Title
3D Motifs as Signatures of Protein Function and Evolution

Author
Polacco, Benjamin John

Publication Date
2007-07-24

3D Motifs as Signatures of Protein Function and Evolution

by

Benjamin John Polacco

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

UNIVERSITY OF CALIFORNIA, SAN FRANCISCO

[redacted]

Acknowledgments

I would like to thank my research advisor Patricia Babbitt for her support and insight; both were essential for the completion of this work. The members of the Babbitt lab were always ready to help when asked, and offered much feedback through formal group meetings as well as informal discussions.

Thanks are also due to my orals and thesis committee for their guidance and criticisms of my ideas when I presented them. Several good ideas contained here can be attributed to the insights of my committee, while all the questionable ideas are strictly my own.

While I often think this could have been completed more quickly without the responsibilities of being a father to my family, these years would have been much less fulfilling without them. I especially thank my wife for her support of me emotionally, and the family financially throughout this work.

This dissertation is divided into an introduction, four chapters, and a conclusion. Three chapters are based on work that is either published already or expected to published soon. Chapter 1 is entirely my work, though some of its findings were published together with the work of Elaine Meng in the journal *Proteins*. The text of Chapter 2 is a reprint of the material as it appears in the journal *Bioinformatics.* The coauthor listed in that publication directed and supervised the research that forms the basis for this chapter. Chapter 4, is a manuscript that will be submitted for publication in a peer-reviewed journal.

Meng, E. C., B. J. Polacco and P.C. Babbitt. (2004). "Superfamily active site templates." Proteins 55(4): 962-76.

Polacco, B. J. and P. C. Babbitt (2006). "Automated discovery of 3D motifs for protein function annotation." Bioinformatics 22(6): 723-30.

Abstract: 3D Motifs as Signatures of Protein Function and Evolution

Benjamin Polacco

The ability to predict a protein's function from its structure is becoming more important with the increasing pace at which international structural genomics projects make structures available for proteins with no known function. The function of a protein is frequently determined by relatively small regions in an overall structure. This dissertation investigates signature 3D motifs, or small subsets of a protein's residues, that capture the critical structural determinants of function shared by an entire group of proteins. First, with an investigation of randomly selected 3D motifs I show that motifs built from important functional residues are better at identifying proteins to a superfamily with a common functional mechanism than any other motifs. Next I develop a genetic algorithm, named GASPS, that chooses a motif based on its ability to identify a group of proteins. I demonstrate its effectiveness on four divergent superfamilies, and a convergent group of serine proteases. Again, I demonstrate that the best motifs, as chosen by GASPS this time, contain known functional residues. Chapter 3 investigates the use of a geometrical statistical model to predict the number of expected random matches to a motif. This simple geometrical model performs very well overall, but it under-predicts matches to motifs that are the result of general physical and chemical characteristics of proteins, such as disulfide bridges and hydrophobic clusters. This model is rejected for its use in GASPS in favor of the original empirical method. Finally, I report a broad survey of signature 3D motifs, generated by applying GASPS to all available functionally similar and homologous groups of proteins. Motifs are mostly restricted to homologous groups, with a higher chance of a better motif in homologous and isofunctional groups. I report on general trends in structural conservation and find that catalytic, ligand binding, disulfide, and stabilized charged residues are over-represented among conserved motifs. Additionally, I find that glycines appear to be the most frequently conserved residue, especially important in ligand binding sites. This collection of motifs is useful for identification of function in unknown proteins, as well as describing trends in protein evolution.

Table of Contents

List of Figures

Chapter 1

Chapter 2

Chapter 2 Supplementary Materials

Chapter 3

Chapter 4

Appendix 3

List of Tables

Chapter 2

Chapter 2 Supplementary Materials

Chapter 3

Chapter 4

Introduction

As proteins are the major gene products that act in living cells, understanding the functions of proteins is a critical step in translating genomic sequences into useful biological knowledge relevant to the health sciences. With today's efforts in structural genomics that aim to provide for each protein a model of its shape or structure in a cell (Blundell et al. 2000), knowledge of a protein's structure is becoming a more common starting point for determining a protein's function (Teichmann et al. 2001; Watson et al. 2007). As different functions can be performed by proteins that have very similar overall structures and folds (Chothia 1992; Todd et al. 1999), it is clear that we have to look at fine-scale details or local protein structure to accurately describe a protein's function. Over evolutionary time, identical proteins can diverge to have very different sequences by the accumulation of random neutral changes that do not change function (neutral drift), but these proteins will still share whatever structural components have been critical to their function. Additionally, as proteins evolve to perform new functions they can make use of existing local structural features that contribute the same partial function to both the new and old functions (Gerlt et al. 2001; Bartlett et al. 2003). This explains, for the most part, why all members of a diverse group of proteins often make use of the same configuration of a small number of amino acids that can be directly related to function. We can use these clusters of amino acids, called three dimensional (3D) motifs, as signatures of function. This work investigates these signature 3D motifs to show how the identification and understanding of protein function can be advanced through these repeated structural elements.

Because 3D motifs are closely tied to the evolution of function, a study of 3D motifs also describes the manner in which protein function evolves. The evolution of new function proceeds through one of two paths tied to the existence of 3D motifs. First, as descendants of a single protein diverge in function, existing functional components can be entirely replaced by new functional components so that no 3D motif will persist between modern day proteins. On the other hand, as new functions evolve, proteins can make use of existing functional components to perform one or more components in the overall function. If across these different functions, the same functional component is reused, a 3D motif will persist in modern day proteins. 3D motifs can also be present in convergent proteins, those that perform the same function but have no common ancestor (Dodson et al. 1998). If we observe frequent cases of convergent motifs this is evidence that the possible ways any proteins can evolve to perform a single function are limited.

Much work has been done by others examining 3D motifs. Studies have shown their effectiveness on a handful of cases (Wallace et al. 1996; Fetrow et al. 1998; Russell 1998), tools have been developed that can search protein structures for matches to motifs (Artymiuk et al. 1994; Kleywegt 1999; Barker et al. 2003), and motifs are being collected from literature descriptions of enzyme active sites (Torrance et al. 2005). Still, no study has yet systematically asked on how many and on what types of various protein groups can we use signature 3D motifs. This dissertation extends our knowledge of 3D motifs by inventing a novel method for discovering signature 3D motifs and applying this method to a large set of protein groups. This generates a set of motifs that are not only useful for protein annotation, but because they were systematically generated, provide an even and unbiased picture of the distribution of 3D motifs and patterns within them. Specifically, we see that homology is the most important generator of signature 3D motifs, but functional diversity also plays a role. Though groups with diverse functions and signature 3D motifs are not uncommon, homologous groups with many varied functions are less likely to have a signature motif.

Chapter 1 Summary

The work I describe in Chapter 1 lays the foundation for my method, and demonstrates how significant findings and research paths are often stumbled upon by accident. I worked together with Elaine Meng, who was investigating signature 3D motifs in the active sites of enzyme superfamilies (Meng et al. 2004). While Elaine assembled motifs from residues known to be functionally important for the superfamilies, I performed the control study to showed that motifs based on this expert knowledge identified the superfamily better than motifs assembled from randomly chosen residues. To make a more compelling comparison, I tested constraining the residues in the random motifs by distance from each other, then conservation, and then residue type. I added an automated system for allowing position specific substitutions based on a multiple sequence alignment. While these increased the quality of the randomly generated motifs, the published result from this work remained that the motifs built from functional knowledge always outperformed the automatically generated random motifs (Meng et al. 2004). This same result viewed from a slightly different angle would provide the inspiration that led to this entire dissertation: with a few simple constraints, a random guess could produce motifs that begin to approach the quality of expert derived motifs.

Chapter 2 Summary

This earliest work not only provided the insight that would lead to the development of my method, but also provided most of the software development. My method was given the acronym GASPS for Genetic Algorithm Search for Patterns in Structures. A genetic algorithm develops solutions to problems by choosing from among a set of guesses the best ones, then making new guesses by adding to, deleting from, or recombining the best guesses made so far. Using most of the random motif generation system I presented in Chapter 1 to create the first guesses, I added a method for measuring performance of motifs, a system to alter and recombine motifs, and an iterative process. This resulted in a version of GASPS that I described in a published manuscript (Polacco et al. 2006), included here as Chapter 2. As an alternative to building motifs from often-limited expert knowledge, GASPS identifies patterns of 3 to 10 residues that maximize function prediction. The unbiased approach of GASPS allowed us to test the assumption that residues that provide function are the most informative for predicting function. I applied GASPS to superfamilies with varied functions as well as the serine proteases, an example of convergent evolution of active sites (Dodson et al. 1998). The motifs found by GASPS are as good at function prediction as 3D motifs based on expert knowledge. The GASPS motifs with the greatest ability to predict protein function consist mainly of known functional residues.

Chapter 3 Summary

In an effort to improve GASPS, I investigated the theoretical statistics of random matches to 3D motifs, or false positives. GASPS seeks to find a motif for a group where all group members match within a deviation threshold stringent enough to make random matches to unrelated proteins rare. The original GASPS determines this threshold empirically, by searching for matches to each candidate motif among all non-group proteins that it should not match. The work described in Chapter 3, answers whether this empirical distribution of matches is necessary or instead is a theoretical statistical model of matches to 3D motifs sufficient. Computing the empirical distribution takes more time than any other GASPS step, so if it could be replaced it would significantly reduce the computing time necessary to generate motifs with GASPS. I show how the scoring function that GASPS uses to rank motifs can be modified to use a statistical model of motif matches developed by Stark et al. (2003). Qualitatively, motifs generated by this faster GASPS are very similar to the original GASPS, with similar rates of overlap with functionally significant residues. However, these motifs fail to identify new structures to the appropriate group with the same accuracy. This decreased accuracy is due more to false positives than false negatives, indicating the motifs are not as unique as the model would predict. This results from the use of a solely geometrical model that cannot account for common physically favorable interactions frequently observed across various protein groups, such as salt bridges or disulfides. This makes the faster GASPS a useful tool for discovering a motif that is well conserved by a group, but not for generating motifs useful for annotation of new structures. This faster GASPS was not used for any other work described here.

Chapter 4 Summary

The pieces are now in place to apply GASPS across as much of the protein universe as possible in order to generate as many signature 3D motifs as possible. Doing so allows for an examination of the evolution of fine scale protein structure by determining how widespread are conserved 3D motifs, and what structural features tend to be conserved. I apply GASPS to homologous superfamilies and families in the Structural Classification of Proteins (SCOP) (Murzin et al. 1995), as well as isofunctional groups defined by the Gene Ontology (GO) (Ashburner et al. 2000). I find that non-homologous but isofunctional groups do not commonly share a motif. This suggests that most protein functions, at least as they are commonly described, can be accomplished by very different means in unrelated proteins. Homologous groups more often share a conserved motif, with about one third of all SCOP groups showing a strongly conserved motif. Many of these superfamilies with strong motifs have very diverse functions, revealing where evolution has reused functional components to produce different overall reactions. The remaining two thirds of groups with less-conserved motifs reveal that evolution of new functions in homologous groups is not usually constrained to maintain the positions of a critical set of residues.

These motifs also allow us to examine what features are among the most conserved. Again, we see a strong relationship between motifs and function. The motifs frequently overlap with known catalytic, metal and other ligand binding sites. Additionally, disulfides as well as stabilized charged residue pairs are frequent components of the most conserved motifs. Residue distribution among the motifs is mostly as expected based on these common features: cysteine, histidine, aspartate and glutamate are among the most frequent. More surprisingly, glycine, leucine and proline are ranked first, fourth and seventh, respectively, among the most frequent motif residues. The dominant role of leucine can be attributed mostly to its high frequency among the entire proteins. Glycine is well conserved where its unique backbone angles and space allowances (Jornvall et al.

1984; Dym et al. 2001) are critical for function. The unique geometry afforded glycine seems especially important at binding sites: glycines in motifs show the greatest rate of non-metal ligand interaction among all residue types.

To maximize the impact of this work, I have made available the motifs generated in this broad study via a web resource named GASPSdb (http://gaspsdb.rbvi.ucsf.edu). The motifs at this site can be searched, browsed or downloaded. One search capability enables users to search for matches to the GASPS motifs among a protein structure they can provide or choose from the Protein Data Bank (PDB) (Berman et al. 2000). Because each motif is generated to be a signature motif for a functional or homologous group, a matching motif indicates that the new structure is a likely member of the group, and the matched residues are likely to be important for the protein's function. I show that the GASPSdb resource provides a greater coverage than other available 3D motif resources (Stark et al. 2003; Torrance et al. 2005). It also proves effective on low quality structural models computed from homology. This effectiveness on homology models is very important for the description of function in the homology models that structural genomics aims to make possible (Blundell et al. 2000).

Conclusions

This study has grown from its beginnings where its goal was to merely show the ineffectiveness of randomly chosen motifs, to show how when combined with an effective selection and recombination process those same random motifs can become useful signatures of protein function and evolution. While I generated a large number of signature motifs that will enable us to more accurately annotate structures, I find that not all groups can be identified by a signature motif. The distributions of these signature motifs represent just a single but useful view on the evolution of protein structure and function at a fine scale. We observe that evolution has used both schemes I presented regarding function and local structure. I identify both the homologous groups that have re-used functional features for multiple different overall functions, as well as groups which keep no single functional feature as they evolve to perform new overall functions. There are many known (and probably unknown) protein groups with insufficient structures for GASPS to work on effectively. As new structures are solved, an automated process like GASPS is well suited to continue to analyze new groups and new structures.

References

Artymiuk, P. J., A. R. Poirrette, et al. (1994). "A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures." J Mol Biol **243**(2): 327-44.

Ashburner, M., C. A. Ball, et al. (2000). "Gene ontology: tool for the unification of biology. The Gene Ontology Consortium." Nat Genet **25**(1): 25-9.

Barker, J. A. and J. M. Thornton (2003). "An algorithm for constraint-based structural template matching: application to 3D templates with statistical analysis." Bioinformatics **19**(13): 1644-9.

Bartlett, G. J., N. Borkakoti, et al. (2003). "Catalysing new reactions during evolution: economy of residues and mechanism." J Mol Biol **331**(4): 829-60.

Berman, H. M., J. Westbrook, et al. (2000). "The Protein Data Bank." Nucleic Acids Res **28**(1): 235-42.

Blundell, T. L. and K. Mizuguchi (2000). "Structural genomics: an overview." Prog Biophys Mol Biol **73**(5): 289-95.

Chothia, C. (1992). "Proteins. One thousand families for the molecular biologist." Nature **357**(6379): 543-4.

Dodson, G. and A. Wlodawer (1998). "Catalytic triads and their relatives." Trends Biochem Sci **23**(9): 347-52.

Dym, O. and D. Eisenberg (2001). "Sequence-structure analysis of FAD-containing proteins." Protein Sci **10**(9): 1712-28.

Fetrow, J. S. and J. Skolnick (1998). "Method for prediction of protein function from sequence using the sequence-to-structure-to-function paradigm with application to glutaredoxins/thioredoxins and T1 ribonucleases." J Mol Biol **281**(5): 949-68.

Gerlt, J. A. and P. C. Babbitt (2001). "Divergent evolution of enzymatic function: mechanistically diverse superfamilies and functionally distinct suprafamilies." Annu Rev Biochem **70**: 209-46.

Jornvall, H., H. von Bahr-Lindstrom, et al. (1984). "Extensive variations and basic features in the alcohol dehydrogenase-sorbitol dehydrogenase family." Eur J Biochem **140**(1): 17-23.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Mol Biol **285**(4): 1887-97.

Meng, E. C., B. J. Polacco, et al. (2004). "Superfamily active site templates." Proteins **55**(4): 962-76.

Murzin, A. G., S. E. Brenner, et al. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures." J Mol Biol **247**(4): 536-40.

Polacco, B. J. and P. C. Babbitt (2006). "Automated discovery of 3D motifs for protein function annotation." Bioinformatics **22**(6): 723-30.

Russell, R. B. (1998). "Detection of protein three-dimensional side-chain patterns: new examples of convergent evolution." J Mol Biol **279**(5): 1211-27.

Stark, A. and R. B. Russell (2003). "Annotation in three dimensions. PINTS: Patterns in Non-homologous Tertiary Structures." Nucleic Acids Res **31**(13): 3341-4.

Stark, A., S. Sunyaev, et al. (2003). "A model for statistical significance of local similarities in structure." J Mol Biol **326**(5): 1307-16.

Teichmann, S. A., A. G. Murzin, et al. (2001). "Determination of protein function, evolution and interactions by structural genomics." Curr Opin Struct Biol **11**(3): 354-63.

Todd, A. E., C. A. Orengo, et al. (1999). "Evolution of protein function, from a structural perspective." Curr Opin Chem Biol **3**(5): 548-56.

Torrance, J. W., G. J. Bartlett, et al. (2005). "Using a Library of Structural Templates to Recognise Catalytic Sites and Explore their Evolution in Homologous Families." J Mol Biol **347**(3): 565-81.

Wallace, A. C., R. A. Laskowski, et al. (1996). "Derivation of 3D coordinate templates for searching structural databases: application to Ser-His-Asp catalytic triads in the serine proteinases and lipases." Protein Sci **5**(6): 1001-13.

Watson, J. D., S. Sanderson, et al. (2007). "Towards fully automated structure-based function prediction in structural genomics: a case study." J Mol Biol **367**(5): 1511-22.

Chapter 1: Random Motifs and Superfamily Active Site Templates

Introduction

When this work was started there were already multiple studies showing that specific active site 3D motifs could be used successfully to identify specific protein functions (Artymiuk et al. 1994; Wallace et al. 1997; Fetrow et al. 1998; Russell 1998; Kleywegt 1999). A major appeal of 3D motifs is that they provide a direct link between structural details and function in a way that sequence based or whole protein fold-based comparisons could not. In addition to a detailed view on structure, accurately describing the linkage between structure and function can benefit from a detailed view of protein function. Instead of treating an enzyme's function as a single unit, it can be broken down into smaller mechanistic steps, and superfamilies of enzymes can share one or more functional steps (Gerlt et al. 2001; Babbitt 2003). The work I present here was my part of a collaboration to show that superfamilies of enzymes, and therefore just the smaller element of function that they share can be identified by a single motif (Meng et al. 2004). Superfamily active site template was the name given to a 3D motif that is shared among members of a diverse superfamily that are responsible for the superfamily's shared function. While previous studies of 3D motifs have constructed motifs based on knowledge of functional residues, none looked specifically at the question of whether there were other informative residues—residues that uniquely identified the group of proteins. Investigating this question was my contribution to the study. While my collaborator did the traditional motif-building jobs of compiling lists of functional residues and their similarities between related proteins, I constructed thousands of motifs at random to determine whether the functional residues were required or whether there were other residues that were conserved in three dimensions across a superfamily.

The work I present in this chapter is an important component of the published superfamily study. Additionally, it provides an analysis of random motifs that would guide the development of my technique, named GASPS, described in Chapter 2. With GASPS I make use of random guesses in a genetic algorithm, so the knowledge of which constraints can lead to better random guesses and the ways in which partial solutions score compared to an overall solution are important.

Portions of this work were published previously in the journal *Proteins* (Meng et al. 2004).

Materials and Methods

Motif Searches

Active site template searching was performed with SPASM (Kleywegt 1999). A motif is supplied to SPASM as a file containing the atomic coordinates of the residues of interest. These coordinates are taken from the original Protein Data Bank (PDB) (Berman et al. 2000) file of each source structure. SPASM allows explicit specification of the residue types that can match each motif residue, referred to as substitutions later. The α-carbon (CA) and computed side-chain centroid (SC) are used to describe each motif residue. The internal CA-CA and SC-SC distances of the motif and each candidate match are compared, and candidate matches are pruned if they exceed user-specified maximum deviations, in our case the maximum CA-CA distance deviation was set to 5.0 Å, and the maximum SC-SC distance deviation was 3.8 Å. The remaining candidate matches are reoriented onto the motif and those fulfilling a user-specified RMSD cutoff are saved (3.2 Å). Thus, the input parameters include motif coordinates, allowed substitutions, a maximum CA-CA deviation cutoff, a maximum SC-SC deviation cutoff, a maximum RMSD cutoff, and what database to search. SPASM-searchable databases are derived directly from PDB files, but have been preprocessed down to the CA and computed SC coordinates for each residue. The preprocessing program, MKSPAZ, is available along with SPASM from the Uppsala Software Factory (http://xray.bmc.uu.se/usf/index.html).

Structure Libraries

Motif sensitivity and specificity was evaluated by searching a sequence-unique subset of the PDB; this database, spasm100, can be downloaded from the Uppsala Software Factory (http://xray.bmc.uu.se/usf/index.html). The July 2002 version of spasm100 (8255 entries, including 22 true positive enolase superfamily members) was used.

Calculating Conservation

Conservation of positions in a protein structure were calculated from a multiple sequence alignment generated by BLAST (Altschul et al. 1997) with default values against a non redundant protein sequence database, nrdb90 (Holm et al. 1998). Conservation was calculated from the multiple sequence alignment by a method that weights to reduce the effects of redundancy, considers conservative substitutions based on a substitution matrix, and penalizes gaps (Valdar 2002).

Results

Scores of randomly generated motifs

The superfamily motif derived from knowledge of functionally important residues from mandelate racemase (PDB id 2mnr) performed better than motifs from other available structures at identifying superfamily members with high sensitivity and specificity (Meng et al. 2004). Did the functional information identify the best residues for a motif, or could other motifs perform as well? To answer this, I generated motifs at random by selecting five residues entirely at random from the mandelate racemase structure. Each motif was scored by calculating the area under an ROC plot to 10 false positives based on the root mean squared deviation (RMSD) between the motif and its match in the structure (ROC10). All scores were normalized so that the maximum allowable ROC10 score was set to 1.0, the score that implies all superfamily structures match at a lower RMSD than any false positive. The vast majority of about 500 randomly generated motifs do not have an ROC10 area greater than 0.18 (Figure 1). This score corresponds to matching only the four superfamily structures that are most similar to 2mnr.

