

Parameter ranges: [NONE, POS, NEG, ONE_OR_GREATER, BETWEEN_ZERO_-ONE].

3.8.2.7 double* ParamSet::incr

Parameter increment sizes.

3.8.2.8 double* ParamSet::init

Initial parameter values.

3.8.2.9 int* ParamSet::opt

Array of parameter indices in set[] that are to be optimized.

3.8.2.10 int ParamSet::nopt

Number of optimized parameters.

3.8.2.11 int ParamSet::optIndex

Index into opt[] of the current parameter being optimized (useful to line search).

3.8.2.12 int ParamSet::direction

Direction of parameter perturbation (+/-) in line search.

3.8.2.13 int ParamSet::index

Index into set[] of the current parameter being optimized.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.9 ParamStep Struct Reference

`#include <optimize-types.h>`

3.9.1 Detailed Description

Stores information pertinent to a single parameter step.

Useful for recapitulating optimization runs from a param.log file.

Data Fields

- double error
  MSE of this set of parameters.
- char paramStr [2048]
  Parameter line parsed from param.log.
- double * param
  Parameter values; only read for successive decreases in MSE.
- int nparam
  Number of parameters.

3.9.2 Field Documentation

3.9.2.1 double ParamStep::error

MSE of this set of parameters.

3.9.2.2 char ParamStep::paramStr[2048]

Parameter line parsed from param.log.

3.9.2.3 double* ParamStep::param

Parameter values; only read for successive decreases in MSE.

3.9.2.4 int ParamStep::nparam

Number of parameters.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.10 PoseCache Struct Reference

`#include <optimize-types.h>`

3.10.1 Detailed Description

Stores information relevant to a pose cache.

Data Fields

- Conformer ** pose
  Array of poses.
- int size
  Number of poses, usually DEFAULT_POSE_CACHE_SIZE.
- int best
  Index into pose[] of the best scoring pose.
- int worst
  Index into pose[] of the worse scoring pose (first to be replaced).

3.10.2 Field Documentation

3.10.2.1 Conformer** PoseCache::pose

Array of poses.

3.10.2.2 int PoseCache::size

Number of poses, usually DEFAULT_POSE_CACHE_SIZE.

3.10.2.3 int PoseCache::best

Index into pose[] of the best scoring pose.

3.10.2.4 int PoseCache::worst

Index into pose[] of the worse scoring pose (first to be replaced).

The documentation for this struct was generated from the following file:

- optimize-types.h

3.11 Protein Struct Reference

`#include <optimize-types.h>`

3.11.1 Detailed Description

Stores cached information relevant to a protein during optimization.

Data Fields

- char path [MAX_BUFFER_SIZE]

  Full pathname to protein file.

- char protopath [MAX_BUFFER_SIZE]

  Full pathname to protomol file.

- Molecule * mol

  Protein Molecule.

- Molecule * protomol

  Protomol Molecule.

- Grid * grid

  Grid caches interesting active site atoms.

- unsigned int firstLig

  Index into OptData->input[] of this protein's first ligand (assumes ligands are organized in OptData->input[] in contiguous protein-specific blocks).

- unsigned int numLigands

  Number of ligands for this protein.

- unsigned int numTrue

  Number of positive examples (TPs) for this protein.

3.11.2 Field Documentation

3.11.2.1 char Protein::path[MAX_BUFFER_SIZE]

Full pathname to protein file.

3.11.2.2 char Protein::protopath[MAX_BUFFER_SIZE]

Full pathname to protomol file.

3.11.2.3 Molecule* Protein::mol

Protein Molecule.

3.11.2.4 Molecule* Protein::protomol

Protomol Molecule.

3.11.2.5 Grid* Protein::grid

Grid caches interesting active site atoms.

3.11.2.6 unsigned int Protein::firstLig

Index into OptData->input[] of this protein's first ligand (assumes ligands are organized in OptData->input[] in contiguous protein-specific blocks).

3.11.2.7 unsigned int Protein::numLigands

Number of ligands for this protein.

3.11.2.8 unsigned int Protein::numTrue

Number of positive examples (TPs) for this protein.

The documentation for this struct was generated from the following file:

- optimize-types.h

3.12 ProteinData Struct Reference

`#include <optimize-types.h>`

3.12.1 Detailed Description

Stores information relevant to all proteins during optimization.

Data Fields

- Protein * protein
  Array of Proteins.
- unsigned int numP
  Number of Proteins.
- HashTable * hash
  Hash for quick lookup of name->index into protein[].
- PoseCache * temp
  Copy placeholder for cache updates.

3.12.2 Field Documentation

3.12.2.1 Protein* ProteinData::protein

Array of Proteins.

3.12.2.2 unsigned int ProteinData::numP

Number of Proteins.

3.12.2.3 HashTable* ProteinData::hash

Hash for quick lookup of name->index into protein[].

3.12.2.4 PoseCache* ProteinData::temp

Copy placeholder for cache updates.

The documentation for this struct was generated from the following file:

- optimize-types.h

Chapter 4

surflex-opt File Documentation

4.1 hash.c File Reference

4.1.1 Detailed Description

HashTable code.

Public domain code by Jerry Coffin, with improvements by HenkJan Wolthuis.

```
#include <string.h>
#include <stdlib.h>
#include "hash.h"
```

Functions

- HashTable * HashConstructTable (HashTable *table, size_t size)

  Initialize the HashTable to the size asked for.

- static unsigned hash (const char *ptr)
- void * HashInsert (char *key, void *data, HashTable *table)

  Insert 'key' into hash table.

- void * HashLookup (char *key, HashTable *table)

  Returns a pointer to the data associated with a key.

- void * HashDel (char *key, HashTable *table)

  Deletes an entry from the table.

- void HashFreeTable (HashTable *table, void(*func)(void *))

  Frees a hash table.

- void HashEnumerate (HashTable *table, void(*func)(char *, void *))

  Goes through a hash table and calls the function passed to it for each node that has been inserted.

- char ** HashGetKeys (HashTable *table)

  Enumerates through all keys in the hashtable, returning an array of char's (keys), of size table->currSize.*

- HashIter * HashNewIterator (HashTable *ht)

  HashIter constructor.

- void * HashIterateNext (HashIter *hi)

  Method for traversing to the next key->value pair in the hashtable.

- double HashDoublePlusPlus (HashTable *ht, char *key)

  Increment by one the int value stored for the given key.

- double HashDoubleMinusMinus (HashTable *ht, char *key)

  Decrement by one the double value stored for the given key.

- int HashPlusPlus (HashTable *ht, char *key)

  Add one to the int value stored for the given key.

- HashTable * LoadHashDouble (char *filename)

  Create a hashtable from the given file.

4.1.2 Function Documentation

4.1.2.1 static unsigned hash (const char * *ptr*) `[static]`

4.1.2.2 HashTable* HashConstructTable (HashTable * *table*, size_t *size*)

Initialize the HashTable to the size asked for.

This is used to construct the table.

Allocates space for the correct number of pointers and sets them to NULL. If it can't allocate sufficient memory, signals error by setting the size of the table to 0.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

4.1.2.3 void* HashDel (char * *key*, struct HashTable * *table*)

Deletes an entry from the table.

Returns a pointer to the data that was associated with the key so the calling code can dispose of it properly.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.1.2.4 double HashDoubleMinusMinus (HashTable * *ht*, char * *key*)

Decrement by one the double value stored for the given key.

If no value exists for this key, initialize it to -1.

Parameters:

ht

key

Returns:

The newly incremented value

4.1.2.5 double HashDoublePlusPlus (HashTable * *ht*, char * *key*)

Increment by one the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.1.2.6 void HashEnumerate (struct HashTable * *table*, void(*)(char *, void *) *func*)

Goes through a hash table and calls the function passed to it for each node that has been inserted.

The function is passed a pointer to the key, and a pointer to the data associated with it.

Parameters:

table HashTable

func Function to call on user data

4.1.2.7 void HashFreeTable (HashTable * *table*, void(*)(void *) *func*)

Frees a hash table.

For each node that was inserted in the table, it calls the function whose address it was passed, with a pointer to the data that was in the table. The function is expected to free the data. Typical usage would be: free_table(&table, free); if the data placed in the table was dynamically allocated, or: free_table(&table, NULL); if not. (If the parameter passed is NULL, it knows not to call any function with the data.)

Parameters:

table HashTable

func Function to free user data

4.1.2.8 char** HashGetKeys (HashTable * *table*)

Enumerates through all keys in the hashtable, returning an array of char*'s (keys), of size table->currSize.

Parameters:

table Hashtable

Returns:

Newly allocated array of key strings

4.1.2.9 void* HashInsert (char * *key*, void * *data*, HashTable * *table*)

Insert 'key' into hash table.

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Returns pointer to old data associated with the key, if any, or NULL if the key wasn't in the table previously.

Parameters:

key Key string
data User data
table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

4.1.2.10 void* HashIterateNext (HashIter * *hi*)

Method for traversing to the next key->value pair in the hashtable.

Parameters:

hi A HashIter

Returns:

User data or NULL if all data has been returned by this HashIter.

4.1.2.11 void* HashLookup (char * *key*, struct HashTable * *table*)

Returns a pointer to the data associated with a key.

If the key has not been inserted in the table, returns NULL.

Parameters:

key Key string
table HashTable

Returns:

User data or NULL if not found

4.1.2.12 HashIter* HashNewIterator (HashTable * *ht*)

HashIter constructor.

Parameters:

ht HashTable

Returns:

A newly allocated HashIter

4.1.2.13 int HashPlusPlus (HashTable * *ht*, char * *key*)

Add one to the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.1.2.14 HashTable* LoadHashDouble (char * *filename*)

Create a hashtable from the given file.

The file contents are a set of key->value pairs, one on each row in the following format: key[tab]value[newline]

Parameters:

filename A data file

Returns:

Newly allocated HashTable

4.2 hash.h File Reference

4.2.1 Detailed Description

HashTable public interface.

```
#include <stddef.h>
#include <stdio.h>
```

Data Structures

- struct bucket

  A hash table consists of an array of these buckets.

- struct HashTable

  Stores information related to a hash table.

- struct HashIter

  Iterator data structure for traversing the hashtable.

Functions

- HashTable * HashConstructTable (HashTable *table, size_t size)

  This is used to construct the table.

- void * HashInsert (char *key, void *data, struct HashTable *table)

  Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

- void * HashLookup (char *key, struct HashTable *table)

  Returns a pointer to the data associated with a key.

- void * HashDel (char *key, struct HashTable *table)

  Deletes an entry from the table.

- void HashEnumerate (struct HashTable *table, void(*func)(char *, void *))

  Goes through a hash table and calls the function passed to it for each node that has been inserted.

- char ** HashGetKeys (HashTable *table)

  Enumerates through all keys in the hashtable, returning an array of char's (keys), of size table->currSize.*

- void HashFreeTable (HashTable *table, void(*func)(void *))

  Frees a hash table.

- HashIter * HashNewIterator (HashTable *ht)

  HashIter constructor.

- void * HashIterateNext (HashIter *hi)

  Method for traversing to the next key->value pair in the hashtable.

- int HashPlusPlus (HashTable *ht, char *key)

  Add one to the int value stored for the given key.

- double HashDoublePlusPlus (HashTable *ht, char *key)

  Increment by one the int value stored for the given key.

- double HashDoubleMinusMinus (HashTable *ht, char *key)

  Decrement by one the double value stored for the given key.

- HashTable * LoadHashDouble (char *filename)

  Create a hashtable from the given file.

4.2.2 Function Documentation

4.2.2.1 HashTable* HashConstructTable (HashTable * *table*, size_t *size*)

This is used to construct the table.

If it doesn't succeed, it sets the table's size to 0, and the pointer to the table to NULL.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

This is used to construct the table.

Allocates space for the correct number of pointers and sets them to NULL. If it can't allocate sufficient memory, signals error by setting the size of the table to 0.

Parameters:

table An existing HashTable to reinitialize (NULL if unnecessary)

size Initial number of buckets

Returns:

Newly allocated HashTable

4.2.2.2 void* HashDel (char * *key*, struct HashTable * *table*)

Deletes an entry from the table.

Returns a pointer to the data that was associated with the key so the calling code can dispose of it properly.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.2.2.3 double HashDoubleMinusMinus (HashTable * *ht*, char * *key*)

Decrement by one the double value stored for the given key.

If no value exists for this key, initialize it to -1.

Parameters:

ht

key

Returns:

The newly incremented value

4.2.2.4 double HashDoublePlusPlus (HashTable * *ht*, char * *key*)

Increment by one the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.2.2.5 void HashEnumerate (struct HashTable * *table*, void(*)(char *, void *) *func*)

Goes through a hash table and calls the function passed to it for each node that has been inserted.

The function is passed a pointer to the key, and a pointer to the data associated with it.

Parameters:

table HashTable

func Function to call on user data

4.2.2.6 void HashFreeTable (HashTable * *table*, void(*)(void *) *func*)

Frees a hash table.

For each node that was inserted in the table, it calls the function whose address it was passed, with a pointer to the data that was in the table. The function is expected to free the data. Typical usage would be: free_table(&table, free); if the data placed in the table was dynamically allocated, or: free_table(&table, NULL); if not. (If the parameter passed is NULL, it knows not to call any function with the data.)

Parameters:

table HashTable

func Function to free user data

4.2.2.7 char** HashGetKeys (HashTable * *table*)

Enumerates through all keys in the hashtable, returning an array of char*'s (keys), of size table->currSize.

Parameters:

table Hashtable

Returns:

Newly allocated array of key strings

4.2.2.8 void* HashInsert (char * *key*, void * *data*, HashTable * *table*)

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Note that this makes a copy of the key, but NOT of the associated data.

Parameters:

key Key string

data User data

table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

Inserts a pointer to 'data' in the table, with a copy of 'key' as its key.

Returns pointer to old data associated with the key, if any, or NULL if the key wasn't in the table previously.

Parameters:

key Key string

data User data

table HashTable

Returns:

Collision data or NULL if bucket was unoccupied

4.2.2.9 void* HashIterateNext (HashIter * *hi*)

Method for traversing to the next key->value pair in the hashtable.

Parameters:

hi A HashIter

Returns:

User data or NULL if all data has been returned by this HashIter.

4.2.2.10 void* HashLookup (char * *key*, struct HashTable * *table*)

Returns a pointer to the data associated with a key.

If the key has not been inserted in the table, returns NULL.

Parameters:

key Key string

table HashTable

Returns:

User data or NULL if not found

4.2.2.11 HashIter* HashNewIterator (HashTable * *ht*)

HashIter constructor.

Parameters:

ht HashTable

Returns:

A newly allocated HashIter

4.2.2.12 int HashPlusPlus (HashTable * *ht*, char * *key*)

Add one to the int value stored for the given key.

If no value exists for this key, initialize it to 1.

Parameters:

ht

key

Returns:

The newly incremented value

4.2.2.13 HashTable* LoadHashDouble (char * *filename*)

Create a hashtable from the given file.

The file contents are a set of key->value pairs, one on each row in the following format: key[tab]value[newline]

Parameters:

filename A data file

Returns:

Newly allocated HashTable

4.3 optimize-types.h File Reference

4.3.1 Detailed Description

Surflex-opt data structures, macros, #defines.

```
#include "hash.h"
```

Data Structures

- struct ParamSet

  Stores data pertinent to a parameter set.

- struct ParamStep

  Stores information pertinent to a single parameter step.

- struct ParamLog

  Stores information pertinent to a parameter log.

- struct PoseCache

  Stores information relevant to a pose cache.

- struct Protein

  Stores cached information relevant to a protein during optimization.

- struct ProteinData

  Stores information relevant to all proteins during optimization.

- struct LigData

  Stores information relevant to ligands during optimization.

- struct ConstraintData_struct

  Stores information relevant to a constraint.

- struct OptData

  Stores information relevant to optimization.

Defines

- #define SAVE_MINIMA 0

  Turns on saving all minima in logs (deactivated in code).

- #define MINIMA_THRESH 1.58

  Save minima with MSE less than this threshold.

- #define GEN_CONF_WITH_PARAM 0

  Turns on generation of beginning conformations using the initial parameters.

- #define INIT_ZERO 0

  Turns on initialization of all parameters to zero (ignoring initial values from param.file).

- #define DEFAULT_MAX_EPOCHS 100000

  Default stopping condition: maximum number of optimization epochs.

- #define DEFAULT_INIT_MSE 100000

  Default initial MSE value.

- #define DEFAULT_MIN_MSE 0.0001

  Default stopping condition: minimum MSE value.

- #define DEFAULT_RW_MAX_EPOCHS_NO_IMPROVE 200

  Default stopping condition: maximum epochs without improvement when performing random walk search.

- #define DEFAULT_N_CYCLE_NO_BETTER_BEFORE_STOP 2

  Default stopping condition: maximum number of random walk/line opt.

- #define DEFAULT_STEP_SIZE 100

  Pertubation search step size: 1/STEP_SIZE ∗ PARAM_INIT = increment size.

- #define DEFAULT_RW_STEP_SIZE 10
- #define DEFAULT_PARAM_RANGE 5

  Random walk search range: [-PARAM_INIT ∗ PARAM_RANGE ∗ 0.5, PARAM_INIT ∗ PARAM_RANGE ∗ 0.5].

- #define DEFAULT_POSE_CACHE_SIZE 5

  Default number of poses to keep for every ligand during optimization.

- #define SCORE_FIRST_UPDATE 1

  If on, statically score all poses in cache, then optimize the best scoring pose.

- #define OPT_FIRST_UPDATE 0

  If on, optimize all poses in cache (expensive).

- #define DEFAULT_NGOOD_2_OPTPOSE 5

  Default number of epochs with MSE improvement before reoptimizing all poses in posecache.

- #define NEG_MAX_SCORE 4.0

  Default FP score threshold.

- #define POS_MIN_SCORE 5.0

  Default TP score threshold.

- #define MAX_PARAMS 17

  Number of scoring function parameters.

- #define LOG_EPOCH_DATA -1

  Log data every n epochs (turn off by setting to -1).

- #define LOG_EPOCH_PARAM 1

  Log parameters sampled every n epochs (turn off by setting to -1).

- #define DATA_LOG_FLUSH 100

  Flush data log to disk every n writes.

- #define MAX_NAME_LEN 16

  Maximum string length for various fields.

- #define ROC_ERROR 1

  Output ROC performance data to log.

- #define SCORE_STATIC_POSES 0

  If true, no in-line pose optimization (disabled pose cache).

- #define CONSTRAINT_SCORE_EQUAL 1
- #define CONSTRAINT_SCORE_GREATER 2
- #define CONSTRAINT_SCORE_LESSER 3
- #define CONSTRAINT_ROC 4
- #define CONSTRAINT_GEOM_DECOY 5
- #define CONSTRAINT_TAG_ROC "roc"
- #define CONSTRAINT_TAG_GEOM_DECOY "geometric_decoy"
- #define CONSTRAINT_TAG_EQUAL "="
- #define CONSTRAINT_TAG_GREATER ">"
- #define CONSTRAINT_TAG_LESSER "<"
- #define CONSTRAINT_TAG_COMMENT "#"

- #define CONSTRAINT_TAG_GROUP "<groups>"
- #define CONSTRAINT_TAG_WEIGHT "<weights>"
- #define CI_PERCENT 95.0

  Default confidence interval percentage.

- #define CI_SAMPLES 1000

  Default number of samples for CI calculation.

- #define TRUE 1
- #define FALSE 0
- #define NEG -1
- #define POS 1
- #define TP34 2
- #define NONE 0
- #define ZERO 0.00000001
- #define ONE_OR_GREATER 11
- #define BETWEEN_ZERO_ONE 10
- #define NO_OPT "NO_OPT"
- #define MAX_BUFFER_SIZE 1024
- #define NO_INIT_SCORE -9998
- #define FAILED_DOCKING_SCORE -9999
- #define STATIC 0
- #define QUICK 1
- #define THOROUGH 2
- #define LIG_PENALTY(conf) (conf → data[8])

  Access the ligand penalty score.

- #define PROT_PENALTY(conf) (conf → data[9])

  Access the protein penalty score.

- #define BEST_CACHE_SCORE(ligdata) (ligdata → cache → pose[ligdata → cache → best] → score)

  Highest score in the pose cache.

- #define BEST_CACHE_POSE(ligdata) (ligdata → cache → pose[ligdata → cache → best])

  Index of the best pose in the pose cache array.

- #define CONSTRAINT_TYPE(cd) (cd → type == CONSTRAINT_ROC ? "roc" : (cd → type == CONSTRAINT_GEOM_DECOY ? "geometric_decoy" : (cd → type == CONSTRAINT_SCORE_EQUAL ? "score=" : (cd → type == CONSTRAINT_SCORE_LESSER ? "score<" : "score>"))))

  Quick test on constraint type.

Typedefs

- typedef struct ConstraintData_struct ConstraintData

4.3.2 Define Documentation

4.3.2.1 #define BEST_CACHE_POSE(ligdata) (ligdata → cache → pose[ligdata → cache → best])

Index of the best pose in the pose cache array.

4.3.2.2 #define BEST_CACHE_SCORE(ligdata) (ligdata → cache → pose[ligdata → cache → best] → score)

Highest score in the pose cache.

4.3.2.3 #define BETWEEN_ZERO_ONE 10

4.3.2.4 #define CI_PERCENT 95.0

Default confidence interval percentage.

4.3.2.5 #define CI_SAMPLES 1000

Default number of samples for CI calculation.

4.3.2.6 #define CONSTRAINT_GEOM_DECOY 5

4.3.2.7 #define CONSTRAINT_ROC 4

4.3.2.8 #define CONSTRAINT_SCORE_EQUAL 1

4.3.2.9 #define CONSTRAINT_SCORE_GREATER 2

4.3.2.10 #define CONSTRAINT_SCORE_LESSER 3

4.3.2.11 #define CONSTRAINT_TAG_COMMENT "#"

4.3.2.12 #define CONSTRAINT_TAG_EQUAL "="

4.3.2.13 #define CONSTRAINT_TAG_GEOM_DECOY "geometric_decoy"

4.3.2.14 #define CONSTRAINT_TAG_GREATER ">"

4.3.2.15 #define CONSTRAINT_TAG_GROUP "<groups>"

4.3.2.16 #define CONSTRAINT_TAG_LESSER "<"

4.3.2.17 #define CONSTRAINT_TAG_ROC "roc"

4.3.2.18 #define CONSTRAINT_TAG_WEIGHT "<weights>"

4.3.2.19 #define CONSTRAINT_TYPE(cd) (cd → type == CONSTRAINT_-ROC ? "roc" : (cd → type == CONSTRAINT_GEOM_DECOY ? "geometric_decoy" : (cd → type == CONSTRAINT_SCORE_EQUAL ? "score=" : (cd → type == CONSTRAINT_SCORE_LESSER ? "score<" : "score>"))))

Quick test on constraint type.

4.3.2.20 #define DATA_LOG_FLUSH 100

Flush data log to disk every n writes.

4.3.2.21 #define DEFAULT_INIT_MSE 100000

Default initial MSE value.

4.3.2.22 #define DEFAULT_MAX_EPOCHS 100000

Default stopping condition: maximum number of optimization epochs.

4.3.2.23 #define DEFAULT_MIN_MSE 0.0001

Default stopping condition: minimum MSE value.

4.3.2.24 #define DEFAULT_N_CYCLE_NO_BETTER_BEFORE_STOP 2

Default stopping condition: maximum number of random walk/line opt.

4.3.2.25 #define DEFAULT_NGOOD_2_OPTPOSE 5

Default number of epochs with MSE improvement before reoptimizing all poses in posecache.

4.3.2.26 #define DEFAULT_PARAM_RANGE 5

Random walk search range: [-PARAM_INIT $*$ PARAM_RANGE $*$ 0.5, PARAM_-INIT $*$ PARAM_RANGE $*$ 0.5].