Most of the above randomly generated motifs appear very different from what commonly used 3D motifs look like. In an effort to make a more compelling comparison between randomly generated motifs, and those based on expert knowledge, I tested applying constraints on the generation of motifs. 3D motifs are typically composed of residues that are known to interact, so they must be close in space. The first constraint I applied to make the random motifs look more like typical motifs was therefore to restrict the residues to a 7.5 Å neighborhood, measured at their α-carbon, of an initial chosen residue. By itself, this restriction offered only a slight improvement to the ROC10 scores of the generated motifs.

A good 3D motif is maintained by evolution in all group structures, therefore its residues cannot be among the most variable in close relatives. The next constraint I tested was then to eliminate the most variable or least conserved residues observed in close homologs. Just eliminating residues with conservation below 0.6 showed a significant improvement in ROC10 areas, with no other constraints.

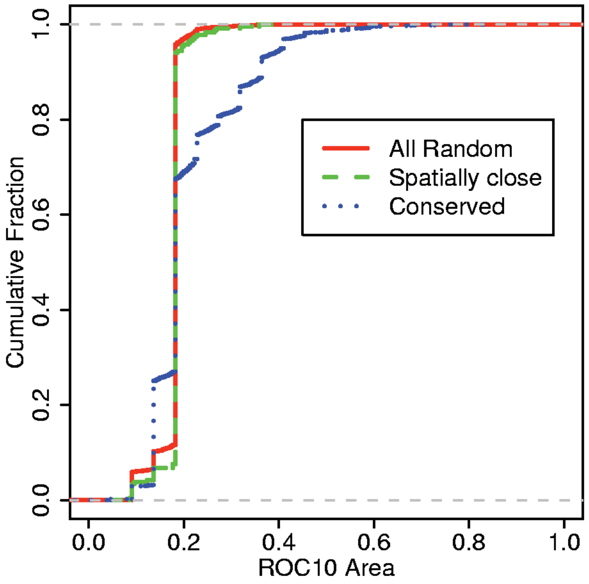


Figure 1. Cumulative histograms of scores of randomly generated motifs. The red line, “All Random”, represents 5331 motifs of five residues chosen entirely at random from a single mandelate racemase structure. The green line, “Spatially Close”, represents 237 similar motifs with the only constraint that residues within a single motif are restricted to lie within a 7.5 Å neighborhood. The blue line, “Conserved”, represents 2825 random motifs with the only constraint that all residues must have a conservation score (see Methods) greater than 0.6.

While the spatial constraint by itself showed little effect, putting the conservation and spatial constraints together resulted in an even greater improvement in ROC10 areas (Figure 2). This effect is strongest when the residues are maintained within a 7.5 Å neighborhood, compared with larger neighborhoods. By enforcing such a small neighborhood, I significantly decrease the number of possible random motifs because each residue only has a very limited set of residues it could build a motif with. This greatly minimizes the number of motifs that many conserved residues could be a part of because they do not cluster spatially with large numbers of other conserved residues. Others have actually used clusters of sequence-conserved residues on a protein structure to identify functionally important residues (Lichtarge et al. 1996). These constraints used here significantly enrich the available residues with functional residues, which can explain the increase in ROC10 areas for the smaller neighborhoods.

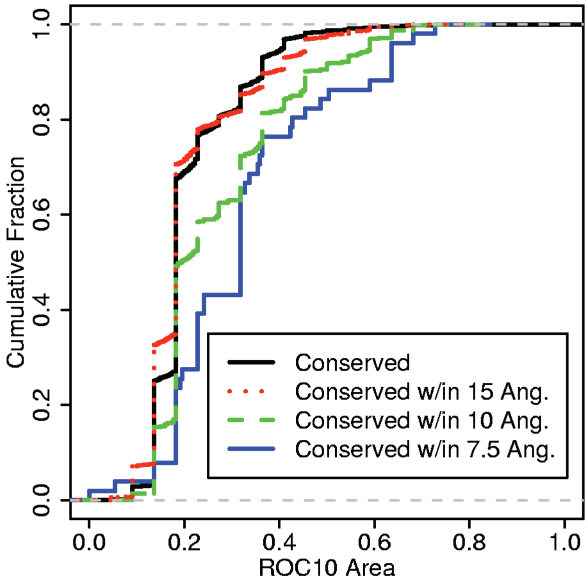


Figure 2. Cumulative histograms of scores of conserved and close random motifs. The black line, labeled "Conserved", is identical to the same-labeled data shown in Figure 1. The remaining lines show the effects of adding an additional constraint that all residues must lie in a 7.5, 10, or 15 Å neighborhood. These lines represent 51, 496 and 3093 motifs, respectively.

Substitutions in random motifs

While the simple constraints show improvement in the scores of random motifs, none performed as well as the motif based on expert knowledge. This difference is, in large part, due to the position-specific substitutions allowed in the expert motif. Three of the five residues in the expert derived motif allow a specific list of substitutions, and these substitutions are important for the high score of the motif. Not allowing these substitutions lowers its ROC10 area from 0.97 to 0.27. To provide randomly generated motifs this same flexibility, I allowed for position-specific substitutions chosen from the same multiple sequence alignment I used to measure conservation. Positions with poor conservation would have a very long allowable substitution list and could match most any residue in any protein, so it only makes sense to use conserved residues with this substitution scheme. Choosing substitutions from the BLAST-generated multiple sequence alignments showed a large shift in the middle of the distribution to higher ROC10 areas, but it did not change the maximum score (Figure 3). The sequences in the BLAST alignment are all much more similar than the most distant relatives of mandelate racemase in the superfamily. These substitutions allow more frequent matching of the relatively close structures, but not the more distant ones. Capturing the substitutions necessary to match more distant relations will require a multiple sequence alignment that includes sequences that are more distant. Alignments of entire superfamilies are not accurately generated by automatic methods, but the use of a manually curated multiple sequence alignment (Babbitt et al. 1996) shows an increase in the maximum scores achieved by randomly generated motifs.

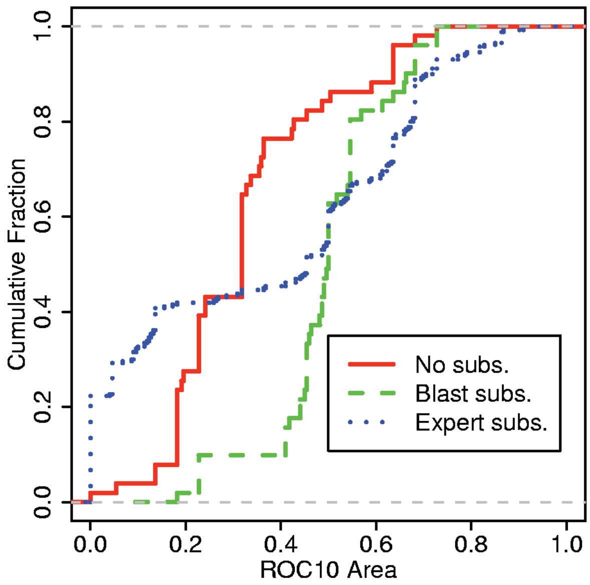


Figure 3. Cumulative histograms of scores of random motifs with allowed substitutions.
Red, “No subs.” line is identical to blue “Conserved w/in 7.5 Ang.” line in Figure 2. All motifs shown are constrained by conservation and spatial proximity (7.5 Å neighborhood). Green, “Blast subs.” line is the identical set of motifs with substitutions chosen from a BLAST-generated alignment. Blue, “Expert subs.” line represents 260 motifs generated identically except that the alignment is a manually curated superfamily alignment.

Limiting Residue Types in Random Motifs

One notable feature of the distributions of motifs based on very diverse sequence alignments is the number of motifs with ROC10 areas at 0. Most of these motifs are sensitive enough to match the four structures that are very similar to 2mnr with low RMSD, but they also match many false positives at equivalently low or lower RMSD. Inspection of these motifs shows that these are composed mostly of hydrophobic residues that are freely substituted by other hydrophobic residues, especially at great evolutionary distances. It appears that matching a hydrophobic cluster is very easy among unrelated proteins. Furthermore, most previously described motifs and catalytic sites are composed of polar residues. As a final constraint to test, I restricted the motifs to use only the polar residues. This eliminated the large number of motifs that score at 0.0, and shifted the entire distribution to the higher ROC10 areas.

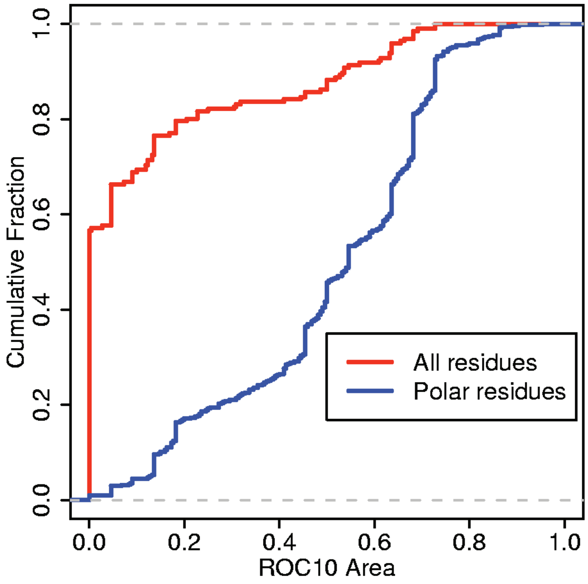


Figure 4. Cumulative histograms of scores of random motifs with only polar residues.
All motifs were generated with conservation and spatial proximity constraints (12 Å), as well as substitutions chosen from a manually curated superfamily multiple sequence alignment. The motifs represented by the blue, “Polar residues” line were further constrained to include only polar residues.

Important Residues in Motifs

The analysis so far has focused mostly on the scores of motifs and less on the features of the motifs that contribute to the score. I have shown that no randomly generated motifs have classification ability as high as a motif based on the functional site, but many still have high classification ability. Are there other regions of the protein with high classification ability that are not in the active site? I examined the residues used in motifs constrained by only conservation to find other residues that could contribute to high scores. Figure 5 shows each residue and an ROC10 area for each motif that contains it. While a functional residue is not sufficient for a high scoring motif–the spread of scores for all positions goes from close to zero to near the maximum for that position, at least some functional residues appear necessary for the highest scoring motifs. The highest-scoring randomly generated templates were very similar to the manually chosen template; all included at least one of the metal ligands, and most included two. Three of the five residues in the expert motif appear among the top scoring motifs, two metal binding ligands and a base. Six other residues appear in top-scoring motifs. Half are close sequence neighbors of these three functional residues. The remaining three top scoring residues are glycines that are more distant from the active site, though they always occur in top scoring motifs with at least one of the expert motif residues. As a final demonstration of the importance of these three functional residues to high scoring motifs, the ROC10 score for motifs built with only four of the expert motif residues show the greatest decline when the left-out residue is one of these three (Figure 6).

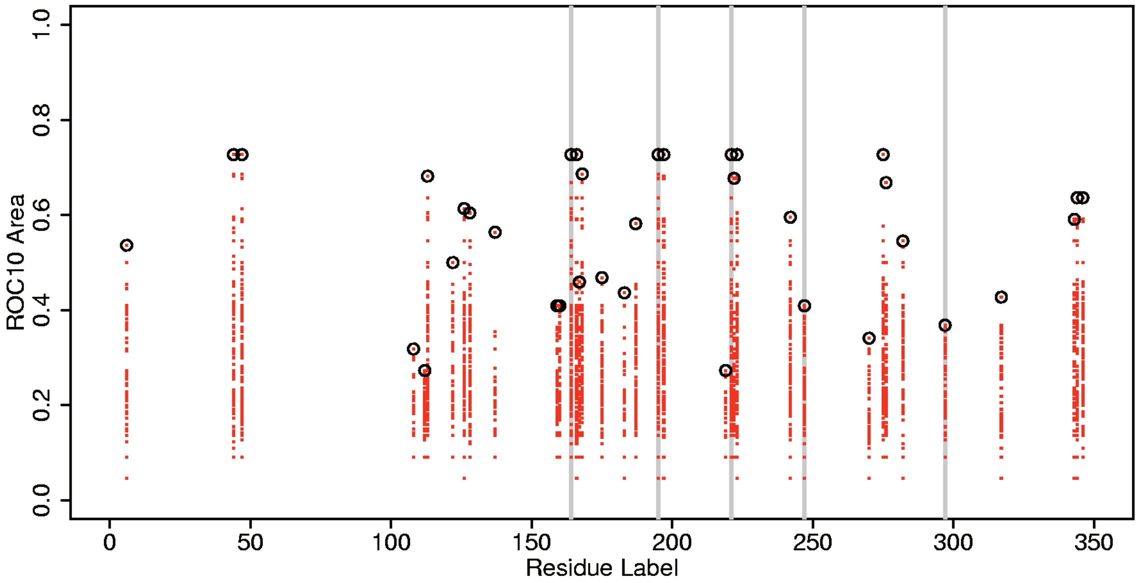


Figure 5. Residues that contribute to motif scores.
For each motif among the 2825 motifs labeled “Conserved” in Figure 1, their ROC10 area score is plotted against each motif residue (red dots). The gray vertical lines show the residues that make up the motif based on functional knowledge, Lys 164, Asp 195, Glu 221, Glu 247 and His 297. The top score for each residue is circled in black.

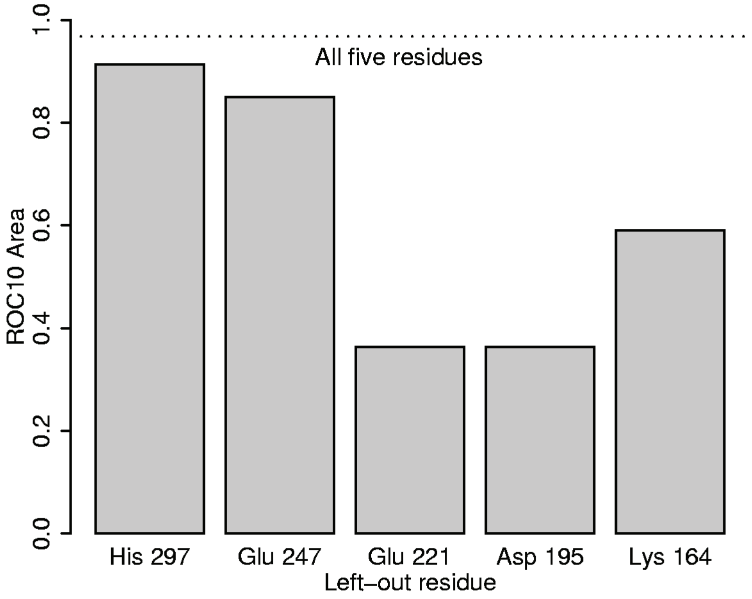


Figure 6. Scores of partial motifs based on the functional site.
Shown are the scores of five motifs made by leaving one residue out of the motif based on functional knowledge. The ROC10 area for the entire five-residue motif is shown by the dotted line labeled “All five residues”.

Discussion

The original purpose of this work was to show that expert knowledge of functionally important residues generated unique and concise 3D motifs in enzyme superfamilies. I have shown that very few randomly generated motifs approach the ability of active site motifs to uniquely identify a whole superfamily. Even when motifs are chosen based on constraints to make them more like active site motifs, randomly generated motifs only approach the ability of the expert motif when they contain most of the residues in an expert motif. It should be pointed out that these results are from a very limited data set: all analysis is based on motifs from just one structure in one group of proteins. While preliminary analysis of another structure (1ebh) in the enolase superfamily revealed similar trends (data not shown here), other proteins in other groups could potentially generate very different conclusions.

A major factor explaining the inability of any of the random motifs to match the effectiveness of the expert motif is the allowed substitutions at each position. While I tried different multiple sequence alignments for choosing allowed substitutions, I could not recreate the list of allowed substitutions that can be generated by detailed manual analysis. Part of the problem is inherent to residue-based motifs. Because multiple amino acids can provide the same chemical capabilities, and a single amino acid can provide multiple different chemical capabilities, residue based motifs are a simplified model of functional protein elements. On the other hand, the residues are the ‘atoms’ of protein evolution: individual changes are at the level of individual residues, not chemical groups. Nevertheless, the requirements of function often allow residues to be replaced based on the chemical requirements. For example, when a glutamate is required for its carboxylate group, often an aspartate can serve as well in the same location. In this simple case, the use of substituting residue types can adequately describe the system, but there are cases that are more complex. For example, two residues in one structure might interact to perform the role of a single residue in a different structure, as a His-Asp dyad serves the role of a lysine in some members of the enolase superfamily (Babbitt et al. 1996). Active site descriptions that use chemical groups or physical and chemical descriptions could avoid this issue.

These results suggest that an automated method to choose a motif could easily be developed based on optimization of random guesses. It is promising that motifs that contain only a fraction of the most important residues, together with other less important residues, produce an intermediate score (see Figures Figure 5 and Figure 6). This could allow an optimization to incrementally learn the best motif through small changes. In my later work, I chose to optimize these motifs through a genetic algorithm (described in Chapter 2), and these results helped guide the development of that genetic algorithm. As a genetic algorithm relies on random guesses, making better random guesses could help reach a solution faster. It is important to recognize a balance though between applying constraints that provide useful limits as opposed to restrictive limits. As functional residues or simply residues that are useful classifiers should always be less variable than residues under no selective pressure, requiring a minimal level of conservation for motif residues was a useful limit. Requiring close spatial proximity of residues, on the other hand, could be an overly restrictive limit, even though it provides for a higher rate of high scoring guesses. While I see the best enrichment in high scoring guesses by restricting motif residues to a 7.5 Å neighborhood, this would eliminate the best observed motifs in some cases. For my genetic algorithm, I only restrict initial guesses to a 12 Å neighborhood and allow the optimizations to ignore residue distances. Likewise, for the final version of my genetic algorithm the restriction by residue type to polar residues was not used.

References

Altschul, S. F., T. L. Madden, et al. (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res **25**(17): 3389-402.

Artymiuk, P. J., A. R. Poirrette, et al. (1994). "A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures." J Mol Biol **243**(2): 327-44.

Babbitt, P. C. (2003). "Definitions of enzyme function for the structural genomics era." Curr Opin Chem Biol **7**(2): 230-7.

Babbitt, P. C., M. S. Hasson, et al. (1996). "The enolase superfamily: a general strategy for enzyme-catalyzed abstraction of the alpha-protons of carboxylic acids." Biochemistry **35**(51): 16489-501.

Berman, H. M., J. Westbrook, et al. (2000). "The Protein Data Bank." Nucleic Acids Res **28**(1): 235-42.

Fetrow, J. S. and J. Skolnick (1998). "Method for prediction of protein function from sequence using the sequence-to-structure-to-function paradigm with application to glutaredoxins/thioredoxins and T1 ribonucleases." J Mol Biol **281**(5): 949-68.

Gerlt, J. A. and P. C. Babbitt (2001). "Divergent evolution of enzymatic function: mechanistically diverse superfamilies and functionally distinct suprafamilies." Annu Rev Biochem **70**: 209-46.

Holm, L. and C. Sander (1998). "Removing near-neighbour redundancy from large protein sequence collections." Bioinformatics **14**(5): 423-9.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Mol Biol **285**(4): 1887-97.

Lichtarge, O., H. R. Bourne, et al. (1996). "An evolutionary trace method defines binding surfaces common to protein families." J Mol Biol **257**(2): 342-58.

Meng, E. C., B. J. Polacco, et al. (2004). "Superfamily active site templates." Proteins **55**(4): 962-76.

Ponomarenko, J. V., P. E. Bourne, et al. (2005). "Assigning new GO annotations to protein data bank sequences by combining structure and sequence homology." Proteins **58**(4): 855-65.

Russell, R. B. (1998). "Detection of protein three-dimensional side-chain patterns: new examples of convergent evolution." J Mol Biol **279**(5): 1211-27.

Valdar, W. S. (2002). "Scoring residue conservation." Proteins **48**(2): 227-41.

Wallace, A. C., N. Borkakoti, et al. (1997). "TESS: a geometric hashing algorithm for deriving 3D coordinate templates for searching structural databases. Application to enzyme active sites." Protein Sci **6**(11): 2308-23.

Introduction to Chapter 2

The work described in Chapter 2 is a natural progression on the work of Chapter 1. With the knowledge of how to make the best randomly derived motifs, I next sought to optimize the best random motifs to build towards motifs as good as or better than expert-built motifs at identifying a group of proteins with similar functions. For the optimization, I chose to use a genetic algorithm. Genetic algorithms discover optimized solutions to problems by randomly modifying and combining the best observed guesses in an iterative process. In my case, solutions are 3D motifs, or simply a collection of residue coordinates chosen from a protein structure. They can be modified by adding or removing residues, and recombined by taking a random subset of the combination of two motifs. It was a straightforward step to take my programming work used in Chapter 1 to generate random guesses, and put it together with these simple modification and recombination routines together with a slightly modified scoring function to generate a simple but effective genetic algorithm. The results, as described in more detail in the body of the chapter, show that while the expert built motifs still showed advantages for certain structures in certain groups, the genetic algorithm can build useful and functionally related motifs that are often as good as the expert built motifs.

The remainder of this chapter represents a verbatim copy of this manuscript, including the supplementary materials, in the journal Bioinformatics (Polacco et al. 2006).

Polacco, B. J. and P. C. Babbitt (2006). "Automated discovery of 3D motifs for protein function annotation." Bioinformatics **22**(6): 723-30.

Chapter 2: Automated Discovery of 3D Motifs for Protein Function Annotation

Abstract

Motivation: Function inference from structure is facilitated by the use of patterns of residues (3D motifs), normally identified by expert knowledge, that correlate with function. As an alternative to often limited expert knowledge, we use machine-learning techniques to identify patterns of 3 to 10 residues that maximize function prediction. This approach allows us to test the assumption that residues that provide function are the most informative for predicting function.

Results: We apply our method, GASPS, to the haloacid dehalogenase, enolase, amidohydrolase and crotonase superfamilies and to the serine proteases. The motifs found by GASPS are as good at function prediction as 3D motifs based on expert knowledge. The GASPS motifs with the greatest ability to predict protein function consist mainly of known functional residues. However, several residues with no known functional role are equally predictive. For four groups, we show that the predictive power of our 3D motifs is comparable to or better than that of approaches that use the entire fold (CE) or sequence profiles (PSI-BLAST).