4.3.2.27 #define DEFAULT_POSE_CACHE_SIZE 5

Default number of poses to keep for every ligand during optimization.

4.3.2.28 #define DEFAULT_RW_MAX_EPOCHS_NO_IMPROVE 200

Default stopping condition: maximum epochs without improvement when performing random walk search.

4.3.2.29 #define DEFAULT_RW_STEP_SIZE 10

4.3.2.30 #define DEFAULT_STEP_SIZE 100

Pertubation search step size: 1/STEP_SIZE $*$ PARAM_INIT = increment size.

4.3.2.31 #define FAILED_DOCKING_SCORE -9999

4.3.2.32 #define FALSE 0

4.3.2.33 #define GEN_CONF_WITH_PARAM 0

Turns on generation of beginning conformations using the initial parameters.

4.3.2.34 #define INIT_ZERO 0

Turns on initialization of all parameters to zero (ignoring initial values from param.file).

4.3.2.35 #define LIG_PENALTY(conf) (conf → data[8])

Access the ligand penalty score.

4.3.2.36 #define LOG_EPOCH_DATA -1

Log data every n epochs (turn off by setting to -1).

4.3.2.37 #define LOG_EPOCH_PARAM 1

Log parameters sampled every n epochs (turn off by setting to -1).

4.3.2.38 #define MAX_BUFFER_SIZE 1024

4.3.2.39 #define MAX_NAME_LEN 16

Maximum string length for various fields.

4.3.2.40 #define MAX_PARAMS 17

Number of scoring function parameters.

4.3.2.41 #define MINIMA_THRESH 1.58

Save minima with MSE less than this threshold.

4.3.2.42 #define NEG -1

4.3.2.43 #define NEG_MAX_SCORE 4.0

Default FP score threshold.

4.3.2.44 #define NO_INIT_SCORE -9998

4.3.2.45 #define NO_OPT "NO_OPT"

4.3.2.46 #define NONE 0

4.3.2.47 #define ONE_OR_GREATER 11

4.3.2.48 #define OPT_FIRST_UPDATE 0

If on, optimize all poses in cache (expensive).

4.3.2.49 #define POS 1

4.3.2.50 #define POS_MIN_SCORE 5.0

Default TP score threshold.

4.3.2.51 #define PROT_PENALTY(conf) (conf → data[9])

Access the protein penalty score.

4.3.2.52 #define QUICK 1

4.3.2.53 #define ROC_ERROR 1

Output ROC performance data to log.

4.3.2.54 #define SAVE_MINIMA 0

Turns on saving all minima in logs (deactivated in code).

4.3.2.55 #define SCORE_FIRST_UPDATE 1

If on, statically score all poses in cache, then optimize the best scoring pose.

4.3.2.56 #define SCORE_STATIC_POSES 0

If true, no in-line pose optimization (disabled pose cache).