Introduction

The increasing availability of structural data for proteins of unknown function creates a demand for *in silico* methods to infer the function of these proteins using structural information (Teichmann et al. 2001). But while comparison of overall structures can extend homology detection to evolutionary distances where sequence similarity is undetectable (Chothia et al. 1986), fold comparison often does not identify similarities among functionally significant residues or atoms involved in a protein function's mechanism. Together, the coordinates of these residues or atoms can define a 3D motif. There are many available motif-matching methods that can be used to identify a protein with a matching motif and thus a similar function and mechanism (for example, Artymiuk et al. 1994; Fetrow et al. 1998; Barker et al. 2003). Such methods offer useful complements to fold-based homology comparisons, especially in cases where homologs have diverged in function.

In earlier studies, 3D motifs have typically been chosen based on expert knowledge of functionally important residues in enzyme active sites such as the catalytic triad of the serine proteases. These motifs have been successful at identifying specific enzymatic activities (Torrance et al. 2005), binding relationships (Artymiuk et al. 1994), and superfamily membership (Meng et al. 2004). However, in the absence of a large data source of functional information, accumulation of motifs is slow. The catalytic site atlas (CSA) is a new effort to create a comprehensive database of functional information gleaned from the literature (Porter et al. 2004). It currently provides 147 non-redundant active site motifs for enzymes (Torrance et al. 2005). Similarly, Arakaki et al. (2004) presented an automated method that used the functional information in feature records of the Swiss-Prot database to construct 3D motifs for 162 different enzymes. Even this method is limited by the shortage of functional information in Swiss-Prot. There are numerous other examples of computational approaches to predict functionally important residues (for example, Zvelebil et al. 1988; Elcock 2001; Wangikar et al. 2003), but these may not be accurate enough to translate to useful motifs (see Discussion).

An alternative is the use of automated 3D motif detection methods. These have shown some success, though none has mapped motifs to specific protein functions with the design goal of characterizing novel proteins with high accuracy. PINTS detects repeated patterns of sidechains between pairwise comparisons of diverse structures, and has generated a large set of repeated motifs (Russell 1998). A similar data-mining approach that compares all patterns across an entire library of structures finds the catalytic triads of proteases along with metal binding sites, salt bridges and similar structural features (Oldfield 2002). Although such general structural features do not provide much specific functional information, they dominate the databases of motifs generated by these types of methods.

We present here a new approach for automated 3D motif generation named GASPS (Genetic Algorithm Search for Patterns in Structures). GASPS was developed with two basic design goals. First, for any specified group of proteins, GASPS should find the motif most useful for identifying the group. Second, GASPS should rely as little as possible on knowledge about what is likely a predictive or functionally important residue. We validate the effectiveness of GASPS on four highly divergent groups of enzymes: the convergent serine proteases (SP), the amidohydrolase superfamily (AHS), the enolase superfamily (ES), and the haloacid dehalogenase superfamily (HADS). These motifs verify that many, but not all of the previously known functionally important residues are the best predictive residues (along with additional unexpected residues). We describe the crotonase superfamily (CS) as an example of a group that is not well suited for characterization by 3D motifs as they are typically defined.

Methods

Motif Representation and Matching

As an initial test of principle, we adopted the motif model and matching algorithm of SPASM (Kleywegt 1999), although GASPS can be adapted for use with other motif matching algorithms as well. A motif is a small set of residues (<10 for this study) taken from a single chain, here called the query chain. For each position, SPASM requires a matching residue to be of the identical type with no substitutions. Alternatively, a unique set of residues at each position may be specified that can be substituted with no penalty, though in the course of our study we were unable to use this feature effectively (see Supplementary Materials). SPASM models each residue with just two points, backbone Ca and the sidechain geometrical center. SPASM computes a superposition root-mean-squared deviation (RMSD) for each match it finds within user-defined thresholds of RMSD, sidechain distance deviations (SCD), and Cα distance deviations (CαD). For this study, thresholds were set to RMSD=3.2Å, SCD=3.8Å, and CαD=5.0Å. SPASM allows the use of several additional constraints that were not used for this study. Only the match with the best RMSD is considered from each structure.

GASPS

GASPS generates motifs by selecting residues from a single query chain. Here, functional sites and motifs that span more than one chain are not directly addressed. These motifs are scored for their ability to accurately discriminate the positive from the negative sets. There are four main components to a GASPS run: query processing, initial guesses, scoring, and refined guesses.

Query Processing To limit the search space, only the 100 most conserved residues in the query chain are considered for inclusion in a motif. Conservation is calculated from a multiple sequence alignment by weighting sequences to reduce the effects of redundancy, considering conservative substitutions based on a substitution matrix, and to penalize gaps (Valdar 2002). All multiple sequence alignments were generated by a two-iteration PSI-BLAST (Altschul et al. 1997) search against nrdb90 (Holm et al. 1997) built in February, 2004.

Initial Guesses Fifty candidate motifs are initially chosen spread equally across the linear sequence of the query chain to provide coverage of all regions. For each random guess, a first residue is selected from the query chain and then four other residues are randomly chosen such that each α-carbon is within 12Å of the first α-carbon.

Scoring Function Candidate motifs are scored for their ability to discriminate between the positive and negative proteins based on the best RMSD matches from a SPASM search. The query structure, which is always a perfect match to the motif, is excluded from the positive set. The scoring function is primarily the normalized area under a receiver-operator characteristic (ROC) plot to five false positives (a false positive rate of ~0.001). If the sorted RMSD scores for structures in the negative set are (f_1, f_2, f_3,…f_n), then this area, called R, can be computed explicitly as:

$$R = \frac{1}{5}\sum_{i=1}^{5}\frac{T(f_i)}{T_{\max}}$$

where T(f) is the number of true positives with a better RMSD match than a given false positive and T_{max} is the size of the positive set. R ranges from 0 to 1. Because R is based on discrete counts, different motifs will frequently have identical R scores. To avoid ties, we include an additional term in the GASPS scoring function. This term, S, is the normalized difference in median RMSD between the true positives and false positives, only considering those that score better than the fifth false positive (f5). This can be explicitly defined as:

$$S = \frac{median(f_{1-5}) - median(t_{1-m})}{median(f_{1-5})}$$

where $t_{1\text{-}m}$ is the set of RMSDs from the true positive matches that are better (less) than the fifth false positive (f5). When no true positives are hit (R=0), S is set to zero. The overall GASPS score (G) is then the sum of S and R weighted to emphasize the ROC score, and is composed:

$$G = 1.0R + 0.1S$$

Refined Guesses The 16 highest scoring motifs of any round are included in the next round and used as parents for constructing 36 novel motifs via one random process: deletion, insertion, mutation or recombination. The only restriction on the new motifs is that they contain at least 3 residues and at most 10. Deletions and insertions generate a new motif by removing or adding a residue to a single parent motif. A mutation is a combination of a deletion and an insertion. A recombination is a random subset of the combination of two parent motifs. The top-scoring motif after fifty rounds of refinement is considered the final winner. Most GASPS runs in this study took between 12 and 18 hours on a single 2.667 GHz Intel Xeon processor. Most of this time was spent completing the SPASM searches against the negative set, which time scales directly with the number of proteins in the negative set.

Structure Library

Most analyses in this study used a set of structures selected from the Protein Data Bank (PDB) (Berman et al. 2000) to represent all families in The Structural Classification of Proteins (SCOP) version 1.65 (Murzin et al. 1995). The selection algorithm treats each SCOP family individually and has three main goals: 1) mutant removal based on text matching PDB fields, 2) sequence redundancy filter to 40% identity, and 3) favoring the highest quality structures based on resolution. No distinction is made between apo and holo structures. The entire corresponding PDB chains for each of the SCOP domains are included, regardless of similarities at other domains. On SCOP version 1.65, this selection results in 5440 unique domains on 4243 unique chains.

Positive and Negative Sets

We chose five well-characterized positive groups so that all members within each group share a similar function, and this shared function is dependent on known functional residues (Table 1). Definitions for the four superfamilies were taken from the Structure-Function Linkage Database (SFLD) (Pegg et al. 2005). However, the SFLD as yet contains only a few superfamilies, so to mimic a more typical usage of GASPS on less than perfect classifications, for all five groups of proteins studied here, a positive set of structures was selected based on SCOP superfamily and family classifications (given in Table 1). Each positive set is a subset of the structure library with the modification that all chains within a PDB structure file are included. Sequence identities between all pairs of homologous chains used as query chains range from 14% to 40%. The negative set is the entire structure library, excluding all chains that contain at least one domain meeting the criteria for inclusion in the positive set.

Cross-Validation

Complete rounds of leave-one-out cross-validation were performed for several query structures in each group. For the smaller groups, each structure in the positive set was used once as a query structure. For the larger groups, AHS and SP, a randomly selected subset of the structures was used. For each query structure, all possible positive training sets were produced by excluding one other (non-query) positive structure. The corresponding positive test sets each contained just the excluded structure. Similarly, the negative set was equally divided to produce as many negative test sets as positive test sets. The corresponding negative training sets are simply the entire negative set excluding a negative test set. Using ES as an example, this procedure required 42 runs of GASPS (7 query structures multiplied by 6 left-out positive structures). The reported sensitivity on the test sets is the portion of GASPS runs where the final GASPS motif from each training run was able to discriminate the left-out positive member from the left-out negative test set at an RMSD threshold equal to the RMSD of the fifth-best false positive match on the training sets. Those runs for which the final trained GASPS motif did not score significantly on the training set were excluded.

Table 1. Functionally Similar Protein Groups

Group	SCOP groups	N[1]	Known Functional Residues
Amidohydrolase Superfamily	c.1.9	16	(1a4m) H15 H17 H214 H238 D295
Enolase Superfamily	c.1.11	7	(2mnr) K160 D191 E219 D244 K268
Crotonase Superfamily	c.14.1.3	7	(1mj3) backbone A98 G141
Haloacid Dehalogenase Superfamily	c.108.1	12	(1fez) D12 T126 R160 D186 D190
Serine Proteases	b.47.1.1, b.47.1.2, b.47.1.3, c.41.1.1	38	(2hlc) H57 D102 S195

1) Number of non-redundant structures in positive set.

PSI-BLAST and CE Libraries

For BLAST (Altschul et al. 1997) and PSI-BLAST comparisons with GASPS the libraries were the set of unique chains from the same PDB files used in the positive and negative sets for GASPS (described above). For the Combinatorial-Extension algorithm (CE) (Shindyalov et al. 1998), to avoid computing all-by-all pairwise comparisons, the negative sets were reduced to the likely high-scoring members for each positive group. For most groups, this meant limiting the negative set to those chains with the same SCOP fold as a catalytic domain in the positive set. However, according to SCOP, HADS is the only superfamily of the HAD-like fold, so its negative set for CE was chosen based on CATH (Orengo et al. 1997) instead. For SP, there were an insufficient number of same-fold structures that were not serine proteases to provide negative sets for both the subtilisins and trypsins. An additional SCOP fold (b.43: Reductase/isomerase/elongation factor common domain) was included in the library that commonly scored highly against SP folds according to the CE internet database (http://cl.sdsc.edu/ce.html).

Results

Validation of GASPS

Significance of Optimized GASPS Scores. To determine whether any GASPS motif likely represents more than a chance co-occurrence of residues, we computed significance cutoffs from empirical distributions of GASPS motifs due to chance alone. To ensure that any motif was due to chance, artificial positive groups were generated by randomly selecting structures from the structure library, each with a different fold. Based on these distributions, provided in Supplementary Materials, we can set GASPS score thresholds for moderate significance ($p < 0.01$): for groups of approximately 5 structures motifs must score greater than 0.55 and in larger groups of 10 or more structures they must score above 0.4.

Cross-Validation Studies. To estimate the performance of GASPS on new proteins, leave-one-out cross-validation studies were completed on each of the groups in Table 1. RMSD thresholds were chosen for each top GASPS motif to give a false positive rate of approximately 0.0013 (5 false positives) on the training set. With the exception of CS, sensitivity is high and there is a close correspondence between the training and test sets (Figure 1). Sensitivity on the test sets for most cases is approximately 90% of that on the training cases. The false positive rate (and its complement, sensitivity) shows an even tighter correspondence with an average rate of 0.0014 on the test cases. The fact that CS is one of the smallest groups and also that it lacks highly conserved sidechains in the active site, as described below, likely contribute to the poor performance of GASPS for this superfamily.

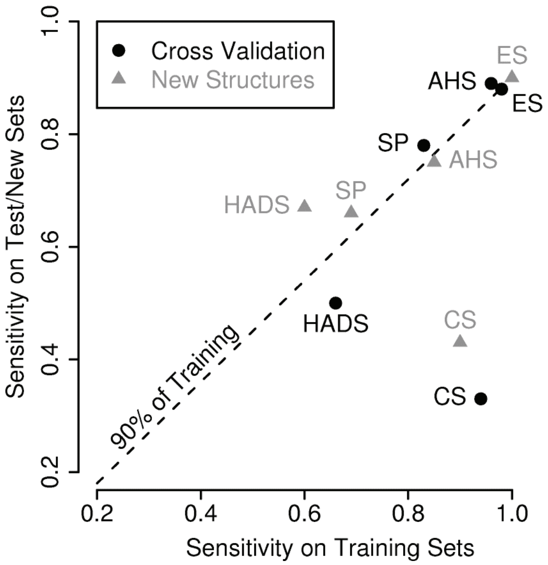


Figure 1. Generality of GASPS motifs based on sensitivity from two experiments: cross-validation and detection of newer structures.
Black filled circles show average sensitivities of motifs from leave-one-out runs on the cross-validation training (x-axis) and test (y-axis) sets. Gray triangles show sensitivities of motifs generated on the full training sets (all motifs in Figure 3) when used to detect structures in the full training set (x-axis) compared with novel structures solved after the training set was established (y-axis).

Detection of New Structures. Across all groups, we identified 12 new structures in the PDB that were not yet classified by SCOP (as of version 1.65, December 2003), by a combination of searches based on literature, annotation and sequence similarity, along with communications with collaborators. These 12 proteins all share less than 40% sequence identity with each other or with any protein in the original training set. Motifs generated on the full training set, one for each query structure (shown in Figure 3), were tested for their ability to match the appropriate new structures within the RMSD thresholds determined on the full training set. For these 12 structures, the group-based average rate of matches is 68% compared with 81% on the structures included in the full training set. If CS is excluded, the group-based average rate of matches is 75%, compared to 79% on the training set (Figure 1). This is an average across all motifs in each group including those with insignificant scores and very poor match rates. The expected match rate for any given motif appears linked to its original GASPS score. Excluding CS, no top-scoring motifs in any one group missed any of the new structures, and only 1 of 9 insignificantly scoring motifs matched any new structures. No new structures, CS included, failed to match any motif in their group.

Comparisons with other 3D motif methods. A key benefit of GASPS is that it requires no knowledge of functionally important residues. However, even on groups where functionally important residues are known, GASPS is still useful if it is able to select a more sensitive motif. We constructed motifs built from the functionally important residues (see Table 1) for all possible query structures and compared their sensitivity to GASPS motifs. For all groups except SP, the GASPS motifs have higher sensitivity than simply using these functional residues (Figure 2, "FUN").

Of other available techniques, the closest to GASPS in principle, is DRESPAT (Wangikar et al. 2003) that detects similar patterns of residues within a group of structures. We used DRESPAT with previously published parameters and a pattern size of four residues to generate patterns for the groups in our data set. The resulting top ranked patterns identify some functionally important residues for all groups in this study except for CS (not shown). However, they fail to identify superfamily members with similar specificity and sensitivities to those of GASPS motifs (Figure 2). It may be possible to adjust the parameters and desired pattern size to improve the performance on a case-by-case basis, but the DRESPAT technique is not designed to automate or aid such a procedure.

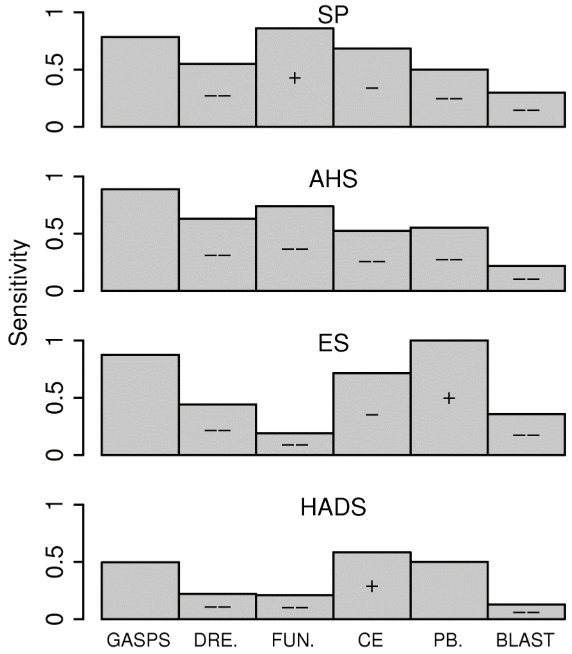


Figure 2. Sensitivity of GASPS motifs compared with other techniques. Sensitivity shown for GASPS is measured by cross validation (Fig. 1). For all other techniques, the sensitivity is measured at the threshold of the fifth false positive. Other techniques are DRESPAT (DRE.) motifs, motifs built from functional residues (FUN.), CE, PSI-BLAST (PB.) and BLAST. Plus signs (+) indicate significantly better sensitivity than GASPS within the protein group, and minus signs indicate significantly worse performance at $p < 0.05$. Double signs (++ or --) indicate a greater degree of statistical significance ($p < 0.0001$). CS results are not shown because no 3D motif methods were able to characterize it effectively

Two other 3D motif libraries have recently been used to identify functional or homologous relationships, the motifs used by PINTS and the CSA. As these libraries were not specifically constructed to identify members of the groups in this study, it is impossible to run a parallel experiment for a direct comparison with the techniques shown in Figure 2. PINTS has been used to confirm superfamily membership and binding relationships of structural genomics proteins by finding matches to motifs derived from proximity to ligands or SITE annotations in PDB records (Stark et al. 2004). We tested this same technique (made available at http://www.russell.embl.de/pints/) by asking whether the structures used in our study matched with high specificity those motifs that came from other non-redundant (<40% sequence identity) group members. The measured sensitivities of GASPS motifs (Figures 1 and 2) greatly outperform PINTS for all five groups at similar or even much lower rates of specificity. To be generous, PINTS could adequately serve its purpose if for any query structure it only detects a single true positive motif and ranks it highest among all matches. Even using this much less stringent definition of sensitivity for PINTS than used for GASPS, only for SP does PINTS score better than our reported sensitivities for GASPS motifs.

The motifs derived from functional knowledge in the CSA are available for searching by the program Jess (Barker et al. 2003) at their website (http://www.ebi.ac.uk/thornton-srv/databases/CSA/). We used each of the structures in our positive sets to search the CSA with Jess and scored true positives according to whether the motif originated from the same group (defined in Table 1) as the original query. Maintaining similar specificity as required for GASPS, we should require that JESS, with only 147 motifs, identify true positives with greater E-value than any false positive. Only structures from SP reliably matched any true positives. Outside of SP, only three structures (one each from AHS, ES and HADS) matched any CSA motif, but all three of these motifs came from structures that shared more than 40% sequence identity with the query. Relaxing the specificity to five false positives only allowed two other HADS structures to match the haloacid dehalogenase motif. Even though several of the false positive matches had E-values that suggested significance (~10-4), none correctly predicted the function or group membership of the original query. These high quality motifs in the CSA are useful for detecting certain specific functions, but they cannot adequately detect the diverse functions or distant relationships covered by the superfamilies studied here.

Prediction Ability Compared to Whole-Chain Tools. We compared the sensitivity of GASPS motifs (as estimated by cross validation) with other tools that use the whole protein chain including the sequence-based tools BLAST and PSI-BLAST (Altschul et al. 1997) and the fold comparison tool CE (Shindyalov et al. 1998). All members of all positive groups were used as queries for each of the methods, and these were searched against the appropriate library as described in Methods. All sensitivities were measured by counting the fraction of positives that scored better than the fifth best-scoring negative for each query (Figure 2). No single method is better than all other methods for all of the groups in this study. CS is easily grouped by most methods with the exception of GASPS motifs. The fold comparison tool CE performs well on groups with unique folds such as HADS. AHS and ES, on the other hand, share the common $(\beta/\alpha)_8$ fold with many other superfamilies, which may help explain why CE performs worse than GASPS in these cases. PSI-BLAST performs better than GASPS only for the least divergent superfamilies considered, ES and CS, where PSI-BLAST performs perfectly. With the exception of CS, GASPS motifs outperform BLAST on all groups.

Detection of Key Functional Residues

An advantage of our method is that the selection of the residues in a motif is unbiased towards any preconceived notions of functionally important residues except indirectly via our exclusion of the least conserved residues. This allows us to ask if there is a relationship between the residues that discriminate proteins of related function and the residues that we know from experimental studies provide function. Table 1 shows the residues that are known to be directly involved in shared function or used in previous functional motif studies. These are used as our gold standard of key functional residues. Every positive structure was used once as a query structure except for SP from which only a diverse subset of structures was chosen. The best motifs from each of these runs are presented in detail in Figure 3. There is a clear trend for the proportion of functional residues in a motif to increase as the motif score rises.

As a stochastic search method, GASPS can be expected to produce different motifs in identically configured runs, and we expect several of the lower scoring motifs presented in Figure 3 are not the best motif a given query can provide. The results of repeated runs for several configurations are presented in Supplementary Materials. Clearly, multiple GASPS runs per group are necessary to ensure that an optimal motif is found for any group. For some single query structures, however, repeated runs suggest there is no combination of residues that provide a useful motif. Meanwhile, the optimal motifs for the majority of other query structures are highly similar. Taken together, these results suggest that to generate a set of the most useful and inclusive motifs for any group of proteins, limited resources are better spent on running GASPS on many different query structures than on running GASPS multiple times on the same structure.

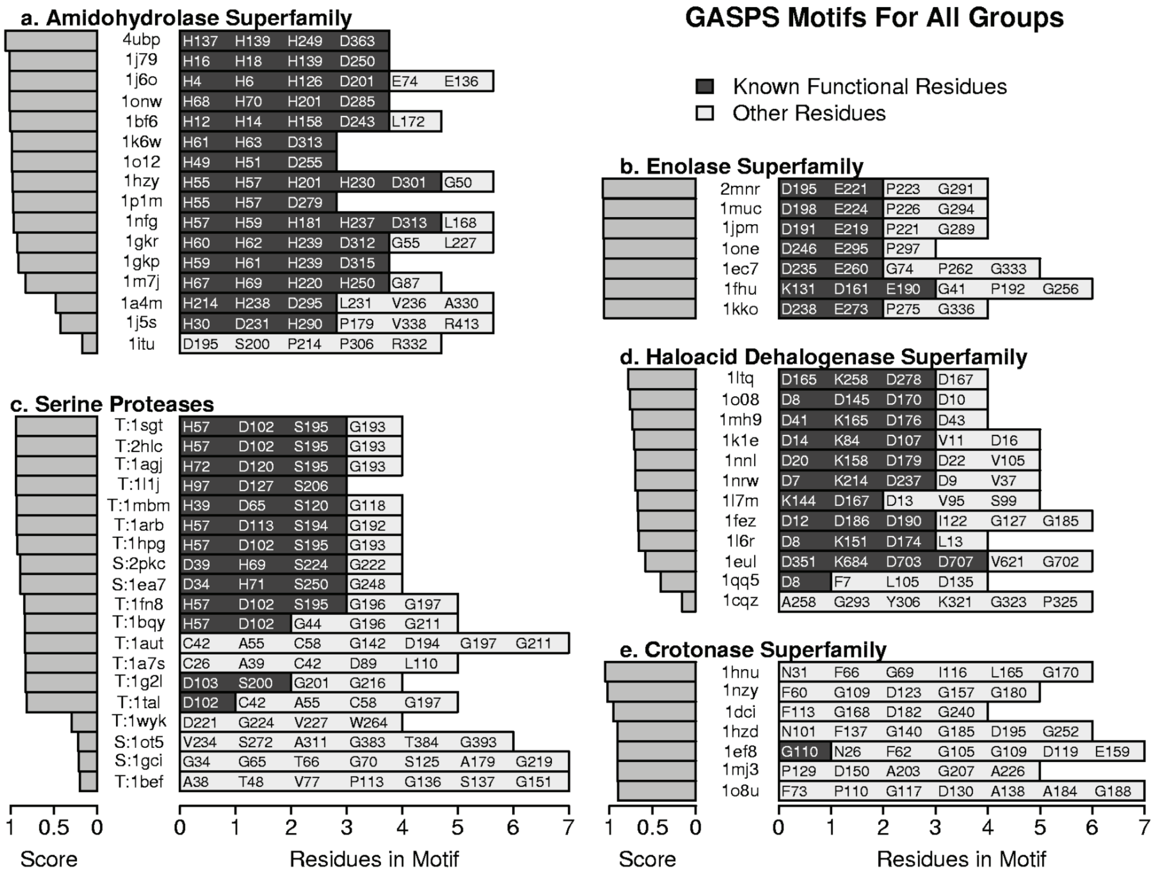


Figure 3. Scores and functional significance of GASPS motifs.
The results of a single GASPS run are presented for each named query structure. Residues in the motif that correspond to previously identified functional residues or known active-site motif residues are darkly shaded. All other residues are lightly shaded regardless of subsequent determination of their functional significance. For the serine proteases, query structures are labeled “T:” to denote trypsin-like folds or “S:” for subtilisin-like folds.

Amidohydrolase Superfamily The amidohydrolase superfamily (AHS) is a functionally diverse superfamily composed of homologs with a $(\beta/\alpha)_8$ (TIM) barrel fold that share a conserved mechanistic step mediated by a conserved set of active site residues (Holm et al. 1997; Gerlt et al. 2003). All known members of the superfamily are metal-dependent and require either one or two divalent metal ions. Five conserved metal ligands comprising four histidines and an aspartic acid have been identified as functionally important in all groups within the superfamily. Only one GASPS run on this superfamily resulted in a motif with an insignificant score and no overlap with any of these metal ligands (Figure 3a). The remaining runs all resulted in motifs that contained at least three of the five conserved ligands. The other residues in the significant motifs are all distant from the metal ligands and thus probably not directly involved in the enzyme's active site.

Enolase Superfamily Like the amidohydrolase superfamily, the enolase superfamily (ES) is made up of homologs with a C-terminal $(\beta/\alpha)_8$ barrel fold plus an N-terminal domain representing a unique fold. All functionally diverse members share a common mechanistic step (Babbitt et al. 1996; Gerlt et al. 2005). Past studies have carefully documented conserved elements responsible for the shared aspects of mechanism, and motifs based on this functional information have been generated with success (Meng et al. 2004). In this study, the conserved residues considered to play a functional role consist of three divalent metal ligands and two basic residues. All motifs resulting from GASPS runs contained at least the same two metal ligands, and one run contained one of the basic residues (Figure 3b). The remaining metal ligand and both basic residues are known to have variable residue types across members of the superfamily, possibly explaining why GASPS has trouble locating them. A highly conserved residue among the GASPS motifs that has not been identified as functionally important is a proline that is two positions downstream from the second metal ligand. Here called the "downstream proline", it appears in all ES motifs.

Haloacid Dehalogenase Superfamily The haloacid dehalogenase superfamily (HADS) comprises enzymes with diverse functions, yet all members share a common mechanistic step associated with hydrolytic nucleophilic substitution via a conserved aspartate and a few other residues (Allen et al. 2004). The HADS fold is unique according to SCOP, though CATH divides it into two domains: a common Rossman fold domain and a domain unique to the superfamily. Our laboratory has previously developed motifs in a manual process based on expert knowledge (Meng et al. 2004), and the residues in these motifs are here considered the conserved functional residues. While the catalytic roles may be conserved at each of these positions, all but the obligate aspartate are substituted in diverse members of the superfamily, as apparently required to accommodate differences in their specific mechanisms and overall functions. Despite these substitutions, most GASPS runs still contain three of the five functional residues (Figure 3d). The nucleophilic aspartate appears in all significant motifs where possible. (The 1l7m structure contains two alternate conformations listed for this aspartate, D11, which precluded it from inclusion in a motif.) Nearly as frequent as the nucleophilic aspartate is another aspartate two positions downstream that has been implicated by others in binding and protonation of the substrate (Allen et al. 2004).

Serine Protease Families The serine proteases (SP) are a polyphyletic group consisting mainly of two non-homologous families: the subtilisins and trypsins. They are grouped together by virtue of their common functions and use of a structurally similar catalytic triad in their active sites that appear to be the result of convergent evolution (Dodson et al. 1998). Slightly more than half of the motifs and the highest scoring (10 of 19) included the entire triad (Figure 3c). Most triad-containing motifs included only one additional residue: a glycine involved in formation of the conserved oxyanion hole (for example, 2hlc G193) in trypsins. Though this glycine matches a conserved glycine in the subtilisins, the NH group in the subtilisins points away from the active site cavity. Of the nine remaining motifs, four had insignificant scores, three included partial catalytic triads, and one was built from a heparin binding protein (1a7s) that, despite its homology to the trypsins, does not contain the catalytic triad or perform protease activity. Many significant runs seemed to be distracted by a disulfide bridge and neighboring alanine near the active site (C42-C58, A55, see Figure 3c and Supplementary Materials), which are well conserved among the trypsins but not the subtilisins.

Crotonase Superfamily Members of the crotonase superfamily (CS) display great catalytic diversity, yet all are unified by a common structure-based stabilization of an enolate anion intermediate of acyl-CoA substrates (Holden et al. 2001). Unlike the other groups given in Table 1, however, this shared chemistry is not performed by sidechains but by two structurally conserved NH groups of the peptide backbone that function as part of an oxyanion hole. The sidechains of these residues are not strictly conserved across the superfamily nor are there any other sidechains known or predicted to act in catalysis that are conserved across the entire superfamily. The crotonase superfamily therefore provides a test of GASPS and sidechain-based motifs on a group that may not contain a structural motif focused on sidechains. As expected, an insignificant number of residues in the motifs (1 of 33, for all motifs) is involved in the formation of the characteristic oxyanion hole (Figure 3e). The common residues in the motifs that do discriminate this superfamily seem unlikely to play a direct role in the enzyme’s function, based on their distance from the active site. Examples include a conserved phenylalanine (1hnu F66) that is buried but lines an interior cavity and an aspartate (1hnu D135) involved in a conserved salt bridge.

Discussion

Using GASPS for Function Identification

The performance of GASPS-generated motifs is comparable to that of 3D motifs generated based on expert knowledge of functional sites in other proteins (Artymiuk et al. 1994; Wallace et al. 1997; Fetrow et al. 1998; Kleywegt 1999; Torrance et al. 2005). Furthermore, GASPS motifs improve the coverage of protein functions offered by publicly available sources of 3D motifs (Stark et al. 2003; Porter et al. 2004). Searching with protein fragments in three-dimensional motifs developed by GASPS was also found to be comparable or better than commonly used methods of annotation transfer that use an entire protein chain such as PSI-BLAST or CE. Unlike these methods that use an entire protein or domain, GASPS is able to focus on the features of protein structure most likely to tell us the most about protein function. GASPS therefore provides a method of generating motifs useful for function or superfamily prediction in an automated fashion with no prior knowledge of mechanistic details. Such motifs can be used to verify similarity of active sites in proteins in which only similarity of fold has been previously identified. For example, GASPS motifs could be used for distinguishing functional differences among families of $(\beta/\alpha)_8$ proteins.

GASPS requires only a prior grouping of related proteins, so GASPS is limited only to groups with sufficient available structures. We cannot say for certain how many structures are required, but it appears to depend on the variability among the available structures. In the current study, all structures shared less than 40% sequence identity, and GASPS still was able to find general motifs for groups with as few as seven structures.

While only 14% of superfamilies and 6% of families in the structure library used here have this many structures, these superfamilies and families make up the majority of protein structures (60% and 32%, respectively). Theoretically, it would seem possible for two highly diverged structures to share only their unique functional motif. However, for most proteins, even of different folds, it appears that sharing similar residues in 3D space occurs frequently enough by chance alone to require more than two structures to produce a trusted motif (Wangikar et al. 2003).

SPASM (and therefore GASPS, as used in this study) considers only a single point for each Ca and sidechain. With most catalysis carried out by sidechains (Bartlett et al. 2002), we believe the inclusion of the sidechains allows for better characterization of functional sites than if only the backbone placement were considered. Motifs could be represented with more precision by using the location of chemical groups, or even individual functional atoms. However, given the variability in placements of functional atoms in crystallographic structures, (DePristo et al. 2004; Torrance et al. 2005), approximating the entire sidechain by a single rigid point may be more appropriate.

Location of Functional Information

GASPS makes no assumptions about the location of functional information except that it can be resolved to individual residues and that it will be relatively well conserved. The observed correspondence between information useful for classification and functionally significant residues is a result of the choice of positive sets based on shared chemical activities used in this study. The use of GASPS on sets based on other shared characteristics, such as homology, binding partners, or cofactors, may identify the residues most attributable to those shared characteristics.

It should be noted that the motifs generated by GASPS may not be the only, or even the most informative structural features. GASPS is expected to miss informative structural features if the features are either inconsistent between members of the group, such as the substituted residues in HADS, or if the features are not based on individual sidechains such as backbone interactions or helix dipoles. The CS results provide a case in point.

In addition to the previously identified functionally important positions, other positions occur with high frequency among the motifs for these groups. These positions may, for example, merely provide a simple geometric positioning constraint for the other motif residues that aids specificity. However, based on their conservation in 3D space, these positions are likely to play an important role for the protein, especially when located in the active site. For example, when the conserved "downstream proline" in the enolase superfamily is mutated to alanine in the muconate lactonizing enzyme from *Pseudomonas putida* (equivalent to structure 1muc) it results in an insoluble protein, (R. Nagatani and P. Babbitt, personal communication,) suggesting that this proline may be important for folding or stability of the soluble globular protein.

Based on its ability to identify at least a subset of the functionally important residues, GASPS appears similar to the fully automated DRESPAT, which successfully locates functionally important residues by identifying shared structural patterns in a set of functionally similar protein structures (Wangikar et al. 2003). The main differences between GASPS and DRESPAT are that GASPS compares patterns with a negative set and chooses patterns based on their predictive ability. Wangikar et al.(2003) suggest that DRESPAT patterns may represent useful 3D motifs. However, in the course of this study we found that when DRESPAT patterns were converted to motifs for use by the search tool SPASM, they were not as accurate as those motifs generated by GASPS.

Inference of Function for Diverse Groups

Four of the five groups in this study have been described as “mechanistically diverse superfamilies” (Gerlt et al. 2001) consisting of divergent enzymes that perform many different overall biochemical functions, but utilize a common mechanistic step such as a partial reaction. Any motifs that identify proteins to these groups will therefore identify the shared mechanistic step but not the overall biochemical function. By mapping a specific mechanistic step to specific structural elements, we are using a finer-resolution view of protein function than overall biochemical function, but one that is more appropriate for such diverse groups (Babbitt 2003).

Future Applications

If applied to an exhaustive functional classification of proteins, GASPS has the potential to generate an unbiased set of 3D motifs that can aid in function prediction for novel proteins. In addition to aiding protein classification, the collection of 3D motifs can represent hypotheses about determinants of function shared among related proteins. In this regard, the high-scoring motifs can serve as starting points for studies attempting to link function to structure, especially in a superfamily context. Additionally such a study would systematically investigate the utility of 3D motifs at identification of functions other than catalysis, such as ligand binding.

For groups with few experimental structures available, especially those coming from structural genomics initiatives, GASPS would have insufficient structures without the use of predicted structures, generated by homology modeling, for example. Past work has specifically demonstrated the effectiveness of predicted structures for matching previously determined functional motifs (Arakaki et al. 2004). It remains unclear whether predicted structures can be used reliably for generating motifs. Work is ongoing in our laboratory to investigate this issue.

Acknowledgements

We thank Elaine Meng for her useful discussions and careful reading of an earlier version of the manuscript. This work was supported by NSF grant DBI-0234768.

References

Allen, K. N. and D. Dunaway-Mariano (2004). "Phosphoryl group transfer: evolution of a catalytic scaffold." Trends Biochem Sci **29**(9): 495-503.

Altschul, S. F., T. L. Madden, et al. (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res **25**(17): 3389-402.

Arakaki, A. K., Y. Zhang, et al. (2004). "Large-scale assessment of the utility of low-resolution protein structures for biochemical function assignment." Bioinformatics **20**(7): 1087-96.

Artymiuk, P. J., A. R. Poirrette, et al. (1994). "A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures." J Mol Biol **243**(2): 327-44.

Babbitt, P. C. (2003). "Definitions of enzyme function for the structural genomics era." Curr Opin Chem Biol **7**(2): 230-7.

Babbitt, P. C., M. S. Hasson, et al. (1996). "The enolase superfamily: a general strategy for enzyme-catalyzed abstraction of the alpha-protons of carboxylic acids." Biochemistry **35**(51): 16489-501.

Barker, J. A. and J. M. Thornton (2003). "An algorithm for constraint-based structural template matching: application to 3D templates with statistical analysis." Bioinformatics **19**(13): 1644-9.

Bartlett, G. J., C. T. Porter, et al. (2002). "Analysis of catalytic residues in enzyme active sites." J Mol Biol **324**(1): 105-21.

Berman, H. M., J. Westbrook, et al. (2000). "The Protein Data Bank." Nucleic Acids Res **28**(1): 235-42.

Chothia, C. and A. M. Lesk (1986). "The relation between the divergence of sequence and structure in proteins." Embo J **5**(4): 823-6.

DePristo, M. A., P. I. de Bakker, et al. (2004). "Heterogeneity and inaccuracy in protein structures solved by x-ray crystallography." Structure (Camb) **12**(5): 831-8.

Dodson, G. and A. Wlodawer (1998). "Catalytic triads and their relatives." Trends Biochem Sci **23**(9): 347-52.

Elcock, A. H. (2001). "Prediction of functionally important residues based solely on the computed energetics of protein structure." J Mol Biol **312**(4): 885-96.

Fetrow, J. S. and J. Skolnick (1998). "Method for prediction of protein function from sequence using the sequence-to-structure-to-function paradigm with application to glutaredoxins/thioredoxins and T1 ribonucleases." J Mol Biol **281**(5): 949-68.

Gerlt, J. A. and P. C. Babbitt (2001). "Divergent evolution of enzymatic function: mechanistically diverse superfamilies and functionally distinct suprafamilies." Annu Rev Biochem **70**: 209-46.

Gerlt, J. A., P. C. Babbitt, et al. (2005). "Divergent evolution in the enolase superfamily: the interplay of mechanism and specificity." Arch Biochem Biophys **433**(1): 59-70.

Gerlt, J. A. and F. M. Raushel (2003). "Evolution of function in (beta/alpha)8-barrel enzymes." Curr Opin Chem Biol **7**(2): 252-64.

Holden, H. M., M. M. Benning, et al. (2001). "The crotonase superfamily: divergently related enzymes that catalyze different reactions involving acyl coenzyme a thioesters." Acc Chem Res **34**(2): 145-57.

Holm, L. and C. Sander (1997). "An evolutionary treasure: unification of a broad set of amidohydrolases related to urease." Proteins **28**(1): 72-82.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Mol Biol **285**(4): 1887-97.

Meng, E. C., B. J. Polacco, et al. (2004). "Superfamily active site templates." Proteins **55**(4): 962-76.

Murzin, A. G., S. E. Brenner, et al. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures." J Mol Biol **247**(4): 536-40.

Oldfield, T. J. (2002). "Data mining the protein data bank: residue interactions." Proteins **49**(4): 510-28.

Orengo, C. A., A. D. Michie, et al. (1997). "CATH--a hierarchic classification of protein domain structures." Structure **5**(8): 1093-108.

Pegg, S. C., S. Brown, et al. (2005). "Representing structure-function relationships in mechanistically diverse enzyme superfamilies." Pac Symp Biocomput: 358-69.

Porter, C. T., G. J. Bartlett, et al. (2004). "The Catalytic Site Atlas: a resource of catalytic sites and residues identified in enzymes using structural data." Nucleic Acids Res **32 Database issue**: D129-33.

Russell, R. B. (1998). "Detection of protein three-dimensional side-chain patterns: new examples of convergent evolution." J Mol Biol **279**(5): 1211-27.

Shindyalov, I. N. and P. E. Bourne (1998). "Protein structure alignment by incremental combinatorial extension (CE) of the optimal path." Protein Eng **11**(9): 739-47.

Stark, A. and R. B. Russell (2003). "Annotation in three dimensions. PINTS: Patterns in Non-homologous Tertiary Structures." Nucleic Acids Res **31**(13): 3341-4.

Stark, A., A. Shkumatov, et al. (2004). "Finding functional sites in structural genomics proteins." Structure (Camb) **12**(8): 1405-12.

Teichmann, S. A., A. G. Murzin, et al. (2001). "Determination of protein function, evolution and interactions by structural genomics." Curr Opin Struct Biol **11**(3): 354-63.

Torrance, J. W., G. J. Bartlett, et al. (2005). "Using a Library of Structural Templates to Recognise Catalytic Sites and Explore their Evolution in Homologous Families." J Mol Biol **347**(3): 565-81.

Valdar, W. S. (2002). "Scoring residue conservation." Proteins **48**(2): 227-41.

Wallace, A. C., N. Borkakoti, et al. (1997). "TESS: a geometric hashing algorithm for deriving 3D coordinate templates for searching structural databases. Application to enzyme active sites." Protein Sci **6**(11): 2308-23.

Wangikar, P. P., A. V. Tendulkar, et al. (2003). "Functional sites in protein families uncovered via an objective and automated graph theoretic approach." J Mol Biol **326**(3): 955-78.

Zvelebil, M. J. and M. J. Sternberg (1988). "Analysis and prediction of the location of catalytic residues in enzymes." Protein Eng **2**(2): 127-38.

Supplementary Materials for Chapter 2

Significance of Optimized GASPS Scores

As a way of determining whether any GASPS motif and its score are significant and likely represent more than a chance co-occurrence of residues, we computed empirical distributions of the scores of refined GASPS motifs due to chance alone. To ensure that the motifs were due to chance alone, artificial positive groups were generated by randomly selecting structures from the structure library, each with a different fold. The distributions of final GASPS scores on these artificial positive groups of size 5 and 10 structures are shown in the histograms in Figure i. As expected, these chance motifs do a better job of discriminating the smaller groups (have higher GASPS scores), though in both cases the discrimination is far from perfect. Using these distributions we can set GASPS score thresholds for moderate significance ($p < 0.01$): motifs found in groups of approximately 5 structures must score greater than 0.55 and in larger groups of 10 or more structures they must score above 0.4.

Sources of Variability

As a stochastic search method, GASPS generates some variability in GASPS scores by producing different motifs in identically configured runs. However, it is clear from other observations that some variability stems from variation between query structures. Recently, Torrance *et al.*(2005) reported variation above 1.0Å RMSD at active site motifs for 20% of protein pairs in a set of proteins with less than 20% sequence identity. This variation causes corresponding motifs from different query structures to provide different scores. For example, the GASPS score for the catalytic triad of trypsin-like structure 1hpg is 0.83, but the same motif from another trypsin-like protein, 1agj, is 0.92. To determine how much of the variation seen across GASPS results is due to the query structure or the stochastic nature of the genetic algorithm, the best, worst, and the two queries scoring closest to the median were chosen from each of these groups and run through GASPS five additional times.