4.3.2.57 #define STATIC 0

4.3.2.58 #define THOROUGH 2

4.3.2.59 #define TP34 2

4.3.2.60 #define TRUE 1

4.3.2.61 #define ZERO 0.00000001

4.3.3 Typedef Documentation

4.3.3.1 typedef struct ConstraintData_struct ConstraintData

4.4 optimize.c File Reference

4.4.1 Detailed Description

Surflex-opt code.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "optimize.h"
#include "hash.h"
#include "utils.h"
#include "ucsf-roc.h"
```

Functions

- unsigned int optParam (OptData *data, ParamSet *newParam, unsigned int errorSize, FILE *paramlog, long int *epoch, unsigned int initBetter)

  Entry optimization function.

- OptData * initOptData (char *optID, char **files, int nfiles, double **error, unsigned int *errorSize, ParamSet **newParam)

  Initialize the optimization data.

- void setupSingleRun (OptData *data, ParamSet *newParam, long int epoch)

  Setup optimization data specific to a single run's strategy.

- ConstraintData ** readConstraintFile (char *constraintFile, int *ngroup, unsigned int *errorSize)

  Initialize the constraints data structure from the given file.

- int lookupGroupWeight (HashTable *table, char *constraintLine)

  Hash lookup of a weight assigned to a constraint group Useful while parsing:.

- unsigned int getNumConstraints (char *constraintFile, unsigned int *nweights)

  Get the number of constraints and weights parsed from the constraintFile.

- ConstraintData * readConstraint (char *constraintLine, char *group, int groupWeight)

  Given a line from a constraintFile, initialize a ConstraintData structure.

- void readConstraintROC (ConstraintData *cd, char *tpArchiveFile, char *fpArchiveFile)

  Read a screening constraint.

- void readConstraintGeomDecoy (ConstraintData *cd, char *goodPosesFile, char *badPosesFile)

  Read a geometric constraint.

- void readConstraintScore (ConstraintData *cd, char *ligand, char *scoreType, char *score)

  Read score constraint.

- void readGroupWeight (char *weightLine, HashTable *table)

  Parse the group weight information from a line in a constraint file.

- Protein * setupConstraintProtein (char *proteinFile, char *protoFile)

  Initialize the Protein data structure.

- LigData ** readMol2Archive (char *archive, unsigned int *nmol, int type, char *protein, int poseCacheOn)

  Read in a mol2 archive.

- LigData * setupLigData (Molecule *mol, char *archive, int type, char *protein, int poseCacheOn)

  Initialize LigData structure.

- void parseMoleculeNameField (LigData *ligData, char *name, char *protein)

  Parse mol2 files that have standardized name field.

- LigData * setupConstraintLigData (Molecule *mol, char *archive, int type)
- unsigned int getNumMolecules (char *molArchive)

  Retrieve the number of molecules in a mol2 archive.

- void addConstraintToGroup (char *group, ConstraintData *groupHead, ConstraintData *constraint)

  Add a constraint to the group at the end of the linked list.

- ConstraintData ** buildGroupHeadArr (HashTable *table, int *nhead)

  Convert a HashTable of unique group heads into an array of group heads.

- void emptyFn (void *data)

  This function does nothing.

- unsigned int resolveSamplingFrequency (ConstraintData **groupHead, int ngroup)

  Constraint weight arbitration.

- unsigned int findLCM (unsigned int *arr, int n)

  Find the least common multiple between a set of numbers.

- double optimizeDock (Conformer *ligand, Molecule *protein, Grid *grid, int mode)

  Score a protein and ligand conformation.

- double eval_constraint (LigData *ligand, Protein *protein, long int epoch)

  Wrapper for eval() determines what level of optimization will be used in scoring.

- double scoreConstraintGroup (ConstraintData *groupHead, double *error, long int epoch, FILE *datalog)

  Traverse the group, scoring each constraint.

- void scoreConstraint (ConstraintData *cd, double *error, long int epoch, FILE *datalog)

  Score by constraint type.

- void scoreConstraintROC (ConstraintData *cd, double *error, long int epoch, FILE *datalog)

  Screening constraint: maximize the scoring separation between TPs (actives) and FPs (decoys).

- void scoreConstraintGeomDecoy (ConstraintData *cd, double *error, long int epoch, FILE *datalog)

  Geometric constraint: bogus poses must always score worse than best good pose.

- void scoreConstraintScore (ConstraintData *cd, double *error, long int epoch, FILE *datalog)

  Scoring constraint: ligand must be = or < or > than the targetScore.

- void updatePoses_constraint (OptData *data, unsigned int nbetter)

  Update all poses in every ligand's pose cache.

- FILE * fireupDataLog (OptData *data, long int epoch)

Determine if this epoch will be logged in detail.

- void updateDataLog_constraint (OptData *data, char *filename)

  Write to log the state of the data.

- void updateDataLogROC (FILE *file, ConstraintData *cd)

  Write to the data log the current state of things for an ROC constraint.

- void updateDataLogGeomDecoy (FILE *file, ConstraintData *cd)

  Write to the data log the current state of things for this geometric decoy constraint.

- void updateDataLogScore (FILE *file, ConstraintData *cd)

  Write to the data log the current state of things for this score constraint.

- void writeNewConstraintFile (OptData *data, char *optID)

  Write a new constraint file using newly exported best poses.

- void exportBestPoses_constraint (ConstraintData *cd, char *optID, int constraintID, FILE *constraintFile)

  Write out the best poses found during optimization with the best parameters.

- void writeFinalDataLog (OptData *data, char *optID, double errorSize, char *runtime, long int epoch)

  Do a final scoring with best parameters and then write out the final log.

- void advanceParamIndex (ParamSet *param)

  Advance the index of the parameter to be optimized.

- void writeParam (ParamSet *param, FILE *outfile, char *path, int newDefault, OptData *data, long int nepochs, char *inputFile)

  Write the parameters to a param file.

- int readParam (FILE *file, OptData *data)

  Read parameters from a paramfile.

- ParamSet * callocParam (int nparam, int nopt)

  Allocate space for ParamSet.

- void copyParam (ParamSet *to, ParamSet *from, int setOnly)

  Utility for copying a ParamSet.

- int paramWillBeOpt (ParamSet *param, int index)

  Test if this parameter will be optimized.

- void initEpochData (double *error, unsigned int errorSize, ParamSet *oldParam, ParamSet *newParam, long int epoch)

  Ready the parameters for this epoch.

- void setupParamLog (FILE **log, ParamSet *param)

  Setup the parameter log.

- void writeParamLog (FILE *log, long int epoch, ParamSet *param, double error)

  If necessary, update the parameter log with optimization progress.

- void updateStrategyStatus (FILE *file)

  Write to file our strategy status.

- void sanityCheck (FILE *out, OptData *data)

  Output details regarding the optimization status.

- PoseCache * newPoseCache (int maxsize)

  Allocate memory for PoseCache struct.

- void freePoseCache (PoseCache *c, int freePoses)

  Free memory allocated to PoseCache struct.

- void copyPoseCache (PoseCache *to, PoseCache *from)

  Deep copy of a PoseCache data structure.

- double evalPoseCache (PoseCache *cache, Protein *protein, int scoreMode)

  Return the highest scoring pose in the poseCache.

- void updatePoses (OptData *data, unsigned int nbetter)
- double updatePoseCache (Protein *protein, LigData *ligand, PoseCache *temp)

  Reoptimize the poses in the pose cache.

- double scoreFirstUpdate (Protein *protein, LigData *ligand, PoseCache *temp)

  Optimize only highest scoring pose, then add it to pose cache if necessary.

- double optFirstUpdate (Protein *protein, LigData *ligand, PoseCache *temp)

  Optimize all poses, then add the best scoring pose if necessary.

- int isUniquePose (PoseCache *cache, Conformer *pose)

Uniqueness test for a pose within a pose cache.

- void exportBestPoses (OptData *data)

  Write out the best pose for each ligand, given the current parameters.

- double rocSorted_generalized (LigData **ligand, unsigned int numLig, unsigned int ntp, char *logname, char *proteinName)

  Wrapper for ucsf-roc, outputs detailed log file.

- int compLigScore (const void *p1, const void *p2)

  qsort comparison function for LigData scores.*

- void my_dock_list (LigData **ligData, int numLig, Molecule *protmol, Molecule *protomol, Grid *grid, char *logpath, int writeLog)

  Dock a list of molecules.

- ParamLog * readParamLog (char *paramLogFile, ParamStep **allstep, unsigned int *ntotal)

  Read in a parameter log file.

- int getNumParam (FILE *file)

  From first line of param.log, count the number of params.

- ParamStep * grabProgressiveSteps (char *paramLog, int nparam, ParamStep **step, unsigned int *ntotal, unsigned int *ngood)

  Grab only the progressive parameter steps from a parameter log file.

- int compStepError (const void *p1, const void *p2)

  qsort comparison function for ParamStep by error.

- ParamStep * grabParamSteps (FILE *file, unsigned int totalSteps, int nparam)

  Parse each parameter step from the parameter log.

- void getNextParamStep (ParamLog *paramLog, ParamSet *param)

  Move one parameter step forward.

- void writeProgressiveSteps (char *paramLogFile, ParamStep *allsteps, unsigned int nsteps)

  Output a log of only the progressive steps made in parameter search space.

- void freeParamSet (ParamSet *p)

  Free memory allocated to ParamSet struct.

- void freeProteinData (ProteinData *p)

  Free memory allocated to ProteinData struct.

- void freeProtein (Protein *p)

  Free memory allocated to Protein struct.

- void freeLigData (LigData *l)

  Free memory allocated to LigData struct.

- void freeConstraintData (ConstraintData *c)

  Free memory allocated to ConstraintData struct.

- void freeParamLog (ParamLog *log)

  Free memory allocated a ParamLog.

- void freeParamStep (ParamStep *p, int freePtr)

  Free memory allcoated a ParamStep.

- void resetArr (double *arr, int numElem, double val)

  Reset all elements of an arr to val.

- double avgArr (double *arr, int numElem)

  Calculate the avg of a double arr.

- double sqError (double a, double b)

  Calculate square error of two numbers.

- void repeatOptimize (char *optID, char *constraintFile, char *paramFile, int nreps)

  Wrapper function to repeat optimize() calls.

- double optimize (char *optID, char *constraintFile, char *paramFile)

  Fully automated, constraint-based scoring function optimization.

- void redoOpt (char *paramLogFile)

  Recapitulate an optimization run from a param.log.

- void initParam (OptData *data, char *paramFile)

  Get initial values for ParamSet from a parameter file.

- void updateParam (ParamSet *oldParam, ParamSet *newParam, long int epoch)

  Update the old parameters by some increment based on the search strategy.

- void setParam (ParamSet *param, long int epoch)

  Set the parameters.

- double eval (ParamSet *param, OptData *data, int index, long int epoch)

  Run the evaluation function on the given input example.

- void updateParamLog (FILE *log, long int epoch, ParamSet *param, double error)

  Specialized printing for the parameter log.

- void freeOptData (OptData *data, double *error)

  Free memory allocated to OptData struct.

- double my_score (char *ligFile, char *proteinFile, char *protoFile, char *paramFile, int optdockFlag)

  Given files containing their conformation, score a protein and ligand.

- double loadParam (char *filename)

  Parse the parameters from the standard .param file.

- void my_dock_list_tpfp (char *truePath, char *decoyPath, char *protopath, char *protpath, char *logpath)

  Dock a list of molecules for screening enrichment.

Variables

- int crlf_p
- int _RANDOM_WALK

  If on, perform random walk search.

- int _RANDOM_SEARCH

  If on, perform bounded random search.

- int _SCORE_DATA

  If on, only score data.

- int _SINGLE_RUN

  If on, perform only a single search run.

- int _EXPORT_POSES

If on, export optimized poses with best parameters.

- double _MAX_EPOCHS

  Stopping condition: maximum optimization epochs.

- double _RW_MAX_EPOCHS_NO_IMPROVE

  Stopping condition: maximum epochs of random walk search without improvement.

- double _MIN_MSE

  Stopping condition: minimum MSE.

- int _N_CYCLE_NO_BETTER_BEFORE_STOP

  Stopping condition: maximum number of search cycles without improvement.

- double _STEP_SIZE

  Line search parameter increment size fraction.

- double _RW_STEP_SIZE

  Random walk parameter increment size fraction.

- double _PARAM_RANGE

  Random walk parameter range factor.

- int _NGOOD_2_OPTPOSE

  Number of epochs with improvement before optimizing all poses in pose cache.

- int _POSE_CACHE_SIZE

  Number of poses to store in each ligand's pose cache.

- unsigned int _total = 0

  OPT_FIRST_UPDATE stat tracking: total number of times a pose cache was optimized.

- unsigned int _match = 0

  OPT_FIRST_UPDATE stat tracking: number of times highest scoring pose pre-optimization remained highest scoring post-optimization.

- void ** _LIG_COPY = NULL

  Data storage for roc calculation.

- int n_dock_final
- int max_rot
- ParamLog * _PARAM_LOG = NULL

Parameter log.

- ParamStep * _ALL_STEP = NULL

  Array of ParamSteps.

- unsigned int _N_ALL_STEP = 0

  Number of parameter steps.

4.4.2 Function Documentation

4.4.2.1 void addConstraintToGroup (char * *group*, ConstraintData * *groupHead*, ConstraintData * *constraint*)

Add a constraint to the group at the end of the linked list.

Parameters:

group Group ID

groupHead Head node of the group

constraint New constraint to add to group

4.4.2.2 void advanceParamIndex (ParamSet * *param*)

Advance the index of the parameter to be optimized.

We cycle thru the indices, returning to the first after the last is seen.

See also:

ParamSet::opt
ParamSet::optIndex
ParamSet::index

Parameters:

param A parameter set

4.4.2.3 double avgArr (double * *arr*, int *numElem*)

Calculate the avg of a double arr.

Parameters:

arr Array of doubles

numElem Size of array

Returns:

Average

4.4.2.4 ConstraintData ∗∗ buildGroupHeadArr (HashTable ∗ *table*, int ∗ *nhead*)

Convert a HashTable of unique group heads into an array of group heads.

Parameters:

table HashTable of unique group heads

nhead Storage for number of heads in array

Returns:

Newly allocated array of group heads

4.4.2.5 ParamSet ∗ callocParam (int *nparam*, int *nopt*)

Allocate space for ParamSet.

Parameters:

nparam Number of parameters

nopt Number of parameters to be optimized

Returns:

Newly allocated ParamSet

4.4.2.6 int compLigScore (const void ∗ *p1*, const void ∗ *p2*)

qsort comparison function for LigData∗ scores.

Sorts in descending order.

Parameters:

p1 LigData1

p2 LigData2

Returns:

-1 if p1 > p2, 0 if p1 == p2, 1 if p1 < p2

4.4.2.7 int compStepError (const void * *p1*, const void * *p2*)

qsort comparison function for ParamStep by error.

Sorts in descending order.

Parameters:

p1 ParamStep1
p2 ParamStep2

Returns:

1 if p1 < p2, 0 if p1 == p2, -1 if p1 > p2

4.4.2.8 void copyParam (ParamSet * *to*, ParamSet * *from*, int *setOnly*)

Utility for copying a ParamSet.

Any changes to ParamSet struct needs to be reflected in this function as well.

See also:

ParamSet

Parameters:

to Target ParamSet
from Source ParamSet
setOnly If true, copies only the parameters stored in p->set[]

4.4.2.9 void copyPoseCache (PoseCache * *to*, PoseCache * *from*)

Deep copy of a PoseCache data structure.

All new pointers.

Parameters:

to Target pose cache
from Source pose cache

4.4.2.10 void emptyFn (void * *data*)

This function does nothing.

An argument to HashFreeTable() - it makes sure we don't prematurely free sensitive data.

See also:

HashFreeTable()

Parameters:

data User data

4.4.2.11 double eval (ParamSet * *param*, OptData * *data*, int *index*, long int *epoch*)

Run the evaluation function on the given input example.

Deprecated. Use eval_constraint() instead.

See also:

eval_constraint()

Parameters:

param A parameter set

data Optimization data

index Input example index into optData->input[]

epoch Current optimization epoch

Returns:

Score

4.4.2.12 double eval_constraint (LigData * *ligand*, Protein * *protein*, long int *epoch*)

Wrapper for eval() determines what level of optimization will be used in scoring.

Parameters:

ligand A Ligand

protein A Protein

epoch Current epoch

Returns:

Score

4.4.2.13 double evalPoseCache (PoseCache * *cache*, Protein * *protein*, int *scoreMode*)

Return the highest scoring pose in the poseCache.

Parameters:

cache A pose cache

protein Protein data

scoreMode Level of optimization used in scoring: QUICK / THOROUGH / STATIC

4.4.2.14 void exportBestPoses (OptData * *data*)

Write out the best pose for each ligand, given the current parameters.

Parameters:

data Optimization data

4.4.2.15 void exportBestPoses_constraint (ConstraintData * *cd*, char * *optID*, int *constraintID*, FILE * *constraintFile*)

Write out the best poses found during optimization with the best parameters.

Traverses the constraint data structure.

Parameters:

cd A constraint

optID Optimization specific tag

constraintID Constraint specific tag

constraintFile Open constraint file pointer

4.4.2.16 unsigned int findLCM (unsigned int * *arr*, int *n*)

Find the least common multiple between a set of numbers.

- Find GCF
- LCM = i1 * i2 * ... * in / GCF

Parameters:

arr Array of positive, nonzero integers

n Array size

Returns:

Least common multiple

4.4.2.17 FILE * fireupDataLog (OptData * *data*, long int *epoch*)

Determine if this epoch will be logged in detail.

Open the log for writing if not already done so.

Parameters:

data Optimization data

epoch Current epoch

Returns:

Open file pointer to data log or NULL if no logging is necessary

4.4.2.18 void freeConstraintData (ConstraintData * *c*)

Free memory allocated to ConstraintData struct.

If the linked list is present, continue traversal.

Parameters:

c Constraint data

4.4.2.19 void freeLigData (LigData * *l*)

Free memory allocated to LigData struct.

Parameters:

l Ligand data

4.4.2.20 void freeOptData (OptData * *data*, double * *error*)

Free memory allocated to OptData struct.

Parameters:

data Optimization data

error Deprecated: array of errors

4.4.2.21 void freeParamLog (ParamLog * *log*)

Free memory allocated a ParamLog.

Parameters:

log A parameter log

4.4.2.22 void freeParamSet (ParamSet * *p*)

Free memory allocated to ParamSet struct.

Parameters:

p Parameter set

4.4.2.23 void freeParamStep (ParamStep * *p*, int *freePtr*)

Free memory allcoated a ParamStep.

Parameters:

p A parameter step

freePtr If true, free the pointer to p as well

4.4.2.24 void freePoseCache (PoseCache * *c*, int *freePoses*)

Free memory allocated to PoseCache struct.

Parameters:

c A pose cache

freePoses If true, free the poses within the pose cache as well

4.4.2.25 void freeProtein (Protein * *p*)

Free memory allocated to Protein struct.

Parameters:

p A Protein

4.4.2.26 void freeProteinData (ProteinData * *p*)

Free memory allocated to ProteinData struct.

Parameters:

p Protein data

4.4.2.27 void getNextParamStep (ParamLog * *paramLog*, ParamSet * *param*)

Move one parameter step forward.

Useful for:

See also:

redoOpt() Current step tracked in
ParamLog::currStep

Parameters:

paramLog ParamLog data

param Array of ParamSteps

4.4.2.28 unsigned int getNumConstraints (char * *constraintFile*, unsigned int * *nweights*)

Get the number of constraints and weights parsed from the constraintFile.

Parameters:

constraintFile Full pathname to a constraint file

nweights Number of weights parsed

Returns:

Number of constraints parsed

4.4.2.29 unsigned int getNumMolecules (char * *molArchive*)

Retrieve the number of molecules in a mol2 archive.

Reports the number of times the string '<TRIPOS>MOLECULE' is found in the file.

Parameters:

molArchive Full pathname to mol2 archive

Returns:

Number of molecules found

4.4.2.30 int getNumParam (FILE * *file*)

From first line of param.log, count the number of params.

param.log format:

"<paramIndex>\t<error>\t<param1>\t<param2> ... \t<paramN>"

Parameters:

file Open file pointer to a parameter log file

Returns:

Number of parameters

4.4.2.31 ParamStep * grabParamSteps (FILE * *file*, unsigned int *totalSteps*, int *nparam*)

Parse each parameter step from the parameter log.

Parameters:

file Open file pointer to a parameter log

totalSteps Number of parameter steps

nparam Number of parameters

Returns:

Newly allocated array of ParamSteps

4.4.2.32 ParamStep * grabProgressiveSteps (char * *paramLog*, int *nparam*, ParamStep ** *step*, unsigned int * *ntotal*, unsigned int * *ngood*)

Grab only the progressive parameter steps from a parameter log file.

Parameters:

paramLog Full pathname to a paramater log

nparam Number of parameters

step Storage for array of ParamSteps

ntotal Number of ParamSteps

ngood Number of good ParamSteps

Returns:

Newly allocated array of good ParamSteps

4.4.2.33 void initEpochData (double * *error*, unsigned int *errorSize*, ParamSet * *oldParam*, ParamSet * *newParam*, long int *epoch*)

Ready the parameters for this epoch.

Deprecated: reset error tracking.

Parameters:

error Deprecated: array of error values

errorSize Total number of resamplings for all constraints

oldParam Current best parameters

newParam Placeholder for our parameter changes

epoch Current optimization epoch

4.4.2.34 OptData * initOptData (char * *optID*, char ** *files*, int *nfiles*, double ** *error*, unsigned int * *errorSize*, ParamSet ** *newParam*)

Initialize the optimization data.

- Read in the files
- Initialize appropriate data structures
- Allocate memory for out and error

Constraint-based optimization

```
files[N]:
---------
0 constraint
1 param
```

Parameters:

optID Optimization specific tag

files Array of full path filenames

nfiles Number of files

error Deprecated: array of errors

errorSize Total number of resamplings for all constraints

newParam Placeholder for our parameter changes

Returns:

Newly allocated OptData

4.4.2.35 void initParam (OptData * *data*, char * *paramFile*)

Get initial values for ParamSet from a parameter file.

Will parse for minimum error line.

Parameters:

data Optimization data

paramFile Full pathname to a parameter file

4.4.2.36 int isUniquePose (PoseCache * *cache*, Conformer * *pose*)

Uniqueness test for a pose within a pose cache.

Assumes pose has been scored. Pose is unique if score is not seen in current cache. Could add rigor by requiring RMSD threshold as well.

Parameters:

cache Pose cache

pose A pose

Returns:

True if passes test

4.4.2.37 double loadParam (char * *filename*)

Parse the parameters from the standard .param file.

Assigns read parameters to the proper global variables.

Order of parameters in param file is important. Parameters must be available as global variables.

#define MAX_PARAMS 17 // (from original '96 Surflex paper)

- this should be set appropriately if using expanded paramSet (desolvation)

#define NO_OPT "NO_OPT"

- indicates this parameter will not be optimized, optimize value = initial value

Parameters:

filename Full pathname to a parameter file

Returns:

Error parsed from file

4.4.2.38 int lookupGroupWeight (HashTable * *table*, char * *constraintLine*)

Hash lookup of a weight assigned to a constraint group Useful while parsing:.

See also:

readConstraintFile()
readGroupWeight()

Parameters:

table HashTable mapping groupID => weight

constraintLine Line from constraint file

Returns:

Group weight if found, else parsing fails

4.4.2.39 void my_dock_list (LigData ** *ligData*, int *numLig*, Molecule * *protmol*, Molecule * *protomol*, Grid * *grid*, char * *logpath*, int *writeLog*)

Dock a list of molecules.

Identical to dock_list() in dock.c except:

- No multiproc
- Includes progress meter
- Scores stored in ligData suitable for calculating roc later
- Writes out only best conformation to results file

Operates on all ligands in ligData[]

- logfile, resultsFile are opened by caller
- polar scores returned are not scaled by sf_poz

Parameters:

ligData Array of ligands

numLig Number of ligands

protmol A protein molecule

protomol A protomol

grid Protein grid of cached interesting active site atoms

logpath Full pathname of log file to output

writeLog If true, write details to log file

4.4.2.40 void my_dock_list_tpfp (char * *truePath*, char * *decoyPath*, char * *protopath*, char * *protpath*, char * *logpath*)

Dock a list of molecules for screening enrichment.

Generate ROC statistics.

Parameters:

truePath Full pathname to TP ligand archive

decoyPath Full pathname to TP ligand archive

protopath Full pathname to protomol file

protpath Full pathname to protein file

logpath Full pathname to output log file

4.4.2.41 double my_score (char * *ligFile*, char * *proteinFile*, char * *protoFile*, char * *paramFile*, int *optdockFlag*)

Given files containing their conformation, score a protein and ligand.

Parameters:

ligFile Full pathname to a ligand file

proteinFile Full pathname to a protein file

protoFile Full pathname to a protomole file

paramFile Full pathname to a paramter file

optdockFlag If true, optimize the docking

4.4.2.42 PoseCache * newPoseCache (int *maxsize*)

Allocate memory for PoseCache struct.

Parameters:

maxsize Maximum number of poses in pose cache

Returns:

Newly allcoted PoseCache

4.4.2.43 double optFirstUpdate (Protein * *protein*, LigData * *ligand*, PoseCache * *temp*)

Optimize all poses, then add the best scoring pose if necessary.

Stronger test, guarantees a pose at the scoring extrema. Slower since we have to make a copy of the posecache first we're not keeping all optimized poses, just the best one. We will have snapshots of good poses as the parameters change, allowing us to make sure we're operating on the maximum of our pose-scoring curve.

Tasks:

- Optimize all poses in cache given *current parameters*
- Add the best scoring newPose if it is greater than any other current pose score in the cache
- Replacing the lowest scoring pose

Parameters:

protein A protein

ligand A ligand

temp Placeholder for a temporary pose cache while we do updates

4.4.2.44 double optimize (char * *optID*, char * *constraintFile*, char * *paramFile*)

Fully automated, constraint-based scoring function optimization.

Cycles between two search strategies: random walk and line search. Outputs a log of sampled parameters as well as the optimized parameters. For more hands-on optimization,

See also:

optParam_manual_wrapper().

Parameters:

optID Optimization specific tag

constraintFile Full pathname to a constraint file

paramFile Full pathname to a parameter file

4.4.2.45 double optimizeDock (Conformer * *ligand*, Molecule * *protein*, Grid * *grid*, int *mode*)

Score a protein and ligand conformation.