Figure ii plots the GASPS scores for these runs (along with histograms of insignificant GASPS scores, as in Figure ii for reference). It is encouraging that the highest scores for each query structure seem to be the most frequent in most cases, especially for AHS and HADS. We also note that there are runs for which a high-scoring motif exists as indicated by other successful runs with the same query, but GASPS fails to locate it or any other significant motif; see for example 1hzy from AHS, and 1one and 1kko from ES in Figure ii. There are also structures that seem poorly suited for motif generation with GASPS, see 1itu (AHS), 1cqz(HADS) and 1bef(SP) in Figure ii. Manual examination of these structures suggests the active sites and other structurally conserved sites of these structures may be relatively deformed.

In many cases, while GASPS may fail to find the most significant motif, it can still find a significant motif. For example, three of the six GASPS runs with 2mnr (mandelate racemase from ES) as the query structure do not find the two metal ligands that all top-scoring 2mnr motifs contain, but they do find the conserved lysine (K164) and a nearby lysine (K166) in the active site, along with a distant pair of glycines (G44 and G47) located in the non-catalytic N-terminal domain. Similarly, two runs with 1one (enolase from ES) find another lysine (K345) important for catalysis in enolase(Babbitt et al.

1996) and a glutamine (Q167) that is very close to the active site. As an example from SP, the disulfide bridge near the active site occurred in all less optimal motifs for 1bqy and 2hlc (C42 and C58), usually along with one or more members of the catalytic triad (Figure iii b). Likewise, the less optimal motif for 1nrw in HADS still shares two functional aspartates in common with the best motif (D7 and D237), but matches a different third functional residue (D241 instead of K214) along with three other nearby residues (G42, G236, and A248). Thus, these suboptimal motifs may be useful for identifying additional residues that are potentially important for protein function.

In some cases, however, a suboptimal motif is best viewed as incomplete progression towards the optimal motif. For example, three different motifs that were found for 1ec7 (glucarate dehydratase from ES) all have similar scores and share the two most common metal ligands (D235 and E260). They differ by whether they include either a distant glycine (G115) located in the N-terminal domain, or the “downstream proline” (P262, see the *Enolase Superfamily* section above) with a glycine (G333) located at the N-terminal end of the barrel along with another distant N-terminal glycine (G74) in the third case. Similarly, the less than optimal motif for SP member 1fn8 (catalytic triad plus G196 and G197) is very close to the optimal motifs (catalytic triad plus G193), and GASPS likely would have settled on the optimal motif in this case with a few more rounds of optimization.

Taken together, these observations suggest that to generate a set of the most useful and inclusive motifs, limited resources are better spent on running GASPS on many different query structures than on running GASPS multiple times on the same structure.

Detection of New Unidentified Structures

The GASPS motifs in Figure 3 were used to search for additional group members among a non-redundant (<50% sequence identity) subset of the complete PDB in late September, 2004. Ten structures, previously unidentified by the authors, were identified by these GASPS motifs and later confirmed as group members by consulting the literature or expert collaborators: 7 from HADS, 2 from SP, and 1 from CS. The rate of new false positives at the chosen thresholds was also consistent with expectations. Thresholds were chosen based on a false positive rate of 0.0012 on the training set (5 false positives from among the original library of 4243 structures), and the rate on the newer subset of the PDB was identical within rounding errors (average 8.0 false positives from 6673 structures).

Allowing Substitutions in Motifs

In an attempt to improve the ability of GASPS to identify functionally important residues and identify motifs useful for classifying protein structures, we tested a scheme for allowing position-specific substitutions. Allowed substitutions were chosen based on the multiple sequence alignment used to measure conservation, and were fixed for a position throughout the GASPS run.

The introduction of substitutions allows GASPS to achieve much higher scores on the randomly generated, unrelated groups, though only for HAD and Crotonase superfamilies are statistically significant increases in GASPS scores observed (Wilcoxon ranked sum test(Hollander 1973), $p < 0.001$ and $p < 0.01$, respectively. Data not shown.) However, the improvements did not hold up to a full set of cross-validation analyses using all structures as queries and test structures (Table i). It appears that allowing substitutions raises GASPS scores for some families, especially when substitutions help to identify a few more functionally important residues. However, in its most simple current implementation, allowing substitutions in GASPS appears to be more prone to overfitting. Thus, we cannot say that allowing substitutions in the current scheme provides for more general motifs.

References

Babbitt, P. C., M. S. Hasson, et al. (1996). "The enolase superfamily: a general strategy for enzyme-catalyzed abstraction of the alpha-protons of carboxylic acids." Biochemistry **35**(51): 16489-501.

Hollander, M. D. A. W. (1973). Nonparametric statistical inference. New York, John Wiley & Sons.

Torrance, J. W., G. J. Bartlett, et al. (2005). "Using a Library of Structural Templates to Recognize Catalytic Sites and Explore their Evolution in Homologous Families." J Mol Biol **347**(3): 565-81.

Table i. Improvements in GASPS by using substitutions on Crotonase and HAD superfamilies.

	Average Training Scores		Average Test Scores		Number of test structures matched		
	Subs	No Subs	Subs	No Subs	Subs	No Subs	
CS	1.06	0.98	0.35	0.33	14	14	
HADS	0.85	0.63	0.5	0.45	70	59	$p=0.22^a$

(a) Fisher's exact test on count data for HADS alone.

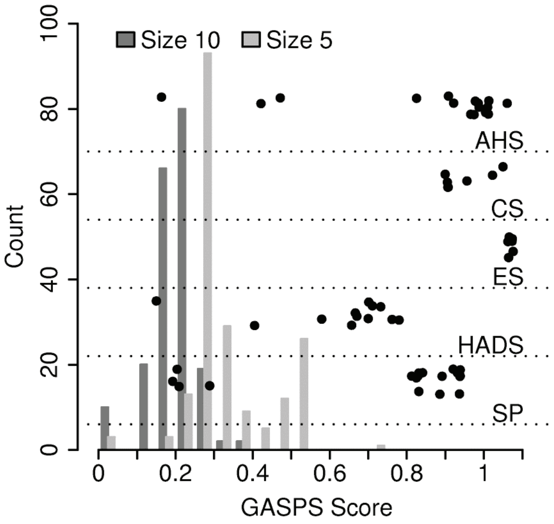


Figure i. Distributions of GASPS scores on artificial and real groups.
GASPS scores on randomly selected artificial groups of 10 and 5 structures are presented as histograms. GASPS scores on real groups (corresponding to motifs in Figure 2) are presented as scatter plots arranged by group; within groups the vertical placement is randomly chosen to avoid overlaps. Each point is a single GASPS run using a unique query chain. The counts on the y-axis are relevant only to the histograms.

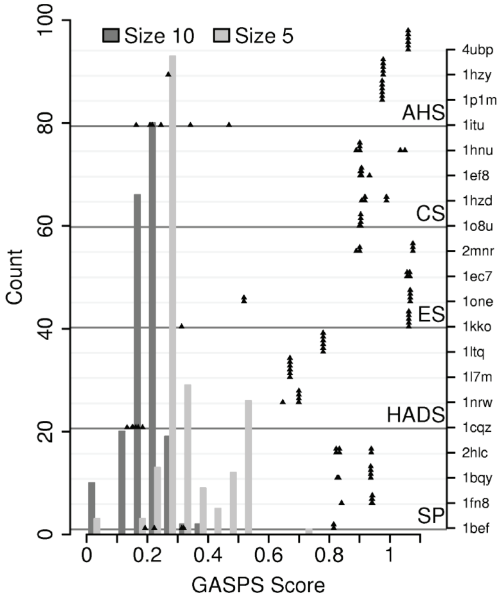


Figure ii. Stochasticity of GASPS results.

For each of the five groups, data are presented from six repeated GASPS runs on four query chains indicated by their PDB identifier. Stacked points (triangles) represent identical GASPS results for the query chain. Histograms are redundant with **Error! Reference source not found.**, but included here for reference. The counts on the left y-axis are relevant only to the histograms. The right y-axis and its connecting horizontal lines identify the query structure used to generate the GASPS scores that stack on each line.

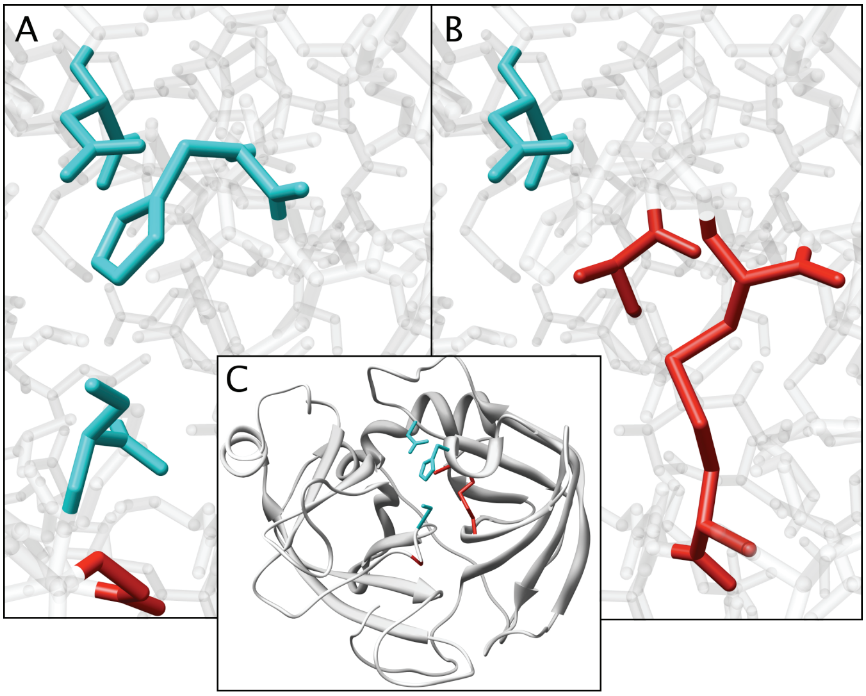


Figure iii. GASPS motifs for 2hlc, a trypsin-like serine protease.
Residues in motifs are highlighted in red or in cyan if the residue is also part of the catalytic triad. (a.) The top-scoring GASPS motif from among all runs includes the entire catalytic triad (H57, D102, S195) and a nearby glycine (G193). (b.) The top-scoring motif from an identically configured, but lower scoring, GASPS run includes D102 of the catalytic triad, and a nearby disulfide bridge and alanine (C42, C58, A55). (c.) Residues highlighted in panels (a) and (b) are shown relative to the entire domain.

Chapter 3: An analysis of computed expectations for random matches to 3D motifs

Introduction

The scientific evaluation of any hypothesis requires a method to evaluate the likelihood that observations could be explained by an alternative, usually simpler and less interesting, hypothesis. For searches of biological databases, in our case protein structures or 3D motifs, the hypothesis of interest is that any match is the result of a meaningful biological relationship, such as shared ancestry or function, and the alternative that a match is the result of chance placements of residues within the physical and chemical constraints of protein structure. For GASPS motifs, the biological relationship of interest is a group of proteins, defined by homology or functional similarity (Polacco et al. 2006). GASPS seeks to find a motif where all group members match within a deviation threshold stringent enough to make matches to non-group proteins rare. The original GASPS determines this threshold empirically, by searching for matches to each candidate motif among all non-group proteins that it should not match. Each run of GASPS on a single modern processor can take as long as 20 hours, and the repeated searches of a structure database take approximately 98% of this time. To see if this step could be replaced by a quicker approach, I evaluated a more theoretical approach to calculating the expected number of random matches to any motif. This is not solely a practical, statistical inquiry. If motifs are adequately scored by a theoretical model based only on geometry, this indicates conserved elements are products solely of the unique functional constraints of a group. Instead, I show evidence for the alternative, that the conserved elements of protein structures are often not unique to a group, so that general physical or chemical constraints of protein structure can lead to similar arrangements of residues in unrelated proteins.

Results

Modifying the GASPS scoring function.

GASPS was designed to find a motif that discriminates between a group of proteins that share a trait of interest (such as the serine proteases) and all other proteins. In effect, GASPS chooses a motif that is matched by all of the group proteins within an RMSD stringent enough to make random matches to other proteins rare. By searching each candidate motif against the background structure library, GASPS empirically computes an expectation for each true positive match based on its relative RMSD. This time-consuming step in GASPS can be replaced by the use of an accurate model that can be used to compute the likelihood of a match within any RMSD threshold. Stark et al. (2003) have developed a method for computing the expected number of matches to a motif based only on the number of residues in the motif, abundance of residues in the database, and number of atoms used per residue in the motif. We compare here the accuracy of using the computed expectation numbers (E_c) with actual expectations, or counts of false positives, computed empirically (E_e).

The majority of the GASPS score of a motif is based on the number of random matches that are expected at better or equal RMSD than to its matches to group proteins. This score is based on the area under an ROC curve to five false positives and so can be computed by summing the vertical “columns” on an ROC plot (Equation 1).

Equation 1

$$Area = \frac{1}{5}\sum_{i=1}^{5}\frac{T(f_i)}{T_{\max}}$$

$T(f_i)$ is the count of true positives with scores better than false positive f_i, and T_{max} is the maximum number of true positives, i.e. the number of proteins in the positive set. For purposes of this explanation, it will be clearer to instead take the equivalent area by summing the "rows" (Equation 2).

Equation 2

$$Area = \frac{1}{T_{\max}}\sum_{i=1}^{T_{\max}}\left(1 - \frac{F(t_i)}{5}\right)$$

$F(t_i)$ is the number of false positive matches with an RMSD equal to or better than that of a given true positive match, t_i, and is assigned a maximum value of five for the equation above. F(t) is treated here as the empirical expectation value (E_e). The summed term (1-$F(t_i)/5$), is the "credit" granted to each true positive match t_i by an ROC area calculation. If a true positive matches better than any false positive, it is given full credit (1-0/5), with partial credit granted if it matches worse than only a few (x) false positives (1-x/5), and no credit (1-5/5) if it matches worse than 5 or more false positives. Traditionally, a probability (P) value can be assigned to any match from an expectation value according to Equation 5. This ROC credit term is approximately equivalent to this P value subtracted from 1 (see Figure 1), so for explanatory purposes we define an "empirical probability" P_e by Equation 3.

Equation 3

$$P_e(t) = \begin{cases} F(t)/5 & \text{for } F(t) \le 5 \\ 1 & \text{for } F(t) > 5 \end{cases}$$

The ROC area component of the G score calculated by Equation 2 is then just the average of (1-P_e) for the best match for each protein in the positive set. With the GASPS scoring function now viewed in terms of P values, a theoretical system for computing P values can easily be substituted for the empirical system. Because a theoretical computation of P values should be continuous, the second term used in the empirical scoring function, which served mainly to break ties when the discrete P_e gave identical scores, is not necessary.

The computed expectation, E_c, is the expected number of matches in the negative set and can be computed at any RMSD (R) by Equation 4.

Equation 4

$$E_c(R) = DA\Phi a_3^N R^{2.93N-5.88}[c_2R^2]^N$$

In this formula, taken form Stark et al. (2003), D is the number of proteins in the database, N is the number of residues in the motif, Φ is the products of abundances (as percentages) of residue types, and the remaining parameters are empirically derived constants: $A=3.70\times10^6$, $a_3=1.79\times10^{-3}$, c=0.196. The expected number of matches is converted to a P value using Equation 5, which depends on a Poisson distribution of matches among the protein structures.

Equation 5

$$P = 1 - e^{-E}$$

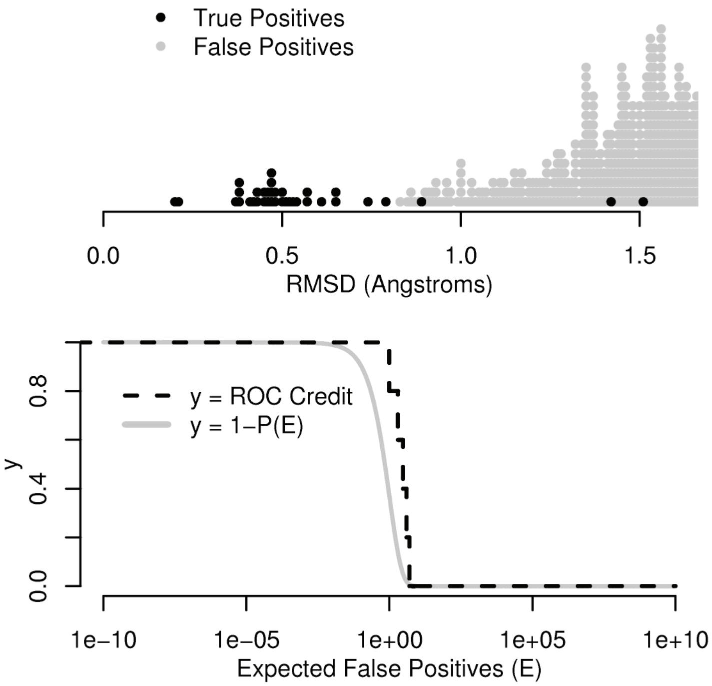


Figure 1. Relation between "ROC Credit", P Values, and Expected False Positives. These two plots are intended to demonstrate the similarity between calculating a P value based on a Poisson distribution and continuous expectation values, and ‘P values’ based on number of observed false positives (1- ‘ROC Credit’). The top plot shows the distributions of true positive and false positive matches to an actual motif from a serine protease. It is drawn so that its RMSD values roughly correspond to the E values on the lower plot. For both systems of scoring true positives, the majority of true positives (black dots) are scored equivalently. Those on the left are given a score of 1, or approximately 1, and those on the right are given a score of 0, or approximately 0. For this case, only the three true positives near E=1 will be treated differently.

Expectation values compared.

Figure 1 shows that values of P_e and P_c are nearly identical given the same expectation values. Therefore, GASPS modified to use a computed score as opposed to an empirical score should provide a similar result as long as E_c accurately tracks E_e. One important difference between E_c (Stark et al. 2003) and E_e as defined here is that E_c predicts the total number of matches, including multiple non-independent matches in the same protein, whereas E_e counts only the number of proteins with one or more matches[1]. Multiple matches to a motif frequently occur within the same protein and make use of some of the same residues, but only the best match is usually biologically significant (Meng et al. 2004). Coincident motifs occur more often than expected by chance: assuming 5 matches are assorted randomly across 4000 proteins, the likelihood of any coincident match in any protein is 0.002 (by 'Birthday Paradox' arguments). Forty percent of motifs with 6 residues, when searched against the negative set used in Chapter 2, have repeated proteins among their best 5 matches. For smaller motifs, this becomes less of a problem with the same percentages being 22%, 9%, and 5% for motifs with 5, 4 and 3 residues, respectively.

I examined the actual relation between E_e and E_c by computing E_c for each false positive match to a set of motifs, and computing E_e by counting the number of actual false positive matches that score with the same or lower RMSD. The motifs used were the high-scoring "surviving" motifs from each round of a single GASPS run for two randomly chosen structures from each of the five groups studied in Chapter 3. Figure 2 shows that while E_e and E_c tend to agree, there is a large degree of variation in both directions: E_c can both underestimate and overestimate E_e depending on the motif. Some of this discrepancy has been previously reported and is not unexpected: motifs that include physically favorable relationships such as salt bridges or disulfide bridges are

---

[1] Equation 5, which is used to compute a P value (the probability of any random match) from an expectation value (the average number of random matches) is based on a Poisson distribution, which assumes independence of counted objects. Repeat matches in the same structure are usually not independent and it would be best to not count them. Regardless, the difference is minor at low expectation values.

expected to occur with greater frequency than a model based solely on geometry would predict. There is an association with motif sizes—matches to the larger motifs are predicted to be more numerous than are actually observed ($E_c > E_e$). This is due in part to the fact that E_c counts similar matches within the same protein, but also because GASPS restricts single atom-pair distance deviations between match and motif to reasonable distances (sidechain-sidechain deviation < 3.8 Å; α-carbon - α-carbon deviation < 5.0 Å). Relaxing both the sidechain and α-carbon distance deviations to 15 Å allows for many more matches at the same RMSD giving a closer correspondence between E_e and E_c, but we feel that deviations up to 15 Å cannot represent true correspondences. It appears from this that GASPS with computed scores would favor smaller motifs as compared to GASPS with empirical scores because Equation 4 tends to over-predict the numbers of false positives for the larger motifs.

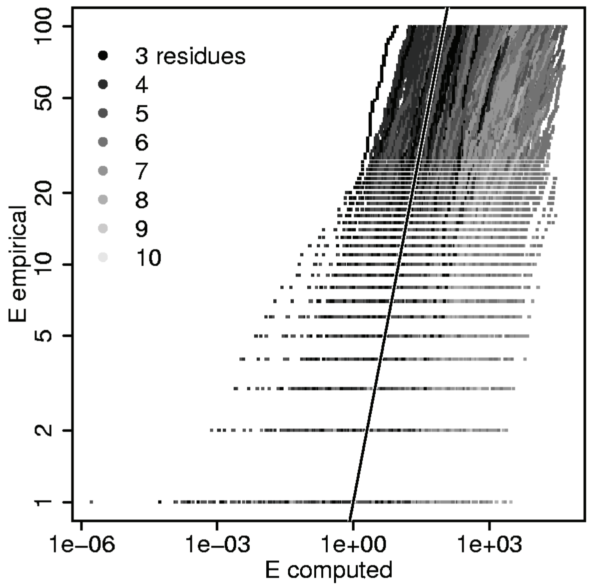


Figure 2. Empirical counts of false positives versus computed expectation values. Each point represents a single false positive match to a motif sampled by GASPS. E empirical is the count of false positives that match with equal or better RMSD, and E computed is calculated based on Equation 4. The solid line is the equivalence point where $E_e = E_c$. Points are shaded by the number of residues in the motif.

GASPS scores compared.

The above discussion of expectation values focuses on the expectation values at values of RMSD from false positives alone. The computation of G scores, both by empirical (G_e) and computational (G_c) methods, depends on the expectation values of true positive or group member matches. While the correspondence should hold between expectations of true positives as it does for false positives, the relevant expectation values are at values that are too low to compare accurately by an analysis like Figure 2. We directly compared the G_e and G_c values for each of the motifs used in the above E_e and E_c comparisons. While we see the same overall correspondence, the main difference being that G_c has a range of 0-1.0 and G_e a range of 0-1.1, there are two noteworthy trends: First we see the expected trend for G_c to be an underestimate of G_e for the larger motifs. Second, there appears to be more distant outliers above the line than there are below. These motifs are predicted to occur very often by G_c but in fact occur much less often than expected. These may represent motifs that are usually physically unfavorable, but are necessary for a unique characteristic of function. For the enolase and amidohydrolase superfamilies, these outliers ($G_e > 0.9$ AND $G_c < 0.6$) are small motifs dominated by very close negatively charged residues that act as metal ligands.

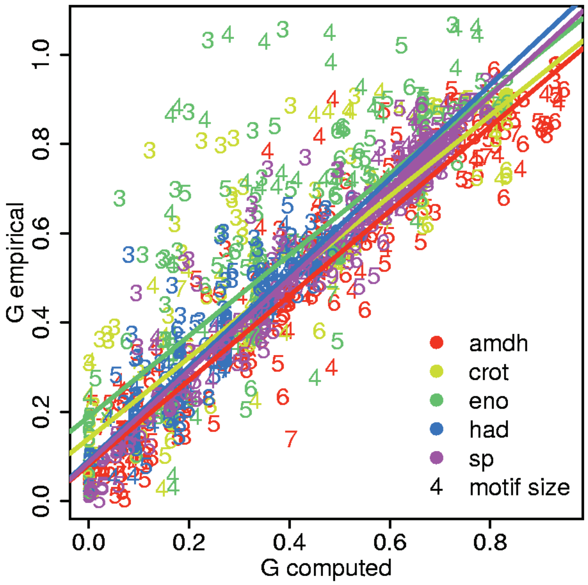


Figure 3. GASPS scores (G) compared between empirical and computed methods. Points are drawn as numbers which describe the number of residues in a motif. The color indicates the group: amidohydrolase superfamily (amdh), crotonase superfamily (crot), enolase superfamily (eno), halo-acid dehalogenase (had) or serine proteases (sp). Colored lines are lines fit by linear least-squares regression on the data split by groups.

GASPS with G_c on Random Groups

The significance of a motif found by GASPS is measured by comparing it against the distribution of motifs generated on randomly constructed groups. For an accurate comparison I used the same randomly constructed groups as used previously for GASPS that used G_e (Chapter 2, Supplementary Materials). Here we see the first major difference in results given by using G_c instead of G_e. GASPS with G_c generates motifs with higher scores for unrelated proteins (Figure 4). The biggest peak for the distributions of groups with 5 and 10 structures corresponds to matching two or three other structures in addition to the query structure. It appears that three unrelated structures chosen at random often share a motif by chance alone. For groups of 10 structures we can set a significance cutoff at approximately G_c=0.5 and for groups of 5 structures the same threshold is as high as G_c=0.75.

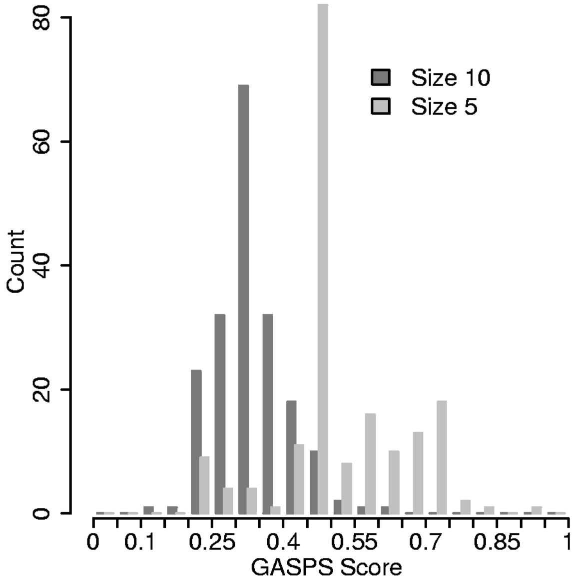


Figure 4. Distributions of motifs by GASPS with G_c on random groups.

Composition of motifs

I next made use of G_c in an updated version of GASPS on the same set of superfamilies and structures as used in Chapter 2. While the new GASPS was much faster, both the distributions of scores and the makeup of motifs were similar (see Figure 5; compare with Chapter 2, Figure 3). I show only the result of a single run of GASPS for each query structure so differences at the level of individual motifs are not significant. Surprisingly we do not see the expected trend for smaller motifs to be favored by G_c. Instead, we actually see larger motifs. This is likely the result of trends at very low expectation values, which cannot be adequately represented by the data in Figure 2.

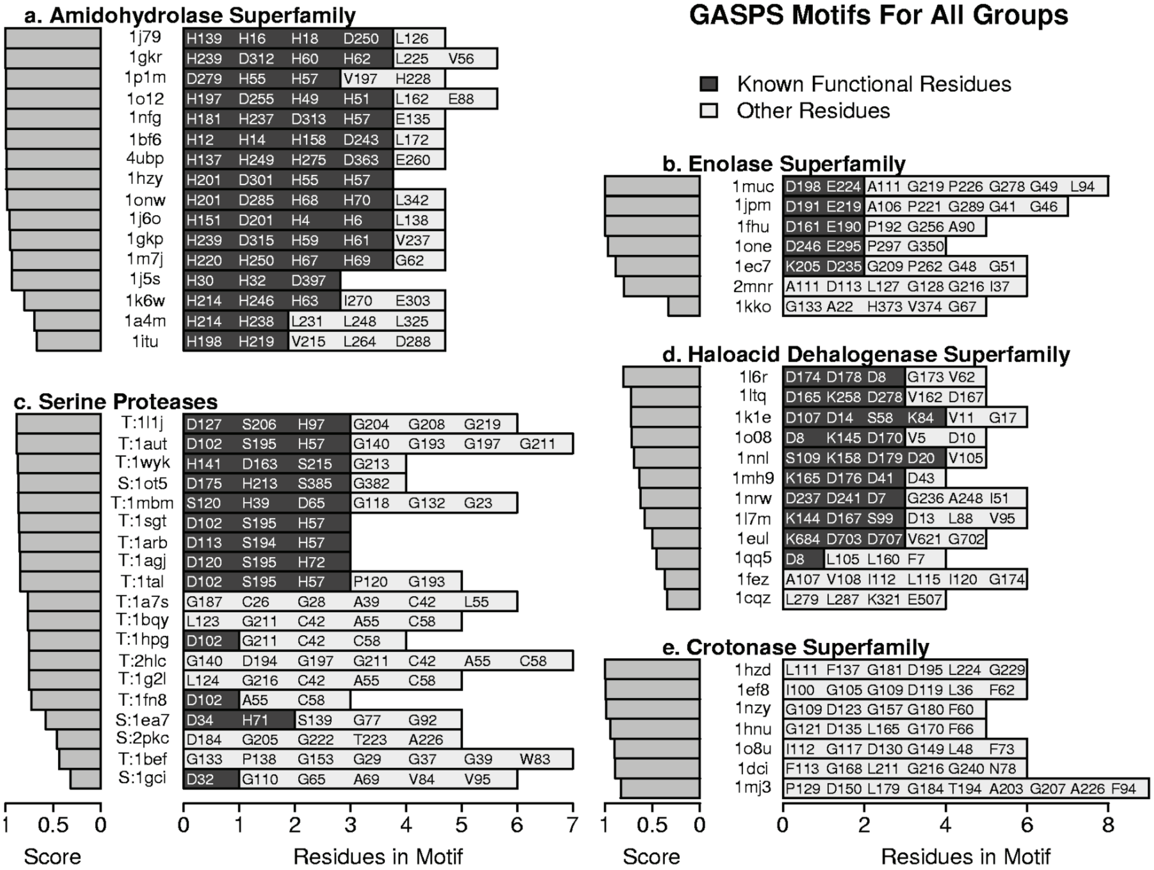


Figure 5. Composition of motifs generated by GASPS with computed G scores. The results of a single GASPS run are presented for each named query structure. Residues in the motif that correspond to previously identified functional residues or known active-site motif residues are darkly shaded. All other residues are lightly shaded regardless of subsequent determination of their functional significance. For the serine proteases, query structures are labeled "T:" to denote trypsin-like folds or "S:" for subtilisin-like folds.

On 92 groups of enzymes defined by the Enzyme Commission (EC) enzyme naming scheme (International Union of Biochemistry and Molecular Biology. Nomenclature Committee. et al. 1992), I asked whether the degree of overlap with catalytic residues was different depending on the method of computing G scores. I used the Catalytic Site

Atlas (CSA) as the source of catalytic residues (Porter et al. 2004). Only the motifs that were considered significant by the G thresholds discussed above were counted. There was no significant difference in the number of overlaps with residues called catalytic by the CSA (see Table 1).

Table 1. Overlap of significant motifs with catalytic sites in CSA.

	CSA overlap	No overlap	% with CSA overlap
Empirical	88	59	60%
Computed	93	46	67%

Chi-squared = 1.237, df=1, p-value = 0.2660

Accuracy of motifs at identifying homologous groups

The final test of using GASPS with G_c scores was to test the effectiveness of the generated motifs at identifying new protein structures to groups. I chose to use homologous groups, the families and superfamilies in the Structural Classification of Proteins (SCOP) version 1.65 (Murzin et al. 1995), because these groups produced higher scoring motifs than available classifications based on function. These motifs are tied to function and are useful at identifying function (see Chapter 4). Only groups with at least 7 structures, after removing redundancy at a threshold of 25% sequence identity, were included. One motif was generated for each structure in each group for both GASPS with G_c and GASPS with G_e. The domains newly added in SCOP version 1.67, compared to version 1.65, were searched against libraries of generated motifs with the program RIGOR (Kleywegt 1999). Only the first match to all motifs by each new domain were counted. Each match was given a P value using Equation 4 and 5, and both sets of motifs matched family or superfamily members at the same rates (true positives), but the motifs generated by computed G scores matched non-group members (false positives) at higher rates, significantly decreasing accuracy. Clearly, the motifs generated using computed G scores are not as specific to the group as those generated by empirical G scores. Several factors may account for this loss of specificity. Computed G scores allow for motifs that may identify a broader homologous class: fold instead of superfamily, or superfamily instead of fold, or they may simply consist of a very general protein motif such as disulfides or hydrophobic clusters. Most of the GASPS with G_c runs that resulted in a much higher scoring motif than the corresponding run with G_e included disulfides or hydrophobic clusters, especially leucines.

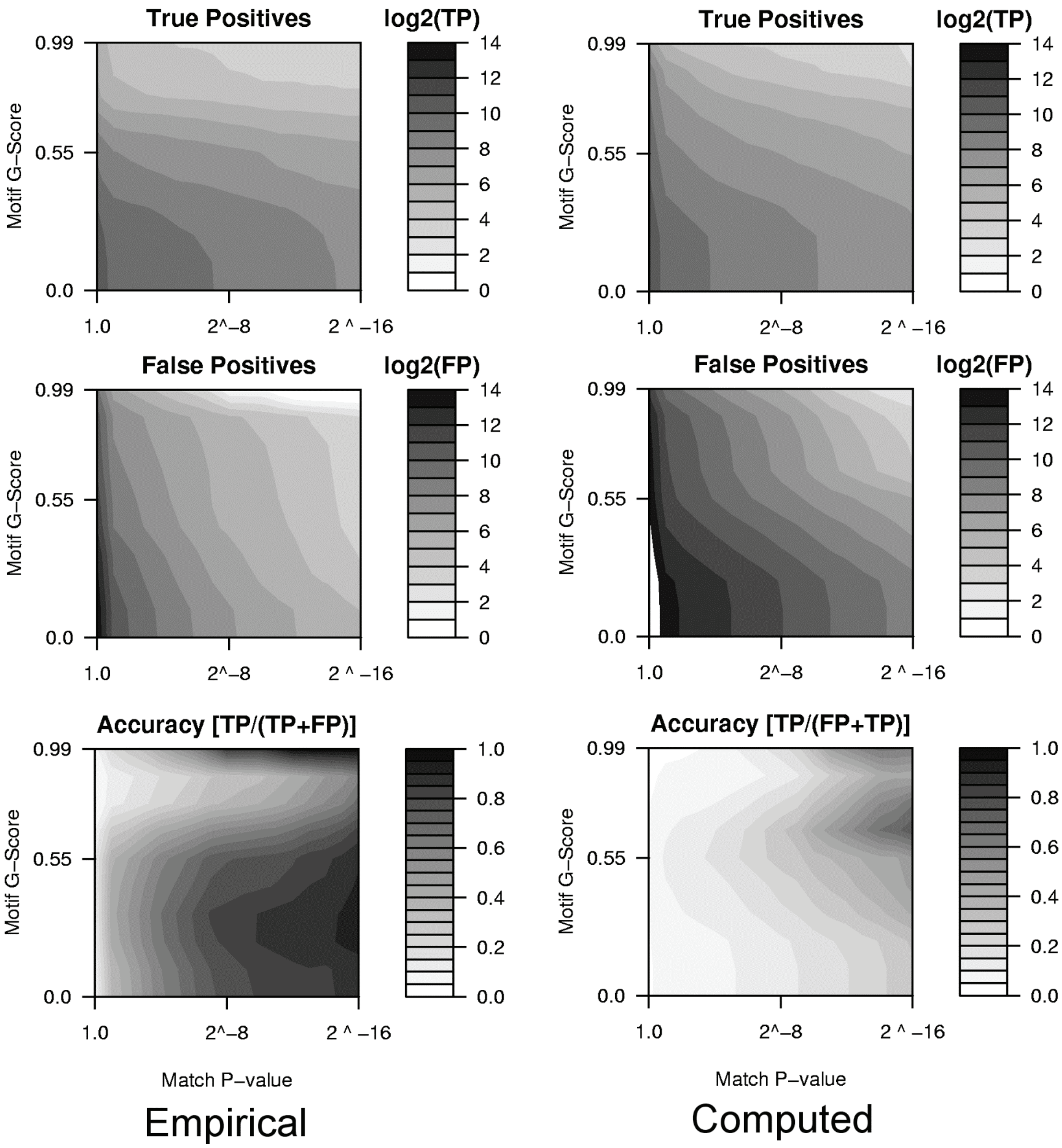


Figure 6. SCOP superfamilies identified by motifs generated by empirical G scores compared to computed G scores.

In all plots, the P value and G score axes are treated as thresholds. All matches with higher G scores or lower P values are counted and plotted at that location. The true positives and false positives are reported as the base 2 logarithms of the count. The x-axis is a logarithmic scale that decreases from left to right.

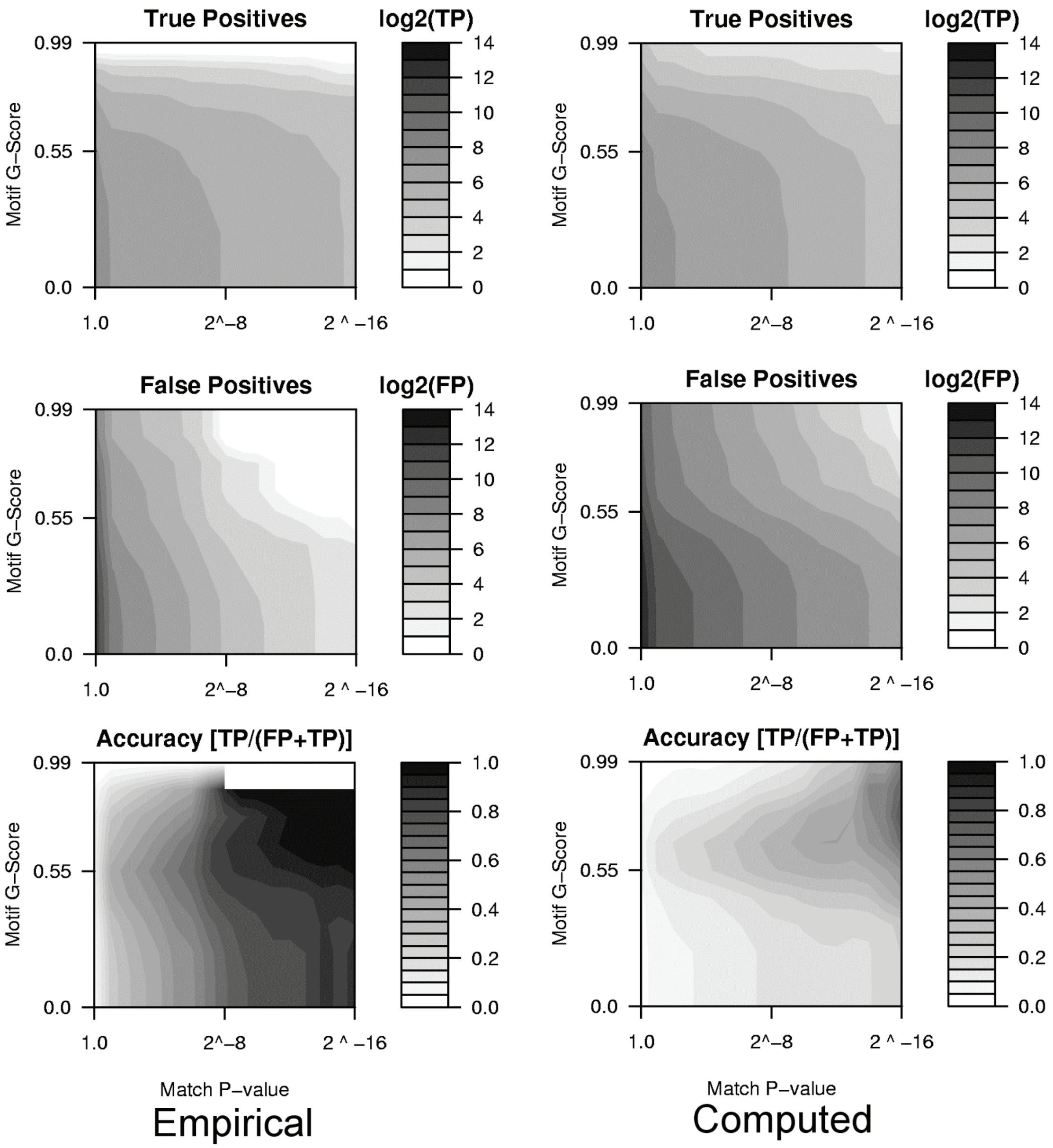


Figure 7. SCOP families identified by motifs generated by empirical G scores compared to computed G scores.
See legend for Figure 6.

Discussion

I tested a new scoring function that could potentially make the running of GASPS much faster. It uses a geometrical model to predict the expected number of random matches to a motif to identify how significant are the true positive matches. Motifs generated with this scoring function appeared to be very similar to those generated with the empirical scoring function. Functional residues were identified with similar rates. As a tool to find a conserved structural pattern within a group, this faster scoring function appears adequate, and future studies should consider using the faster approach. I have discovered two significant shortcomings of this new scoring. First, motifs discovered by the new scoring function have to achieve a much higher score to stand out as significant. This problem is made greater by the fact that G_e scores tend to be greater than G_c scores overall (see Figure 3). Second, the specificity of motifs made using G_c have lower specificity. To make these motifs as useful as possible for annotation of new protein structures I need to eliminate as many false positives as possible. These problems with G_c were serious enough that I chose to not use it for further analysis here.

References

International Union of Biochemistry and Molecular Biology. Nomenclature Committee. and E. C. Webb (1992). Enzyme nomenclature 1992: recommendations of the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology on the nomenclature and classification of enzymes. San Diego, Published for the International Union of Biochemistry and Molecular Biology by Academic Press.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Mol Biol **285**(4): 1887-97.

Meng, E. C., B. J. Polacco, et al. (2004). "Superfamily active site templates." Proteins **55**(4): 962-76.

Murzin, A. G., S. E. Brenner, et al. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures." J Mol Biol **247**(4): 536-40.

Polacco, B. J. and P. C. Babbitt (2006). "Automated discovery of 3D motifs for protein function annotation." Bioinformatics **22**(6): 723-30.

Porter, C. T., G. J. Bartlett, et al. (2004). "The Catalytic Site Atlas: a resource of catalytic sites and residues identified in enzymes using structural data." Nucleic Acids Res **32 Database issue**: D129-33.

Stark, A., S. Sunyaev, et al. (2003). "A model for statistical significance of local similarities in structure." J Mol Biol **326**(5): 1307-16.

Introduction to Chapter 4

All the work described in previous chapters sets the stage for a broad application of GASPS across as many proteins as possible. To understand the benefit of these generated motifs it is important to remember the two main scientific functions that motifs serve. First, motifs provide a way to classify proteins to groups, so such a broad study seeks to generate the maximum impact that GASPS can provide in this area. Beyond this, it also can provide an assessment of how broadly and on what groups the technique of 3D motifs can be effective. Second, motifs represent the most conserved elements in protein structures. The distribution of these motifs among protein groups can answer questions about how local protein structure has evolved. Without a shift in overall function, how often does the evolution of proteins maintain the same set of critical residues? As functions change, how often do proteins make use of existing functional components? Furthermore, the composition of the motifs provides information about what features of proteins tends to be the most conserved and therefore the most critical for maintaining the function of proteins as they evolve. The distribution of these motifs and features among different types of proteins and different types of classifications indicate trends in evolution.

This chapter forms the basis of a manuscript that will be submitted for publication to a peer-reviewed journal.

Chapter 4: An exhaustive survey of 3D motifs.

Abstract

Knowledge of local protein structure, such as individual residues or clusters of interacting residues, is essential for understanding how protein structure delivers function, and especially how structure and function evolve together.

Here we examine the evolution of fine scale protein structure by determining the distribution of conserved 3D motifs, and what structural features tend to be conserved. We apply GASPS, which identifies the most conserved and unique motif from an input group of protein structures, to SCOP superfamilies and families as well as isofunctional groups defined by the Gene Ontology.

We find that homologous relationships are more important than functional relationships for the presence of a highly conserved motif. Non-homologous but isofunctional groups do not commonly share a motif. This suggests that protein functions, as they are commonly described, are usually accomplished by different means in unrelated proteins. About one third of all SCOP groups show a strongly conserved motif. The lack of a conserved motif in the remaining two thirds of groups reveals that evolution of new functions is commonly not constrained to maintain the positions of a critical set of residues.