```
mode action
-------------------
0 static pose scoring
1 quick optimization of pose, then score
2 thorough optimization of pose, then score
```

Parameters:

ligand Ligand conformation

protein Protein conformation

grid Protein grid of cached interesting active site atoms

mode Scoring type; see above for details

Returns:

Score

4.4.2.46 unsigned int optParam (OptData * *data*, ParamSet * *newParam*, unsigned int *errorSize*, FILE * *paramlog*, long int * *epoch*, unsigned int *initBetter*)

Entry optimization function.

Presumes optData, param, error all setup prior to call.

Optimization loop:

- Take step in param space
- Score data
- Assess performance
- Check for stopping conditions

Parameters:

data Optimization data

newParam Placeholder for our parameter changes

errorSize Total number of resamplings for all constraints

paramlog Log tracks our param steps

epoch Current optimization epoch

initBetter Improvement tracking necessary to trigger updatePoses

Returns:

Number of epochs with MSE improvement

4.4.2.47 int paramWillBeOpt (ParamSet * *param*, int *index*)

Test if this parameter will be optimized.

Search the array of parameter indices to be optimized. Array search is a by product of data structure.

Parameters:

param A parameter set

index Parameter index to be tested

Returns:

True if passes test

4.4.2.48 void parseMoleculeNameField (LigData * *ligData*, char * *name*, char * *protein*)

Parse mol2 files that have standardized name field.

Format:

"tp/fp-moleculeName"

"protein filename name initscore initrank"

"tp/fp protein filename name initscore initrank"

Parameters:

ligData Ligand

name Molecule name field from mol2 file

protein Protein name

4.4.2.49 ConstraintData * readConstraint (char * *constraintLine*, char * *group*, int *groupWeight*)

Given a line from a constraintFile, initialize a ConstraintData structure.

Parameters:

constraintLine Line from a constraint file

group Group name

groupWeight Group weight

Returns:

Allocated ConstraintData

4.4.2.50 ConstraintData ** readConstraintFile (char * *constraintFile*, int * *ngroup*, unsigned int * *errorSize*)

Initialize the constraints data structure from the given file.

General file format:

"<protein> <proto> <constraint-type> <arg1> <arg2> <arg3>"

Parameters:

constraintFile Full pathname to a constraint file

ngroup Storage for number of constraint groups parsed

errorSize Total number of resamplings for all constraints

Returns:

Array of head nodes, each the beginning of a linked list representing a constraint group

4.4.2.51 void readConstraintGeomDecoy (ConstraintData * *cd*, char * *goodPosesFile*, char * *badPosesFile*)

Read a geometric constraint.

Constraint format:

protein protomol "geometric_decoy" correct.poses.mol2 bad.poses.mol2

Parameters:

cd Constraint

goodPosesFile Full pathname to a good ligand poses archive

badPosesFile Full pathname to a bogus ligand poses archive

4.4.2.52 void readConstraintROC (ConstraintData * *cd*, char * *tpArchiveFile*, char * *fpArchiveFile*)

Read a screening constraint.

Constraint format:

protein protomol "roc" tp.archive.mol2 fp.archive.mol2

Parameters:

cd Constraint

tpArchiveFile Full pathname to a TP ligand archive

fpArchiveFile Full pathname to a FP ligand archive

4.4.2.53 void readConstraintScore (ConstraintData * *cd*, char * *ligand*, char * *scoreType*, char * *score*)

Read score constraint.

Constraint format:

protein protomol ligand [=, <, >] score

Parameters:

cd Constraint

ligand Full pathname to a ligand archive

scoreType CONSTRAINT_SCORE_EQUAL or CONSTRAINT_SCORE_GREATER or CONSTRAINT_SCORE_LESSER

score Target score

4.4.2.54 void readGroupWeight (char * *weightLine*, HashTable * *table*)

Parse the group weight information from a line in a constraint file.

Hash the groupID => weight for lookup later.

Parameters:

weightLine Group-weight line from constraint file

table HashTable mapping groupID => weight

4.4.2.55 LigData ** readMol2Archive (char * *archive*, unsigned int * *nmol*, int *type*, char * *protein*, int *poseCacheOn*)

Read in a mol2 archive.

Parameters:

archive Full pathname to a ligand archive

nmol Storage for the number of ligands parsed

type Ligand type: [POS, NEG]

protein Protein name

poseCacheOn If on, each ligand will be allocated additional memory for a pose cache

Returns:

Array of initialized LigData

4.4.2.56 int readParam (FILE * *file*, OptData * *data*)

Read parameters from a paramfile.

Also read: "min error: N".

Differs from loadParam():

- Params are not assigned globally
- Handles some optimization bookkeeping:
  - param range
  - which params to ignore/optimize

Parameters:

file Open file pointer

data Optimization data

Returns:

True if read was successful

4.4.2.57 ParamLog * readParamLog (char * *paramLogFile*, ParamStep ** *allstep*, unsigned int * *ntotal*)

Read in a parameter log file.

- Read in each line of paramLog
- Sort by error
- Keep progress starting from initParam
- Save each positive parameter step

4.4.2.58 void redoOpt (char * *paramLogFile*)

Recapitulate an optimization run from a param.log.

Parameters:

paramLogFile Full pathname to a parameter log file

4.4.2.59 void repeatOptimize (char * *optID*, char * *constraintFile*, char * *paramFile*, int *nreps*)

Wrapper function to repeat optimize() calls.

Outputs each log and params found from each individual optimize() call as well as the best parameters found overall.

See also:

optimize()

Parameters:

optID Optimization specific tag

constraintFile Full pathname to a constraint file

paramFile Full pathname to a parameter file

nreps Number of times to call optimize()

4.4.2.60 void resetArr (double * *arr*, int *numElem*, double *val*)

Reset all elements of an arr to val.

Parameters:

arr Array of doubles
numElem Size of array
val Value to set

4.4.2.61 unsigned int resolveSamplingFrequency (ConstraintData ** *groupHead*, int *ngroup*)

Constraint weight arbitration.

Resample Frequency = LCM / n constraints in group * constraint weight.

Parameters:

groupHead Array of group head nodes
ngroup Number of group heads

Returns:

total number of resamples.

4.4.2.62 double rocSorted_generalized (LigData ** *ligand*, unsigned int *numLig*, unsigned int *ntp*, char * *logname*, char * *proteinName*)

Wrapper for ucsf-roc, outputs detailed log file.

See also:

compute_roc()
compute_ci()

Parameters:

ligand Array of ligands
numLig Number of ligands
ntp Number of TPs
logname Name of logfile to output, pass NULL to turn off logging
proteinName Name of the protein

Returns:

ROC area (1.0 is best, 0.5 is random performance)

4.4.2.63 void sanityCheck (FILE * *out*, OptData * *data*)

Output details regarding the optimization status.

Parameters:

out Open file pointer (can be stderr)

data Optimization data

4.4.2.64 void scoreConstraint (ConstraintData * *cd*, double * *error*, long int *epoch*, FILE * *datalog*)

Score by constraint type.

Parameters:

cd Constraint

error Storage for optimization MSE

epoch Current optimization epoch

datalog Open log file pointer

4.4.2.65 void scoreConstraintGeomDecoy (ConstraintData * *cd*, double * *error*, long int *epoch*, FILE * *datalog*)

Geometric constraint: bogus poses must always score worse than best good pose.

Parameters:

cd Constraint

error Storage for optimization MSE

epoch Current optimization epoch

datalog Open log file pointer

4.4.2.66 double scoreConstraintGroup (ConstraintData * *groupHead*, double * *error*, long int *epoch*, FILE * *datalog*)

Traverse the group, scoring each constraint.

Parameters:

groupHead Head of linked list representing a constraint group

error Storage for optimization MSE

epoch Current optimization epoch

datalog Open log file pointer

Returns:

Average error over the entire group

4.4.2.67 void scoreConstraintROC (ConstraintData * *cd*, double * *error*, long int *epoch*, FILE * *datalog*)

Screening constraint: maximize the scoring separation between TPs (actives) and FPs (decoys).

Parameters:

cd Constraint

error Storage for optimization MSE

epoch Current optimization epoch

datalog Open log file pointer

4.4.2.68 void scoreConstraintScore (ConstraintData * *cd*, double * *error*, long int *epoch*, FILE * *datalog*)

Scoring constraint: ligand must be = or $<$ or $>$ than the targetScore.

Parameters:

cd Constraint

error Storage for optimization MSE

epoch Current optimization epoch

datalog Open log file pointer

4.4.2.69 double scoreFirstUpdate (Protein * *protein*, LigData * *ligand*, PoseCache * *temp*)

Optimize only highest scoring pose, then add it to pose cache if necessary.

Faster since we can do this in place (no need for copy of entire posecache). Approximation for keeping our pose near the scoring extrema.

Tasks:

- Score all poses in cache given ∗current parameters∗
- Optimize best scoring pose
- Add the newPose if it is greater than any other current pose score in the cache
- Replacing the lowest scoring pose
- Return the best score in the cache

Parameters:

protein A protein

ligand A ligand

temp Placeholder for a temporary pose cache while we do updates

4.4.2.70 void setParam (ParamSet ∗ *param*, long int *epoch*)

Set the parameters.

Here is where the global variables relevant to scoring are set. Order of assignment is critical. Any changes to scoring function parameters should be reflected here as well.

Parameters:

param Parameter set

epoch Current optimization epoch

4.4.2.71 LigData∗ setupConstraintLigData (Molecule ∗ *mol*, char ∗ *archive*, int *type*)

4.4.2.72 Protein ∗ setupConstraintProtein (char ∗ *proteinFile*, char ∗ *protoFile*)

Initialize the Protein data structure.

Cache grid setup.

Parameters:

proteinFile Full pathname to protein file

protoFile Full pathname to protomol file

Returns:

Newly allocated Protein.

4.4.2.73 LigData * setupLigData (Molecule * *mol*, char * *archive*, int *type*, char * *protein*, int *poseCacheOn*)

Initialize LigData structure.

Parameters:

mol Ligand Molecule

archive Full pathname to a ligand archive

type Ligand type: [POS, NEG]

protein Protein name

poseCacheOn If on, each ligand will be allocated additional memory for a pose cache

Returns:

Newly allocated LigData

4.4.2.74 void setupParamLog (FILE ** *log*, ParamSet * *param*)

Setup the parameter log.

Open the log for editing. Write header information.

Parameters:

log Storage for a new log file pointer

param A parameter set

4.4.2.75 void setupSingleRun (OptData * *data*, ParamSet * *newParam*, long int *epoch*)

Setup optimization data specific to a single run's strategy.

- Stopping conditions
- Parameter search strategy

Parameters:

data Optimization data

newParam Placeholder for our parameter changes

epoch Current optimization epoch

4.4.2.76 double sqError (double *a*, double *b*)

Calculate square error of two numbers.

Parameters:

a value1
b value2

Returns:

(a-b)^2

4.4.2.77 void updateDataLog_constraint (OptData * *data*, char * *filename*)

Write to log the state of the data.

Parameters:

data Optimization data
filename Log filename

4.4.2.78 void updateDataLogGeomDecoy (FILE * *file*, ConstraintData * *cd*)

Write to the data log the current state of things for this geometric decoy constraint.

Parameters:

file Open log file pointer
cd Constraint

4.4.2.79 void updateDataLogROC (FILE * *file*, ConstraintData * *cd*)

Write to the data log the current state of things for an ROC constraint.

Assumes ROC computed ligand state is available in cd->temp[] as performed by:

See also:

rocSorted_generalized().

Parameters:

file Open log file pointer
cd Constraint

*

4.4.2.80 void updateDataLogScore (FILE * *file*, ConstraintData * *cd*)

Write to the data log the current state of things for this score constraint.

Parameters:

file Open log file pointer

cd Constraint

4.4.2.81 void updateParam (ParamSet * *oldParam*, ParamSet * *newParam*, long int *epoch*)

Update the old parameters by some increment based on the search strategy.

Increment size is specific to each parameter.

Parameters:

oldParam Current best parameters

newParam Placeholder for our parameter changes

epoch Current optimization epoch

4.4.2.82 void updateParamLog (FILE * *log*, long int *epoch*, ParamSet * *param*, double *error*)

Specialized printing for the parameter log.

Only parameters being optimized are written to file.

Parameters:

log Open log file pointer

epoch Current optimization epoch

param Current parameter set

error Current optimization error

4.4.2.83 double updatePoseCache (Protein * *protein*, LigData * *ligand*, PoseCache * *temp*)

Reoptimize the poses in the pose cache.

There are two ways of performing this task:

See also:

scoreFirstUpdate()
optFirstUpdate()

scoreFirstUpdate() is the default method due to its increased speed performance.

optFirstUpdate() is more rigorous but requires more compute time.

Parameters:

protein A protein

ligand A ligand

temp Placeholder for a temporary pose cache while we do updates

4.4.2.84 void updatePoses (OptData * *data*, unsigned int *nbetter*)

4.4.2.85 void updatePoses_constraint (OptData * *data*, unsigned int *nbetter*)

Update all poses in every ligand's pose cache.

Reoptimize the poses in each ligand's cache so that the optimal pose exists for the current set of parameters. Traverses the constraint data structure.

Parameters:

data Optimization data

nbetter Number of epochs with improvement in MSE

4.4.2.86 void updateStrategyStatus (FILE * *file*)

Write to file our strategy status.

Parameters:

file An open file pointer (can be stderr)

4.4.2.87 void writeFinalDataLog (OptData * *data*, char * *optID*, double *errorSize*, char * *runtime*, long int *epoch*)

Do a final scoring with best parameters and then write out the final log.

Parameters:

data Optimization data

optID Optimization specific tag

errorSize Total number of resamplings for all constraints

runtime Optimization runtime

epoch Number of optimization epochs

4.4.2.88 void writeNewConstraintFile (OptData * *data*, char * *optID*)

Write a new constraint file using newly exported best poses.

If we are exporting best poses found during optimization with the new parameter set, update the constraint file to point at the new poses.

Parameters:

data Optimization data

optID Optimization specific tag

4.4.2.89 void writeParam (ParamSet * *param*, FILE * *outfile*, char * *path*, int *newDefault*, OptData * *data*, long int *nepochs*, char * *inputFile*)

Write the parameters to a param file.

Parameters:

param Parameters to write out

outfile An open file pointer, pass NULL if starting a new file

path Full pathname for parameter file

newDefault If true, indicates writing a new default param file (optimized = init values)

data Additional optimization data

nepochs Number of optimization epochs

inputFile Constraint filename

4.4.2.90 void writeParamLog (FILE * *log*, long int *epoch*, ParamSet * *param*, double *error*)

If necessary, update the parameter log with optimization progress.

Parameters:

log Open log file pointer

epoch Current optimization epoch

param Current parameter set

error Current optimization error

4.4.2.91 void writeProgressiveSteps (char * *paramLogFile*, ParamStep * *allsteps*, unsigned int *nsteps*)

Output a log of only the progressive steps made in parameter search space.

Parameters:

paramLogFile Log filename

allsteps Array of ParamSteps

nsteps Number of ParamSteps

4.4.3 Variable Documentation

4.4.3.1 ParamStep* _ALL_STEP = NULL

Array of ParamSteps.

See also:

redoOpt()

4.4.3.2 int _EXPORT_POSES

If on, export optimized poses with best parameters.

4.4.3.3 void** _LIG_COPY = NULL

Data storage for roc calculation.

4.4.3.4 unsigned int _match = 0

OPT_FIRST_UPDATE stat tracking: number of times highest scoring pose pre-optimization remained highest scoring post-optimization.

4.4.3.5 double _MAX_EPOCHS

Stopping condition: maximum optimization epochs.

4.4.3.6 double _MIN_MSE

Stopping condition: minimum MSE.

4.4.3.7 unsigned int _N_ALL_STEP = 0

Number of parameter steps.

See also:

redoOpt()

4.4.3.8 int _N_CYCLE_NO_BETTER_BEFORE_STOP

Stopping condition: maximum number of search cycles without improvement.

4.4.3.9 int _NGOOD_2_OPTPOSE

Number of epochs with improvement before optimizing all poses in pose cache.

4.4.3.10 ParamLog* _PARAM_LOG = NULL

Parameter log.

See also:

redoOpt()

4.4.3.11 double _PARAM_RANGE

Random walk parameter range factor.

4.4.3.12 int _POSE_CACHE_SIZE

Number of poses to store in each ligand's pose cache.

4.4.3.13 int _RANDOM_SEARCH

If on, perform bounded random search.

4.4.3.14 int _RANDOM_WALK

If on, perform random walk search.

4.4.3.15 double _RW_MAX_EPOCHS_NO_IMPROVE

Stopping condition: maximum epochs of random walk search without improvement.

4.4.3.16 double _RW_STEP_SIZE

Random walk parameter increment size fraction.

4.4.3.17 int _SCORE_DATA

If on, only score data.

4.4.3.18 int _SINGLE_RUN

If on, perform only a single search run.

4.4.3.19 double _STEP_SIZE

Line search parameter increment size fraction.

4.4.3.20 unsigned int _total = 0

OPT_FIRST_UPDATE stat tracking: total number of times a pose cache was optimized.

4.4.3.21 int crlf_p

4.4.3.22 int max_rot

4.4.3.23 int n_dock_final

4.5 optimize.h File Reference

4.5.1 Detailed Description

Surflex-opt public interface.