We describe the patterns of structural elements and residue types among motifs to reveal trends in conservation of local structure. As expected, the motifs from all groups show a strong link to function, frequently overlapping with known catalytic, metal and other

ligand sites. Additionally, disulfides as well as stabilized charged residue pairs are overrepresented among the most conserved motifs. Residue distribution among the motifs is mostly as expected based on these common elements: cysteine, histidine, aspartate and glutamate are among the most frequent. More surprisingly, glycine is the most common motif residue and glycine in motifs has the greatest rate of non-metal ligand interaction among all other motif residues.

The motifs generated in this study are available via a web resource named GASPSdb, which is effective for annotating protein structures as well as highlighting important residues in new structures. Using these motifs, we show that 3D motifs offer promise for annotating low quality homology models built on distantly related templates.

Introduction

With the recognition that diverse and varied proteins can make use of the same overall fold, recent efforts in computational protein structural biology have shifted from examining large-scale features such as an entire sequence or fold, to a more focused examination of fine-scale features such as the orientations of a small number of sidechains. While the large-scale approach is useful for identifying homology, the fine scale approach allows for the identification of shared and distinct functional differences not apparent from the wider, large-scale view (Watson et al. 2007). One fine-scale approach, the use of 3D or structural motifs (sometimes called templates), relies on the cataloging of functionally related structural components comprised of the types and orientations of a small number of residues or their functional atoms (Fetrow et al. 1998; Laskowski et al. 2005; Torrance et al. 2005). Because these motifs normally contain the elements that actually deliver function, finding these motifs in newly solved or modeled structures can imply a likely function for the protein and a hypothesis at where and how the function is performed.

The development and investigation of this fine-scaled 3D motif approach requires two main components. First, we need tools that can search for matches among protein structures. There are already many well developed available motif search methods (for example, Artymiuk et al. 1994; Fetrow et al. 1998; Barker et al. 2003). Second, we need to know what motifs are indicators of what function. In other words, we need knowledge about how the requirements of function constrain fine scale protein structure, and whether the constraints are strong enough so that the same fine-scale structure will be conserved over great evolutionary distances. While there are observed cases of 3D motifs being tied to a specific function over great evolutionary distances (Meng et al. 2004), no study has yet shown how common such motifs are among a broad cross section of the protein universe.

Here we use our previously described technique named GASPS (Polacco et al. 2006), to describe the patterns of motifs from all groups of proteins, defined by homology and/or function, restricted only by available structures. Such a broad collection of these fine scale motifs provides a resource to aid annotation of protein structures, answers how universally we might be able to apply these focused 3D motif methods, and provides us with a source of more basic biological knowledge. Each motif is a hypothesis about which residues are so important for the function of a protein as to be irreplaceable. These critical residues can provide, for example, a specific step in a catalytic mechanism (Gerlt et al. 2001), a very specific orientation of a binding partner, an important stabilization of an active site, or a specific geometry of the peptide backbone (Dym et al. 2001), to name just a few. Having a broad collection will also allow us to answer what types of residues and structural features tend to be most rigidly conserved. Knowing this can inform future studies of the evolution of protein function within families.

By considering groups defined by homology and function, we recognize that the mechanism of protein function and other conserved details of protein structure can be products of both natural selection and evolutionary history. By examining groups defined by function alone, we are testing the hypothesis that natural selection to perform the same function on unrelated proteins can shape similar functional sites. For groups defined by homology alone, presence of a motif, especially when it is clearly related to functional sites, supports the hypothesis that an existing functional component can be recruited to perform new overall functions. In these cases, the motif cannot indicate a protein's overall function, but instead can indicate a conserved functional step. For example, this might be a single step in an enzyme's reaction pathway, or simply the binding of a metal ligand (Gerlt et al. 2001). Finally when we examine the most specific groups, those that are both homologous and isofunctional, we allow for the detection of motifs in cases where a new function evolved with a new set of critically important residues with no need for ancestral residues.

Ours is not the first publicly available collection of motifs (Stark et al. 2003; Torrance et al. 2005), but it does provide new coverage and a new emphasis on classification ability. Moreover, our technique is a protein-group driven approach that lets us find motifs, if they exist, for all protein groups. Other motifs have typically been chosen based on available prior knowledge of functionally important residues such as the catalytic triad of the serine proteases. These motifs have been shown successful at identifying specific enzymatic activities (Torrance et al. 2005), binding relationships (Artymiuk et al. 1994), and superfamily membership (Meng et al. 2004). The catalytic site atlas (CSA) is a database of high quality catalytic residue designations gleaned from the literature (Porter et al. 2004). It currently provides 147 non-redundant active site motifs for enzymes (Torrance et al. 2005). In lieu of a literature search, a faster method, though more error-prone, is to use the information imbedded directly in protein coordinate files, such as SITE records, or proximity to ligands. This technique has been used by PINTS (Stark et al. 2003) and others (Artymiuk et al. 1994; Kleywegt 1999; Laskowski et al. 2005). Our approach, GASPS, is automated, unbiased and applicable to any group of proteins with sufficient structures (Polacco et al. 2006). It seeks to choose motifs with a high degree of classification ability, measured by a motif's tendency to match group members with high sensitivity and specificity. While it is not biased by accepted trends of functionally important residues, the motifs it finds have been shown to overlap with known functional sites.

We describe here the motifs generated by applying GASPS to isofunctional groups defined by Gene Ontology (GO) molecular function terms (Ashburner et al. 2000), homologous groups defined by the Structural Classification of Proteins (SCOP) superfamilies and families (Murzin et al. 1995), and by homologous isofunctional groups defined by both GO terms and SCOP superfamily. For purposes of protein annotation, we find that GASPS motifs can provide coverage of proteins unavailable in existing motif libraries. Moreover, we find that while the motifs, for the most part, can be related to known functional sites, homology is more important than function for determining the presence of a high-scoring motif.

Methods

GASPS

GASPS (Genetic Algorithm Search for Patterns in Structures) takes as input a group of proteins, the positive group, and a background set of other proteins. It seeks to choose the coordinates of a set of residues from a single positive group member that is well matched by all other members of the positive group, and not matched by members of the background set. Further details are described in an earlier publication (Polacco et al. 2006). GASPS was run once for each member (chain or domain) of each protein group, generating as many motifs for each group as there are members.

Protein groups

For each classification, no groups of proteins were allowed with fewer than seven structures when reduced to a non-redundant set based on a 40% sequence identity cutoff. Where possible without going below seven structures, those groups with sufficient structures and diversity were further reduced based on a 25% sequence identity cutoff. This generated two sets of groups, those that could be reduced to a 25% sequence identify cutoff and those that could not. Homologous groups were created by gathering all domains in SCOP (version 1.65) families and superfamilies. Isofunctional groups were defined by gathering all protein chains that share a single GO molecular function term, including the terms implied by the “is a” hierarchy of GO. It is desirable to limit GO terms to those that are not so general as to make highly improbably any motif. To generate motifs for all suitable terms, and eliminate the obvious artificial groupings (such as all structures sharing GO term 5488, “Binding”), groups were discarded if they had greater than 50 non-redundant structures. Isofunctional homologous groups were generated by gathering all protein chains that shared at least one homologous domain as defined by SCOP superfamilies, then generating unique, but not mutually exclusive, groups defined by GO molecular function terms. Table 1 gives the counts of groups and generated motifs.

Searching motif libraries with proteins

We use the program RIGOR (Kleywegt 1999) to search the libraries of GASPS motifs. RIGOR returns all matches between a protein and each motif that satisfy a superposition RMSD threshold and a maximum distance deviation threshold. Because smaller motifs match randomly with much greater frequency than large motifs, the RMSD threshold was set per motif based on a computation of the number of expected random matches (E-value). GASPS uses relative RMSD between false positive and true positive matches to the same motif to determine the quality of the motif. When we compare matches involving different motifs, the RMSD becomes less meaningful. Random matches to a motif comprising three leucines are much more likely than to a motif with five tryptophans. We computed an E-value based on a slight modification of the method of Stark et al. (Stark et al. 2003) that accounts for residue background frequency, number of residues, and distance between atoms in each residue:

$$E = a_0 \Phi a_3^N R_M^{2.93N-5.88} \prod_{!gly} \frac{R_M^2}{d_r^2}.$$

Here, Φ is the product of residue frequencies as percentages, N is the number of residues, and R_M is the rmsd. All other variables are experimentally determined constants. The constant a_0 accounts for the size of search space; we used $a_0=1.57x10^{10}$ for all searches. While using the same value for a_0 regardless of database size can lead to inaccurate estimates of true expectation values, doing so generates an accurate score that reflects the strength of a pairwise match, allowing for direct comparison of matches between different searches. The right-most product is over each non-glycine residue in the motif, and corrects for the two-atom nature of GASPS and RIGOR non-glycine residues: an α-carbon and a sidechain centroid. Individual factors in this product are ignored when the ratio R_m/d_r, where d_r is the average distance between side chain centroid and α-carbon for a residue type r, is greater than 1. The value for a_3 was taken directly from Stark et al. (2003) at 0.00179.

Table 1. Group and motif counts by classification.

Classification	Groups	Redundancy Filter	# Groups	# Motifs	Avg # Motifs per Group
Gene Ontology	**Molecular Function**		**272**	**4385**	
	(7 < n < 50)	25 PID	177	2593	14.6
		40 PID	95	1792	18.9
SCOP			**323**	**3599**	
	Superfamilies		**186**	**2259**	
		25 PID	131	1801	13.7
		40 PID	55	458	8.3
	Families		**137**	**1340**	
		25 PID	64	670	10.5
		40 PID	73	670	9.2
GO and SCOP	**Superfamilies**		**376**	**4581**	
		25 PID	231	3318	14.4
		40 PID	145	1263	8.7

Results

Quality of Motifs

One goal of the current work is to study the broad applicability of 3D motifs across different types of proteins and different types of classifications. A motif's quality is described by its G score, short for GASPS score, which indicates the motif's ability to identify all other group members with high specificity. Ranging from 0 to 1.1, its main component is the area under a shortened receiver-operator characteristic (ROC) plot. The other component is the relative separation between true and false positive matches and accounts for only 0 to 0.1 of the total G-score. Therefore, most G-scores above 1.0 imply perfect separation in an ROC plot (ROC area = 1.0) though any score above 0.7 is highly significant and scores below about 0.4 are highly suspect. G-scores can give us a sense of how well groups of proteins can be identified by motifs. A high scoring motif for a group is evidence of a unique evolutionary constraint on that group. For homologous groups, the G-scores tell us whether there is a single structural pattern that evolution has not been able to alter, presumably without a loss of protein function. For non-homologous, functionally similar groups, high G-scores indicate convergent evolution where independent inventions of the same function required a common pattern of residues.

Previous work with GASPS showed it to be very effective on a small number of well-studied superfamilies. In this study we included all SCOP-defined superfamilies and families with sufficient structures (see Methods) as well as groups defined by common Gene Ontology (GO) annotations. Figure 1 and Figure 2 show the distribution of the highest scoring motif for each group in SCOP, GO and the GO/SCOP groupings. While a large number of SCOP families and superfamilies have very high G-scores, the majority of protein groups produce motifs with G-scores lower than for the previously well-studied superfamilies (top G-score for haloacid dehalogenase superfamily is 0.8, (Polacco et al. 2006). However, the evolutionary distance between members in a group is important. Those groups composed of members that all share less than 25% sequence identity have significantly lower scores (median G-score = 0.54) than those where we permitted up to 40% sequence identity (median G-score = 0.90). Evolutionary distance measured by sequence identity is even more important than is the evolutionary distance implied by SCOP hierarchy depth: the superfamily and family distributions are much more similar than are the distributions grouped by sequence identity. Because group size is correlated with G-scores, and the groups where we allowed 40% sequence identity were smaller, this could also be a result of group size effects. However, a linear model fit to this data to predict G-scores with parameters for group size, SCOP hierarchy and percent identity, shows the greatest effect to come from percent identity, the second greatest from group size, and finally the effect from SCOP hierarchy depth is nearly insignificant.

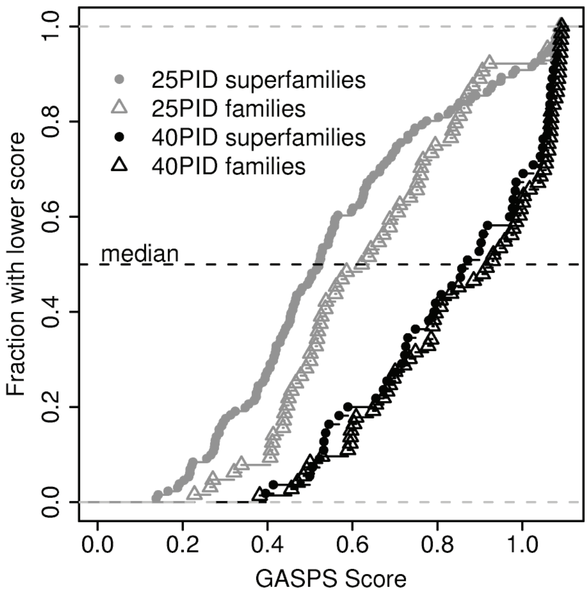


Figure 1. Distribution of motif G-scores on SCOP groups.

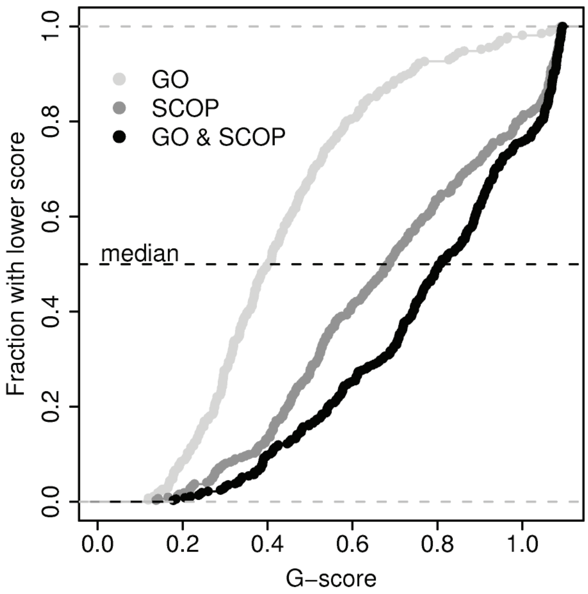


Figure 2. Distribution of motif G-scores on Gene Ontology and SCOP groups.

For groups defined by GO molecular function annotations, the G-scores were even lower than for homologous groups. This is the result of GO defined groups containing unrelated proteins that perform the same molecular function, but doing so in a different way with a different set of important residues. Even when GO groups are matched by a high-scoring motif, the groups are made up of, or at least dominated by, a single homologous group that accounts for the high G-score (not shown). The frequency with which evolution has invented new ways of performing the same function indicates that most protein functions can be performed in many different ways.

If we further subdivide the SCOP superfamilies by GO molecular function annotations, we see an improvement in G-scores over both SCOP superfamilies and GO groups. This shift to higher scores is not simply the result of chance due to the use of dividing the superfamilies in to smaller groups. In fact, the groups in this classification are larger on average owing to the groups not being mutually exclusive. For each superfamily, multiple overlapping groups can be defined depending on the precision of the GO annotations assigned to the individual structures. For example, SCOP superfamily "FAD/NAD(P)-binding domain" (c.3.1) has a group for GO term 16491 "oxidoreductase activity" as well as a group for GO term 15036 "disulfide oxidoreductase activity" that is a subset of the other. This actually results in a larger average group size compared to SCOP superfamilies (see Table 1). The larger groups are over-represented compared to the smaller groups because there are more ways to divide them.

The results on the GO/SCOP groups indicate that groups with more functional diversity are less likely to have a conserved motif. To investigate this finding in more detail, we counted the number of distinct enzyme commission (EC) (International Union of Biochemistry and Molecular Biology. Nomenclature Committee. et al. 1992) classes at the first and second positions for each superfamily and family in SCOP. This measure of functional diversity is an underestimate because it only counts the enzyme functions and only those given an EC number. Figure 3 and Figure 4 show the expected trend that the most functionally diverse are more likely to have a lower G-score, but there are still many groups with highly significant G-scores and significant functional diversity. Among these groups are the well known enolase (Babbitt et al. 1996), haloacid dehalogenase (Allen et al. 2004), and amidohydrolase superfamilies (Holm et al. 1997). The remainder are also good candidates for superfamilies that have evolved according to a similar evolutionary model.

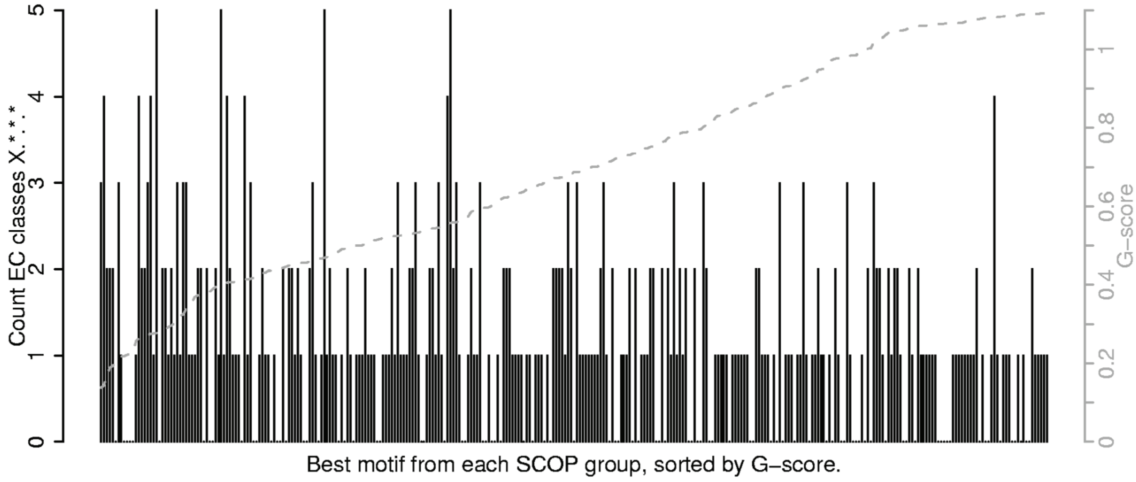


Figure 3. Number of distinct EC classes at first position in each SCOP group. Each vertical bar shows the count of distinct EC codes, only counting the first position, in a SCOP group (superfamilies and families). The SCOP groups are sorted along the x axis by the G-score of their best motif. G-scores are indicated by the dashed gray line.

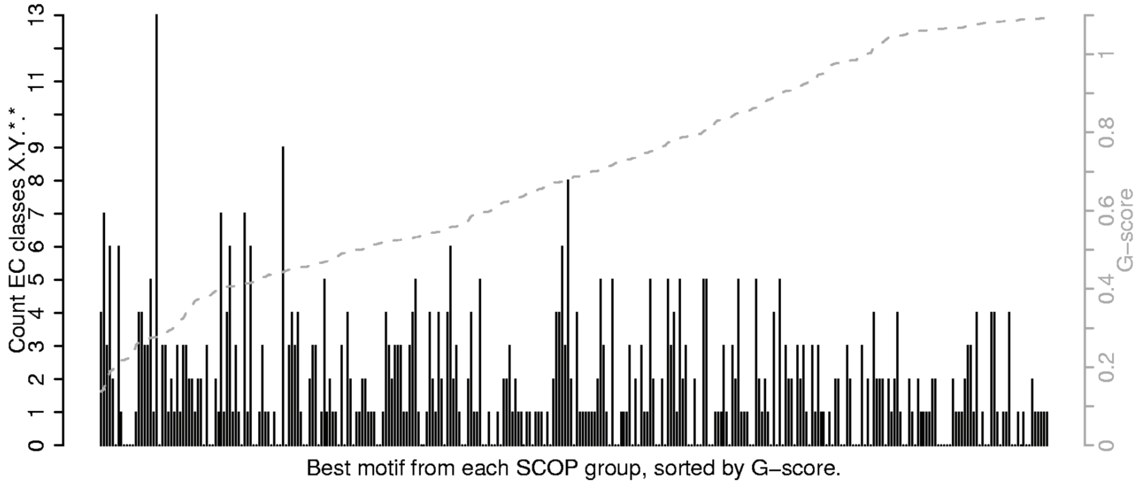


Figure 4. Number of distinct EC classes at first two positions in each SCOP group. Each vertical bar shows the count of distinct EC codes, only counting the first two positions, in a SCOP group (superfamilies and families). Remaining is as in Figure 3.

Patterns of conservation in 3D

The motifs generated by GASPS are chosen for both their conservation in 3D space and their uniqueness, or lack of matches among unrelated protein structures. It appears from this study, that repeated patterns between isofunctional but unrelated proteins are rare. Instead, most well conserved 3D structural patterns of more than two residues are the results of homology and are unlikely to be repeated in a non-homologous group (two residue motifs such as disulfide bridges are frequently conserved but not unique). While the results of chapter 3 indicate the opposite, that some sensitive motifs are not specific to a group, these motifs can be made more specific by the addition of one or two local residues without significantly changing the composition of the motif. Uniqueness therefore plays less of a role, so that GASPS motifs across broad protein groups describe primarily the patterns of conservation in 3D space.