```
#include "../sflib/surflex-public.h"
#include "optimize-types.h"
```

Functions

- double optimize (char *optID, char *constraintFile, char *paramFile)

  Fully automated, constraint-based scoring function optimization.

- void repeatOptimize (char *optID, char *constraintFile, char *paramFile, int nreps)

  Wrapper function to repeat optimize() calls.

- void redoOpt (char *paramLog)

  Recapitulate an optimization run from a param.log.

- OptData * initData (char **files, int nfiles, double **error, unsigned int *errorSize)
- void freeOptData (OptData *data, double *error)

  Free memory allocated to OptData struct.

- void initParam (OptData *data, char *paramFile)

  Get initial values for ParamSet from a parameter file.

- void setParam (ParamSet *param, long int epoch)

  Set the parameters.

- double loadParam (char *filename)

  Parse the parameters from the standard .param file.

- void updateParam (ParamSet *oldParam, ParamSet *newParam, long int epoch)

  Update the old parameters by some increment based on the search strategy.

- double eval (ParamSet *param, OptData *data, int index, long int epoch)

  Run the evaluation function on the given input example.

- double calcError (OptData *data, int index, double out)
- void updateDataLog (FILE *log, OptData *data, int index, double output, double error)
- void updateParamLog (FILE *log, long int epoch, ParamSet *param, double error)

  Specialized printing for the parameter log.

- double my_score (char *ligFile, char *proteinFile, char *protoFile, char *paramFile, int optdockFlag)

  Given files containing their conformation, score a protein and ligand.

- void my_dock_list_tpfp (char *truePath, char *decoyPath, char *protopath, char *protpath, char *logpath)

  Dock a list of molecules for screening enrichment.

4.5.2 Function Documentation

4.5.2.1 double calcError (OptData * *data*, int *index*, double *out*)

4.5.2.2 double eval (ParamSet * *param*, OptData * *data*, int *index*, long int *epoch*)

Run the evaluation function on the given input example.

Deprecated. Use eval_constraint() instead.

See also:

eval_constraint()

Parameters:

param A parameter set

data Optimization data

index Input example index into optData->input[]

epoch Current optimization epoch

Returns:

Score

4.5.2.3 void freeOptData (OptData * *data*, double * *error*)

Free memory allocated to OptData struct.

Parameters:

data Optimization data

error Deprecated: array of errors

4.5.2.4 OptData* initData (char ** *files*, int *nfiles*, double ** *error*, unsigned int * *errorSize*)

4.5.2.5 void initParam (OptData * *data*, char * *paramFile*)

Get initial values for ParamSet from a parameter file.

Will parse for minimum error line.

Parameters:

data Optimization data

paramFile Full pathname to a parameter file

4.5.2.6 double loadParam (char * *filename*)

Parse the parameters from the standard .param file.

Assigns read parameters to the proper global variables.

Order of parameters in param file is important. Parameters must be available as global variables.

#define MAX_PARAMS 17 // (from original '96 Surflex paper)

- this should be set appropriately if using expanded paramSet (desolvation)

#define NO_OPT "NO_OPT"

- indicates this parameter will not be optimized, optimize value = initial value

Parameters:

filename Full pathname to a parameter file

Returns:

Error parsed from file

4.5.2.7 void my_dock_list_tpfp (char * *truePath*, char * *decoyPath*, char * *protopath*, char * *protpath*, char * *logpath*)

Dock a list of molecules for screening enrichment.

Generate ROC statistics.

Parameters:

truePath Full pathname to TP ligand archive
decoyPath Full pathname to TP ligand archive
protopath Full pathname to protomol file
protpath Full pathname to protein file
logpath Full pathname to output log file

4.5.2.8 double my_score (char * *ligFile*, char * *proteinFile*, char * *protoFile*, char * *paramFile*, int *optdockFlag*)

Given files containing their conformation, score a protein and ligand.

Parameters:

ligFile Full pathname to a ligand file
proteinFile Full pathname to a protein file
protoFile Full pathname to a protomole file
paramFile Full pathname to a paramter file
optdockFlag If true, optimize the docking

4.5.2.9 double optimize (char * *optID*, char * *constraintFile*, char * *paramFile*)

Fully automated, constraint-based scoring function optimization.

Cycles between two search strategies: random walk and line search. Outputs a log of sampled parameters as well as the optimized parameters. For more hands-on optimization,

See also:

optParam_manual_wrapper().

Parameters:

optID Optimization specific tag
constraintFile Full pathname to a constraint file
paramFile Full pathname to a parameter file

4.5.2.10 void redoOpt (char * *paramLogFile*)

Recapitulate an optimization run from a param.log.

Parameters:

paramLogFile Full pathname to a parameter log file

4.5.2.11 void repeatOptimize (char * *optID*, char * *constraintFile*, char * *paramFile*, int *nreps*)

Wrapper function to repeat optimize() calls.

Outputs each log and params found from each individual optimize() call as well as the best parameters found overall.

See also:

optimize()

Parameters:

optID Optimization specific tag

constraintFile Full pathname to a constraint file

paramFile Full pathname to a parameter file

nreps Number of times to call optimize()

4.5.2.12 void setParam (ParamSet * *param*, long int *epoch*)

Set the parameters.

Here is where the global variables relevant to scoring are set. Order of assignment is critical. Any changes to scoring function parameters should be reflected here as well.

Parameters:

param Parameter set

epoch Current optimization epoch

4.5.2.13 void updateDataLog (FILE * *log*, OptData * *data*, int *index*, double *output*, double *error*)

4.5.2.14 void updateParam (ParamSet * *oldParam*, ParamSet * *newParam*, long int *epoch*)

Update the old parameters by some increment based on the search strategy.

Increment size is specific to each parameter.

Parameters:

oldParam Current best parameters

newParam Placeholder for our parameter changes

epoch Current optimization epoch

4.5.2.15 void updateParamLog (FILE * *log*, long int *epoch*, ParamSet * *param*, double *error*)

Specialized printing for the parameter log.

Only parameters being optimized are written to file.

Parameters:

log Open log file pointer

epoch Current optimization epoch

param Current parameter set

error Current optimization error

4.6 surflex-opt-main.c File Reference

4.6.1 Detailed Description

Command line entry point into surflex-opt.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "optimize.h"
```

Functions

- void printHelp (char *msg)
  Command line usage help.
- int main (int argc, char **argv)
  Command line argument handler.

Variables

- int _RANDOM_WALK = 0
  If on, perform random walk search.
- int _RANDOM_SEARCH = 0
  If on, perform bounded random search.
- int _SCORE_DATA = 0
  If on, only score data.
- int _SINGLE_RUN = 0
  If on, perform only a single search run.
- int _EXPORT_POSES = 0
  If on, export optimized poses with best parameters.
- double _MAX_EPOCHS = DEFAULT_MAX_EPOCHS
  Stopping condition: maximum optimization epochs.

- double _RW_MAX_EPOCHS_NO_IMPROVE = DEFAULT_RW_MAX_-EPOCHS_NO_IMPROVE

  Stopping condition: maximum epochs of random walk search without improvement.

- double _MIN_MSE = DEFAULT_MIN_MSE

  Stopping condition: minimum MSE.

- int _N_CYCLE_NO_BETTER_BEFORE_STOP = DEFAULT_N_CYCLE_-NO_BETTER_BEFORE_STOP

  Stopping condition: maximum number of search cycles without improvement.

- double _STEP_SIZE = DEFAULT_STEP_SIZE

  Line search parameter increment size fraction.

- double _RW_STEP_SIZE = DEFAULT_RW_STEP_SIZE

  Random walk parameter increment size fraction.

- double _PARAM_RANGE = DEFAULT_PARAM_RANGE

  Random walk parameter range factor.

- int _NGOOD_2_OPTPOSE = DEFAULT_NGOOD_2_OPTPOSE

  Number of epochs with improvement before optimizing all poses in pose cache.

- int _POSE_CACHE_SIZE = DEFAULT_POSE_CACHE_SIZE

  Number of poses to store in each ligand's pose cache.

4.6.2 Function Documentation

4.6.2.1 int main (int *argc*, char ** *argv*)

Command line argument handler.

Parameters:

argc Number of arguments

argv Array of arguments

Returns:

Exit code

4.6.2.2 void printHelp (char * *msg*)

Command line usage help.

Parameters:

msg Specific error message to output

4.6.3 Variable Documentation

4.6.3.1 int _EXPORT_POSES = 0

If on, export optimized poses with best parameters.

4.6.3.2 double _MAX_EPOCHS = DEFAULT_MAX_EPOCHS

Stopping condition: maximum optimization epochs.

4.6.3.3 double _MIN_MSE = DEFAULT_MIN_MSE

Stopping condition: minimum MSE.

4.6.3.4 int _N_CYCLE_NO_BETTER_BEFORE_STOP = DEFAULT_N_CYCLE_NO_BETTER_BEFORE_STOP

Stopping condition: maximum number of search cycles without improvement.

4.6.3.5 int _NGOOD_2_OPTPOSE = DEFAULT_NGOOD_2_OPTPOSE

Number of epochs with improvement before optimizing all poses in pose cache.

4.6.3.6 double _PARAM_RANGE = DEFAULT_PARAM_RANGE

Random walk parameter range factor.

4.6.3.7 int _POSE_CACHE_SIZE = DEFAULT_POSE_CACHE_SIZE

Number of poses to store in each ligand's pose cache.

4.6.3.8 int _RANDOM_SEARCH = 0

If on, perform bounded random search.

4.6.3.9 int _RANDOM_WALK = 0

If on, perform random walk search.

4.6.3.10 double _RW_MAX_EPOCHS_NO_IMPROVE = DEFAULT_RW_MAX_EPOCHS_NO_IMPROVE

Stopping condition: maximum epochs of random walk search without improvement.

4.6.3.11 double _RW_STEP_SIZE = DEFAULT_RW_STEP_SIZE

Random walk parameter increment size fraction.

4.6.3.12 int _SCORE_DATA = 0

If on, only score data.

4.6.3.13 int _SINGLE_RUN = 0

If on, perform only a single search run.

4.6.3.14 double _STEP_SIZE = DEFAULT_STEP_SIZE

Line search parameter increment size fraction.

4.7 ucsf-roc.c File Reference

4.7.1 Detailed Description

UCSF-ROC code.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "ucsf-roc.h"
```

Defines

- #define SMALL -10000000
- #define BIG 10000000

Functions

- double trap_area (int x1, int x2, int y1, int y2)

  Compute the trapezoidal area.

- void quicksort (double *array, int p, int r, int *labels)

  Quicksort an array.

- int partition (double *array, int p, int r, int *labels)

  Partition an array around a pivot.

- void swap (double *array, int left, int right, int *labels)

  Swap the positions of two elements in an array.

- double percentile (double *sortvals, int nvals, double pct)

  Grab the value that exists at a certain percentile in an array.

- int ucsf_roc_main (int argc, char **argv)

  Argument handling for ucsf-roc.

- void compute_ci (double *posvals, int ninput_posvals, double *negvals, int ninput_negvals, char *path, double ci_percent, int ci_num, double *ci_low, double *ci_high, double *mean_area)

  Compute ROC confidence intervals.

- void jain_error (char *string)

  Exit with detailed error message.

- double compute_roc (double *allvals, int *labels, int nallvals, char *prefix)

  Compute the ROC area and generate curves suitable for gnuplot.

Variables

- int nfpthresh = 3

  Number of FP thresholds at which we'll report the TP rate.

- double fpthresh [20] = {1.0, 5.0, 10.0}

  Array of FP thresholds at which we'll report the TP rate.

4.7.2 Define Documentation

4.7.2.1 #define BIG 10000000

4.7.2.2 #define SMALL -10000000

4.7.3 Function Documentation

4.7.3.1 void compute_ci (double * *posvals*, int *ninput_posvals*, double * *negvals*, int *ninput_negvals*, char * *path*, double *ci_percent*, int *ci_num*, double * *ci_low*, double * *ci_high*, double * *mean_area*)

Compute ROC confidence intervals.

Parameters:

posvals Array of TP scores

ninput_posvals Number of TP scores

negvals Array of FP scores

ninput_negvals Number of FP scores

path Full pathname to output the rocstats log file

ci_percent Percentage (e.g. 95) confidence interval to generate

ci_num Number of samplings ($\sim$ 1000) used to generate confidence interval

ci_low Storage for low end of computed confidence interval

ci_high Storage for high end of computed confidence interval

mean_area Storage for average ROC area found over all samplings

4.7.3.2 double compute_roc (double * *allvals*, int * *labels*, int *nallvals*, char * *prefix*)

Compute the ROC area and generate curves suitable for gnuplot.

Also histograms scores of the TPs and FPs in separate files.

Parameters:

allvals Array of ligand scores

labels Array of ligand labels (1 = TP, 0 = FP)

nallvals Number of ligands

prefix Specific prefix tag for all generated logs

Returns:

ROC area under the curve

4.7.3.3 void jain_error (char * *string*)

Exit with detailed error message.

Parameters:

string Error message

4.7.3.4 int partition (double * *array*, int *p*, int *r*, int * *labels*)

Partition an array around a pivot.

Helper function to quicksort().

See also:

quicksort()

Parameters:

array Array of doubles

p Left index

r Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.7.3.5 double percentile (double * *sortvals*, int *nvals*, double *pct*)

Grab the value that exists at a certain percentile in an array.

Parameters:

sortvals Array of doubles; will be sorted after this call

nvals Array size

pct Percentile of interest

Returns:

Value

4.7.3.6 void quicksort (double * *array*, int *p*, int *r*, int * *labels*)

Quicksort an array.

Parameters:

array Array of doubles

p Left index

r Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.7.3.7 void swap (double * *array*, int *left*, int *right*, int * *labels*)

Swap the positions of two elements in an array.

Helper function for parition().

See also:

partition()

Parameters:

array Array of doubles

left Left index

right Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.7.3.8 double trap_area (int *x1*, int *x2*, int *y1*, int *y2*)

Compute the trapezoidal area.

Parameters:

x1 Start x

x2 End x

y1 Start y

y2 End y

4.7.3.9 int ucsf_roc_main (int *argc*, char ** *argv*)

Argument handling for ucsf-roc.

Parameters:

argc Number of arguments

argv Array of command line arguments

Returns:

Exit code

4.7.4 Variable Documentation

4.7.4.1 double fpthresh[20] = {1.0, 5.0, 10.0}

Array of FP thresholds at which we'll report the TP rate.

4.7.4.2 int nfpthresh = 3

Number of FP thresholds at which we'll report the TP rate.

4.8 ucsf-roc.h File Reference

4.8.1 Detailed Description

UCSF-ROC public interface.

```
#include "../sflib/surflex-public.h"
```

Functions

- void jain_error (char *string)

  Exit with detailed error message.

- void quicksort (double *array, int p, int r, int *labels)

  Quicksort an array.

- int partition (double *array, int p, int r, int *labels)

  Partition an array around a pivot.

- void swap (double *array, int left, int right, int *labels)

  Swap the positions of two elements in an array.

- double percentile (double *vals, int nvals, double pct)

  Grab the value that exists at a certain percentile in an array.

- double compute_roc (double *allvals, int *labels, int nallvals, char *path)

  Compute the ROC area and generate curves suitable for gnuplot.

- void compute_ci (double *posvals, int ninput_posvals, double *negvals, int ninput_negvals, char *path, double ci_percent, int ci_num, double *ci_low, double *ci_hi, double *mean_area)

  Compute ROC confidence intervals.

4.8.2 Function Documentation

4.8.2.1 void compute_ci (double * *posvals*, int *ninput_posvals*, double * *negvals*, int *ninput_negvals*, char * *path*, double *ci_percent*, int *ci_num*, double * *ci_low*, double * *ci_high*, double * *mean_area*)

Compute ROC confidence intervals.

Parameters:

posvals Array of TP scores

ninput_posvals Number of TP scores

negvals Array of FP scores

ninput_negvals Number of FP scores

path Full pathname to output the rocstats log file

ci_percent Percentage (e.g. 95) confidence interval to generate

ci_num Number of samplings ($\sim$ 1000) used to generate confidence interval

ci_low Storage for low end of computed confidence interval

ci_high Storage for high end of computed confidence interval

mean_area Storage for average ROC area found over all samplings

4.8.2.2 double compute_roc (double * *allvals*, int * *labels*, int *nallvals*, char * *prefix*)

Compute the ROC area and generate curves suitable for gnuplot.

Also histograms scores of the TPs and FPs in separate files.

Parameters:

allvals Array of ligand scores

labels Array of ligand labels (1 = TP, 0 = FP)

nallvals Number of ligands

prefix Specific prefix tag for all generated logs

Returns:

ROC area under the curve

4.8.2.3 void jain_error (char * *string*)

Exit with detailed error message.

Parameters:

string Error message

4.8.2.4 int partition (double * *array*, int *p*, int *r*, int * *labels*)

Partition an array around a pivot.

Helper function to quicksort().

See also:

quicksort()

Parameters:

array Array of doubles

p Left index

r Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.8.2.5 double percentile (double * *sortvals*, int *nvals*, double *pct*)

Grab the value that exists at a certain percentile in an array.

Parameters:

sortvals Array of doubles; will be sorted after this call

nvals Array size

pct Percentile of interest

Returns:

Value

4.8.2.6 void quicksort (double * *array*, int *p*, int *r*, int * *labels*)

Quicksort an array.

Parameters:

array Array of doubles

p Left index

r Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.8.2.7 void swap (double * *array*, int *left*, int *right*, int * *labels*)

Swap the positions of two elements in an array.

Helper function for parition().

See also:

partition()

Parameters:

array Array of doubles

left Left index

right Right index

labels Integer labels assigned to each double (e.g. 1 is TP ligand, 0 is FP ligand)

4.9 utils.c File Reference

4.9.1 Detailed Description

Utility functions code.

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

Functions

- void exitError (char *msg, int code)
  Exit with error msg and code.
- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)

Copy source file to target.

- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.
- double myRound (double num, int power)
  round the number to the given power of 10.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.

Variables

- int crlf_p
  Newline status: if newlines are "\\r\\n" then true.

4.9.2 Function Documentation

4.9.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.9.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message
code Error code

4.9.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.9.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.9.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file
src Source file to copy

4.9.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open
mode fopen mode

Returns:

Newly opened file pointer if successful

4.9.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.9.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.9.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.9.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) > maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.9.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.9.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.9.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.9.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.9.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.9.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");

- }

3. And after loop completes for pretty printing:

   - fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

4.9.3 Variable Documentation

4.9.3.1 int crlf_p

Newline status: if newlines are "\\r\\n" then true.

4.10 utils.h File Reference

4.10.1 Detailed Description

Utility functions public interface.

```
#include "stdio.h"
```

Functions

- void * my_calloc (size_t num, size_t size, char *type)
  Tests that memory allocated is not null, otherwise exit with error.
- void exitError (char *msg, int code)
  Exit with error msg and code.
- int * newInt (int num)
  Return a newly allocated int with value num.
- double * newDouble (double num)
  Return a pointer to an allocated double with the given value.
- double myRand (double min, double max)
  Return a random value in the interval [min, max].
- double myRound (double num, int power)
  round the number to the given power of 10.
- int my_get_line (FILE *fd, char *string)
  Given an open file pointer, grabs the next line of text.
- void my_check_crlf (char *path)
  Checks for carriage return (\r) before linefeeds (\n) in the given file.
- int countWhiteSpace (char *string)
  Count the number of whitespaces " ", "\t" for this string.
- void removeWhitespace (char *string)
  Remove whitespace from front and back of string.
- void myStrCpy (char *target, char *src, int maxN)
  Copy a maximum of maxN chars from the src string to the target string.

- int parseFilename (char *filename, char **file, char **suffix)
  Given a filename, parse it reasonably.
- FILE * my_fopen (char *filename, char *mode)
  Combines fopen with standard error processing.
- void my_fcopy (char *tgt, char *src)
  Copy source file to target.
- double setupProgressMeter (double span, double *progress, int ntabify)
  Simple text progress bar with 20 clicks over the given span.
- void secondsToDays (double sec, char *buffer)
  Converts time in seconds to string with format: D:H:M:S.

4.10.2 Function Documentation

4.10.2.1 int countWhiteSpace (char * *string*)

Count the number of whitespaces " ", "\t" for this string.

Ignores appended whitespace at end of string.

Parameters:

string A string

Returns:

Number of whitespaces

4.10.2.2 void exitError (char * *msg*, int *code*)

Exit with error msg and code.

Parameters:

msg Error message

code Error code

4.10.2.3 void* my_calloc (size_t *num*, size_t *size*, char * *type*)

Tests that memory allocated is not null, otherwise exit with error.

4.10.2.4 void my_check_crlf (char * *path*)

Checks for carriage return (\r) before linefeeds (\n) in the given file.

Sets global flag crlf_p = 1. Useful for my_get_line().

Parameters:

path File to check for linefeed type.

4.10.2.5 void my_fcopy (char * *tgt*, char * *src*)

Copy source file to target.

Parameters:

tgt Target file

src Source file to copy

4.10.2.6 FILE* my_fopen (char * *filename*, char * *mode*)

Combines fopen with standard error processing.

Parameters:

filename Full pathname to file we want to open

mode fopen mode

Returns:

Newly opened file pointer if successful

4.10.2.7 int my_get_line (FILE * *fd*, char * *string*)

Given an open file pointer, grabs the next line of text.

Lines are delimited by [\n\r]. Newline delimiter is removed. Handles both linefeed forms correctly (\n vs \n\r). Handles parsing of empty lines by correctly by updating the read string to length 0.

Parameters:

fd Open file pointer from which we will read the string

string Allocated string buffer where we will store the read line

Returns:

Number of characters parsed

4.10.2.8 double myRand (double *min*, double *max*)

Return a random value in the interval [min, max].

Parameters:

min Minimum value

max Maximum value

Returns:

Random double value

4.10.2.9 double myRound (double *num*, int *power*)

round the number to the given power of 10.

Parameters:

num A double

power Round to this power of 10

4.10.2.10 void myStrCpy (char * *target*, char * *src*, int *maxN*)

Copy a maximum of maxN chars from the src string to the target string.

If len(src) > maxN, do the right thing.

Parameters:

target Allocated buffer to which we'll copy our source string

src String to copy

maxN Maximum number of characters to copy

4.10.2.11 double* newDouble (double *num*)

Return a pointer to an allocated double with the given value.

Useful as a hash value.

Parameters:

num Double to store in pointer

Returns:

Pointer to newly allocated Double

4.10.2.12 int* newInt (int *num*)

Return a newly allocated int with value num.

Useful as a hash value.

Parameters:

num Integer to store in pointer

Returns:

Pointer to newly allocated Integer

4.10.2.13 int parseFilename (char * *filename*, char ** *file*, char ** *suffix*)

Given a filename, parse it reasonably.

Find the file prefix and suffix. Return a ptr to the beginning of the filename (minus the path), the suffix, and the index of the prefix/suffix delimiter (a period).

Assumes the delimiter is the first period seen working backward when starting from the end of the string.

Parameters:

filename Filename to parse

file Pointer will point to beginning of filename

suffix Pointer will point to beginning of suffix

Returns:

Index of the last period before the suffix

4.10.2.14 void removeWhitespace (char * *string*)

Remove whitespace from front and back of string.

Parameters:

string A string

4.10.2.15 void secondsToDays (double *sec*, char * *buffer*)

Converts time in seconds to string with format: D:H:M:S.

Stores output to given pre-allocated buffer.

Parameters:

sec Seconds

buffer Allocated string buffer to store the string converted time

4.10.2.16 double setupProgressMeter (double *span*, double * *progress*, int *ntabify*)

Simple text progress bar with 20 clicks over the given span.

1. Function calling this will need:

- double fivepercent, progress;

2. Within loop that we're tracking progress, insert the following code: (If while loop, +1 may not be necessary depending on when incremented)

- // progress meter
- if (i + 1 >= (unsigned int)progress) {
- progress += fivepercent;
- fprintf(stderr, ".");
- }

3. And after loop completes for pretty printing:

- fprintf(stderr, "\n");

Parameters:

span Total by which we measure 100% complete

progress Pointer to counter that measures progress, initialized 5% into the future

ntabify Number of tabs to insert before progress meter print out

Returns:

Increment that represent 5% of progress

Publishing Agreement

It is the policy of the University to encourage the distribution of all theses and dissertations. Copies of all UCSF theses and dissertations will be routed to the library via the Graduate Division. The library will make all theses and dissertations accessible to the public and will preserve these to the best of their abilities, in perpetuity.

Please sign the following statement:

I hereby grant permission to the Graduate Division of the University of California, San Francisco to release copies of my thesis or dissertation to the Campus Library to provide access and preservation, in whole or in part, in perpetuity.

December 21st, 2007

Author Signature

Date