It is well recognized that functionally significant residues are well conserved in both sequence and structure. It follows then that we can expect a large number of motifs generated by GASPS to contain residues that are known to be functional. Using the Catalytic Site Atlas (CSA) as an independent source of functional residue information, we do see significant overlap with GASPS motifs and CSA entries. In fact, 63% of protein groups in this study with representatives in the CSA have a motif with one or more residues directly involved in catalysis. As the name implies, the CSA limits its scope to enzymes and uses a strict definition of residues directly involved in catalysis, so that many residues involved in stabilizing or binding in a functional site are not included. Important binding sites from non-enzymes, such as the iron binding site in the ferritin superfamily, or the heme binding sites in globins, as well as binding sites for metals or other cofactors involved in catalysis (such as the metal binding sites in the enolase and amidohydrolase superfamilies) occur with high frequency among the GASPS motifs (Figure 5). We identified motif residues that interact with ligands by identifying residue atoms that are within 4 Å of an atom described by a ‘HETATM’ record. Nearly half of the highest scoring motifs are associated with a ligand. The number is lower for the lower scoring motifs, confirming that G-scores correlate with functional significance, and that the association with ligands is not due to random sampling of protein residues. Approximately 1/4 of all motif ligand interactions are to metal ligands. This number goes as high as 1/3 for the highest scoring motifs revealing that metal binding sites are among the best conserved structural features, and the most reliable to match by structural motifs. It is important to keep in mind that these computed frequencies of ligand interactions by GASPS motifs necessarily underestimate the actual number of motif residues that interact with a ligand biologically. The structures may not have been solved in the presence of a ligand (only two thirds of PDB files used in this study included any HETATM record), and if present, the ligands may have been unresolved. Furthermore, any static description of structure cannot fully represent the dynamic range of biologically relevant protein motions.

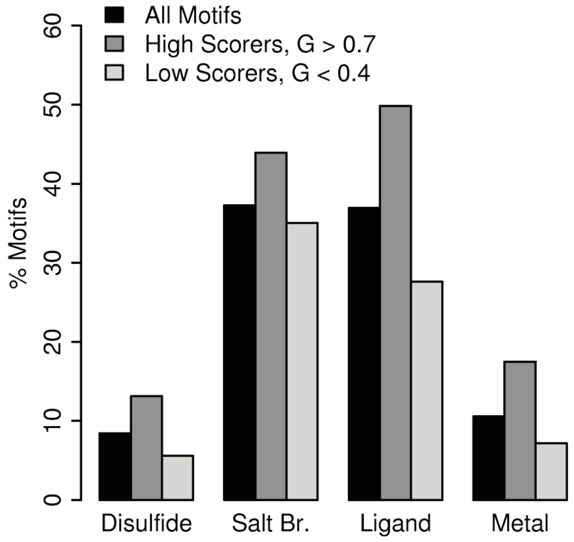


Figure 5. Residue interactions captured by motifs.

Another large group of features found in GASPS motifs are stabilizing residue interactions such as salt bridges or disulfide bridges. Salt bridges were identified by finding acidic (glutamate or aspartate) and basic (histidine, lysine, or arginine) sidechain atoms (O and N) within 4.0 Å. Disulfide bridges were identified by finding cysteine S atoms within 3 Å of each other. While a single such pair of residues is not unique to any group, when paired up with other neighboring conserved residues or other pairs of stabilizing residues they often become useful identifiers. Only 8% of all motifs include at least one cysteine involved in a disulfide bridge, however, the presence of a disulfide bridge is highly correlated with G-score. The motifs with the highest G-scores are twice as likely to contain at least one disulfide partner (13%) as those with the lowest G-scores (6%). Almost as common as the previously discussed ligand interactions are stabilized charged residues or salt bridges. These are also correlated with G-scores, though to a lesser degree.

Residue types in motifs

Other studies with 3D motifs often focus on discovering functional sites, and therefore limit their analysis to those polar and charged residues assumed to be most likely functional. GASPS makes no such distinctions, so that any residue can be included in a motif provided it is conserved in 3D space in relation to other conserved residues. Our approach allows us to ask which residues dominate the motifs, tend to be the most conserved, and provide the most classification information. Indirectly this can tell us how critical to protein function is each residue's unique role. Residue prevalence in motifs was normalized by group size and motif size, so that larger groups or larger motifs would not bias the results. We also split the motifs by G-score to observe trends as classification ability increases. In detail, for each G-score range and residue, the normalized residue frequency for any residue type (f_t) was calculated as:

$$f_t = \frac{1}{n_g} \sum_{groups} \frac{1}{n_m} \sum_{motifs} \frac{n_t}{n_r}$$

where n_t is the number of residues of type t in a motif, n_r is the number of total residues in a motif, n_m is the total number of motifs in a group, and n_g is the total number of groups.

The distributions of dominant amino acids in the highest scoring motifs shows some patterns that are expected from the previous discussion of common structural features, but also indicate that we have not yet described all important structural trends captured by motifs (Figure 6). The presence of cysteine, aspartate, histidine and glutamate among the top seven amino acids are expected from the previous discussion of catalytic sites, ligand–especially metal–sites, and salt bridges. Less expected is the dominance of glycine, and prominent role of leucine. The prevalence of cysteines is not surprising given their unique role in disulfide bridges, catalytic sites and metal binding sites, all of which are well represented among motifs. Likewise, histidine and aspartate frequently play a role in metal binding sites and catalytic sites. The prevalence of leucine among motifs, on the other hand, may simply be a result of the high overall frequency of leucine in the entire proteins. The frequency of a residue type among the motifs is compared in Figure 6 to its frequency among the entire set of residues allowed by GASPS, the background frequency. The frequency of leucine among the high scoring motifs is actually reduced compared to this background frequency. Glycine also has a high background frequency, second only to leucine, but its prevalence among the high-scoring motifs is increased over this background. What accounts for glycine's high prevalence? While it has no sidechain to interact with ligands, a relatively high proportion of the glycines in motifs are within interaction distances of ligands. In fact, glycines in motifs rank fourth behind only histidine, cysteine and aspartate for their rate of interaction with ligands. Most ligand-interacting glycines interact with phosphate containing compounds, the most common being FAD, NAD and ADP. A smaller number of glycines interact with sulfur containing compounds, mostly sulfate. Most inter-atom (non hydrogen) distances for interactions with glycines are between 3 and 4 Å (median=3.25 Å), so that many are outside of the range of a typical hydrogen bond. Instead, the presence of glycine in binding pockets may provide for the tight bending of loops around ligands as well as space for a ligand to bind (Jornvall et al. 1984; Dym et al. 2001). Similarly, proline's unique backbone angles may account for its seventh highest frequency among high-scoring motifs. While the majority of prolines as well as many glycines are not within interaction distance of ligands, they are often near residues that do interact. Still, not all of these conserved glycines and prolines are near known functional sites in motifs, and likely serve to stabilize an overall fold rather than the fine-scale geometries in a catalytic or binding site. Examples of these glycines can be found in motifs from the enolase superfamily (SCOP c.1.11; 2mnr, Gly291), and amylase families (SCOP c.1.8.1; 1esw, chain a, Gly40).

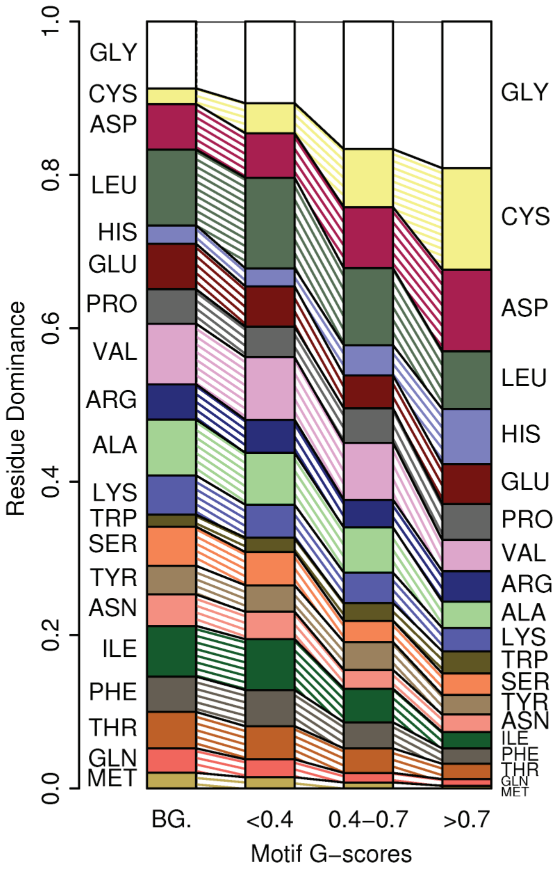


Figure 6. Dominance of residue types, compared against background residue frequency, and at different G-scores.
See text for computations of residue dominance that is shown here calculated for motifs pooled to three different groups by G-score. The column labeled BG. for background refers to the frequency of the residue type among all residues considered by GASPS for inclusion in any motif.

Annotation of protein structures

Beyond a survey of structural conservation among diverse protein groups, the outcome of this study provides for a set of motifs that can be used to help annotate novel protein structures. We have packaged the motifs generated in this study together with structure matching and browsing tools as a web resource named GASPSdb (http://gaspsdb.rbvi.ucsf.edu). We demonstrate the benefit of GASPSdb by showing that its search results provide additional coverage with similar or better accuracy compared to other available libraries of 3D motifs, CSA and PINTS described earlier. Between SCOP version 1.65 (used by GASPSdb) and SCOP version 1.69, an additional 1612 domains were added that were less than 40% identical to each other or earlier domains. GASPSdb contains superfamily motifs for about half of these new domains (790 domains, 49%). For the comparison, we used these 790 new domains as queries and for each search, we considered only the most significant match. Figure 7 shows how often these first matches identified a motif from the correct superfamily, and how often the first matches with significant scores ($E < 0.001$) were true. No difference was found in the rate of matching structures labeled as “putative” in their “Structure Title” or “Structure Description” fields, suggesting that motifs perform as well on structures for which we have little prior knowledge.

By identifying functional sites and functional residues, the fine-scale information of 3D motifs from GASPSdb provides functional details that homology detection methods that use an entire protein such as PSI-BLAST(Altschul et al. 1997) cannot provide. On the other hand, PSI-BLAST is able to give accurate homology predictions for more proteins than 3D motifs generated by any method. Working together though, GASPSdb can extend the annotation power of PSI-BLAST by corroborating low-significance PSI-BLAST hits. On the same set of 790 domains from superfamilies with motifs in GASPSdb, 83% of PSI-BLAST searches return a sequence with the correct superfamily as the most significant match. However, the effectiveness of 3D motifs is mostly independent from sequence similarity, so that 3D motifs can complement homology searches. Considering only those proteins above where PSI-BLAST yields an ambiguous result (first match $E > 1e\text{-}3$), only 57 of 184 PSI-BLAST first matches yield a true hit. A

GASPSdb RIGOR search can corroborate the true match 19 times (33%), not significantly different (chi-squared test, p = 0.46) from its performance on the larger set (see Figure 7), and with only 4 false positives at E<0.001.

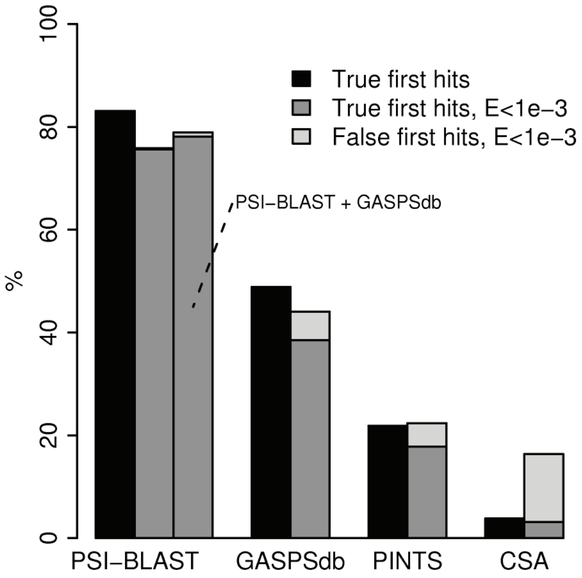


Figure 7. Coverage of GASPSdb compared to other 3D motif libraries and PSI-BLAST.
The percentage on the y-axis is the number of structures giving true or false positives as their first hit when searched against the sequence or motif database on the x-axis.

Homology models

A tool to annotate protein structures would be most useful if it worked on low quality structures, such as homology models generated from distant templates (e.g., less than 30% sequence identical), as well as structures solved empirically. We tested the performance of GASPSdb together with the homology-modeling tool MODELLER (Sali et al. 1993) used to model structures for the 790 new domains in SCOP 1.69 discussed previously. We generated models using only the new domain's sequence and an existing structure from the same superfamily in SCOP 1.65 as a template. To simulate the conditions of low-quality models, we required template structures to match the sequence with a PSI-BLAST E-value worse than 10^{-5}. The sequence-template pairs were automatically aligned and modeled using MODELLER with default settings and its built-in align2d program. Of the 618 resulting model-able pairs, 88 (14%) of the homology models were correctly annotated by GASPSdb motifs, while only 277 (45%) of the actual crystal structures were correctly annotated by any GASPSdb motif. These homology models are of sufficient quality to match a motif 32% (88/277) as often as their actual crystal structure. In another experiment, by purposely choosing a falsely homologous decoy template at similar PSI-BLAST E-values, we find that the rate of false positive motif matches to the decoy superfamily is only 0.005. Accuracy of homology models is known to be highly dependent on alignment accuracy (Martin et al. 1997; Venclovas et al. 2005), and this experiment included the simplest alignment protocol, with many expected alignment errors. Nevertheless, these low quality homology models are accurate enough to match the appropriate motif 32% (88 of 277) as often as their crystal structure. When applied to the large number of unknown sequences in available databases, even this relatively low proportion could prove useful.

The success of GASPS motifs on homology models is not due simply to the overall accuracy of homology modeling, but the tendency for GASPS to choose motifs that are accurately modeled. Among a set of homology models made for structures in GASPSdb, approximately 80% of the randomly generated motifs (those chosen before the first round of GASPS optimization) match with less significant E-values than the final GASPS-optimized motifs.

Discussion

Chosen only for their sensitivity and specificity, yet with frequent overlaps with functional sites, motifs presented in the GASPSdb make a useful tool to describe function and highlight likely functional residues of novel protein structures. These motifs are chosen for their ability to identify a group of proteins. Any protein that matches the motif is expected to share the same function, at least to the extent that function is shared among the original group that produced the motif. An alternative approach that does not require a mapping between motifs and specific functions is to find any significant similarity in local 3D structure between two proteins (Oldfield 2002; Laskowski et al. 2005). One challenge of this technique is the high number of false positives produced by these randomly chosen motifs. The method of Laskowski et al.(Laskowski et al. 2005), to further filter matches based on similarity of residues in the local structure of the motif source and its match, can effectively weed out the false positives and identify true homology between protein structures. While generally useful, these pairwise relationships do not identify to what extent the functions of two proteins is similar. The motifs generated here, like other motifs designed to identify specific groups of proteins, such as EC numbers (Torrance et al. 2005), can provide a useful complement to this pairwise motif technique.

In addition to this immediate practical application, taken together, the motifs generated here provide a view into the trends in evolutionary constraints on function. As discussed earlier, the high-scoring motifs provide evidence of evolutionary constraints. It is not surprising, then, that the high scoring motifs are mostly restricted to homologous groups. While cases of convergent evolution exist that can be described by a structural motif, these cases are rare, and any common motif shared by two convergent groups may be washed out by additional isofunctional but independently evolved groups that do not share the motif. It is notable, though, that while the literature provides cases of diverse superfamilies and families matching a single motif, most groups with sufficient evolutionary distance do not share a single motif as defined here. This does not necessarily imply that the 3D motif approach to annotating function and identifying functional residues has limited application. While a single motif may not exist, subgroups within the larger group can share a common motif. Instead, it has more bearing on how function constrains protein structure. When function remains the same despite the lack of a shared motif, the diverse proteins must discover a new way of completing the function, or at least make do with a different set of residues. When these proteins evolve to perform new functions they do so without maintaining a superfamily-conserved set of residues. It is worth noting that the current study requires identical residue types. In many cases, while a recognized active site architecture is maintained, the roles such as acidic or basic side-chain, can be adequately performed by multiple residue types.

We have described common trends for evolutionary constraints on residue type and simple residue interactions. One interpretation is that the strongest constraints are on those residue types that perform unique roles: cysteines provide disulfides, histidines provide labile acid-base chemistry, glycines and prolines provide for unique backbone angles, and glycines' missing sidechains maximize available space. On the other hand, 3D motif methods require residues to be relatively unmoved across various proteins *and* the various experimental conditions used to solve the structures. Therefore, GASPS should favor features that confine sidechains to a specific position. As examples, metal ions bind their ligands very tightly. Together with their functional importance, this explains why they were common features among motifs. This trend is observed at the residue level as well. The residues that frequently coordinate metal ions, cysteine, histidine, glutamate and aspartate, are among the most frequent. While this effect is not necessarily restricted to metal ligands, they seem to have the strongest effect. Both lysine and arginine in the motifs bind ligands (though not metals) with similar or greater rates as the negative charged and metal-binding aspartate and glutamate. However, the positive-charged residues rank several places below both negatively charged residues in their overall prevalence in motifs.

While this study provides one view on constraints on protein evolution, it is a view that is limited by the mechanics of GASPS. GASPS identifies just the strongest motif or constraint on each run. While repeated runs may reveal less well-conserved motifs, these secondary motifs can easily be missed, so that GASPS often cannot reveal all constraints on a protein group, but only the strongest. Similarly, GASPS looks for motifs that match all proteins in a group, and will give lower scores to those motifs that match only a fraction of the structures even at very high significance. Some of the moderately low G-scores seen here may simply represent a very well conserved motif, but only among members of a subgroup of the larger group. However, the trends for residue distributions of lower-scoring motifs to more closely match background distributions implies that many of the low scoring motifs are at least partially due to simple chance. As mentioned previously, GASPS requires identical residues at each position. It does not even consider conservative substitutions such as aspartate/glutamate. We previously examined modifying GASPS to allow for position-specific substitutions, but found little improvement when applied to groups with already moderately high-scoring motifs. Whether this substitution scheme allows for motifs with improved G-scores for the low scoring groups here is an open question.

GASPS requires sufficient diversity in available protein structures to weed out motifs that exist by chance or recent shared ancestry instead of by their importance to a shared function. As more structures are made available, the range of GASPS' effectiveness will extend to more groups. An automated method like GASPS can easily keep abreast of the latest developments.

References

Allen, K. N. and D. Dunaway-Mariano (2004). "Phosphoryl group transfer: evolution of a catalytic scaffold." Trends Biochem Sci **29**(9): 495-503.

Altschul, S. F., T. L. Madden, et al. (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res **25**(17): 3389-402.

Artymiuk, P. J., A. R. Poirrette, et al. (1994). "A graph-theoretic approach to the identification of three-dimensional patterns of amino acid side-chains in protein structures." J Mol Biol **243**(2): 327-44.

Ashburner, M., C. A. Ball, et al. (2000). "Gene ontology: tool for the unification of biology. The Gene Ontology Consortium." Nat Genet **25**(1): 25-9.

Babbitt, P. C., M. S. Hasson, et al. (1996). "The enolase superfamily: a general strategy for enzyme-catalyzed abstraction of the alpha-protons of carboxylic acids." Biochemistry **35**(51): 16489-501.

Barker, J. A. and J. M. Thornton (2003). "An algorithm for constraint-based structural template matching: application to 3D templates with statistical analysis." Bioinformatics **19**(13): 1644-9.

Dym, O. and D. Eisenberg (2001). "Sequence-structure analysis of FAD-containing proteins." Protein Sci **10**(9): 1712-28.

Fetrow, J. S. and J. Skolnick (1998). "Method for prediction of protein function from sequence using the sequence-to-structure-to-function paradigm with application to glutaredoxins/thioredoxins and T1 ribonucleases." J Mol Biol **281**(5): 949-68.

Gerlt, J. A. and P. C. Babbitt (2001). "Divergent evolution of enzymatic function: mechanistically diverse superfamilies and functionally distinct suprafamilies." Annu Rev Biochem **70**: 209-46.

Holm, L. and C. Sander (1997). "An evolutionary treasure: unification of a broad set of amidohydrolases related to urease." Proteins **28**(1): 72-82.

International Union of Biochemistry and Molecular Biology. Nomenclature Committee. and E. C. Webb (1992). Enzyme nomenclature 1992: recommendations of the Nomenclature Committee of the International Union of Biochemistry and Molecular Biology on the nomenclature and classification of enzymes. San Diego, Published for the International Union of Biochemistry and Molecular Biology by Academic Press.

Jornvall, H., H. von Bahr-Lindstrom, et al. (1984). "Extensive variations and basic features in the alcohol dehydrogenase-sorbitol dehydrogenase family." Eur J Biochem **140**(1): 17-23.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Mol Biol **285**(4): 1887-97.

Laskowski, R. A., J. D. Watson, et al. (2005). "Protein function prediction using local 3D templates." J Mol Biol **351**(3): 614-26.

Martin, A. C., M. W. MacArthur, et al. (1997). "Assessment of comparative modeling in CASP2." Proteins **Suppl 1**: 14-28.

Meng, E. C., B. J. Polacco, et al. (2004). "Superfamily active site templates." Proteins **55**(4): 962-76.

Murzin, A. G., S. E. Brenner, et al. (1995). "SCOP: a structural classification of proteins database for the investigation of sequences and structures." J Mol Biol **247**(4): 536-40.

Oldfield, T. J. (2002). "Data mining the protein data bank: residue interactions." Proteins **49**(4): 510-28.

Polacco, B. J. and P. C. Babbitt (2006). "Automated discovery of 3D motifs for protein function annotation." Bioinformatics **22**(6): 723-30.

Porter, C. T., G. J. Bartlett, et al. (2004). "The Catalytic Site Atlas: a resource of catalytic sites and residues identified in enzymes using structural data." Nucleic Acids Res **32 Database issue**: D129-33.

Sali, A. and T. L. Blundell (1993). "Comparative protein modelling by satisfaction of spatial restraints." J Mol Biol **234**(3): 779-815.

Stark, A. and R. B. Russell (2003). "Annotation in three dimensions. PINTS: Patterns in Non-homologous Tertiary Structures." Nucleic Acids Res **31**(13): 3341-4.

Stark, A., S. Sunyaev, et al. (2003). "A model for statistical significance of local similarities in structure." J Mol Biol **326**(5): 1307-16.

Torrance, J. W., G. J. Bartlett, et al. (2005). "Using a Library of Structural Templates to Recognise Catalytic Sites and Explore their Evolution in Homologous Families." J Mol Biol **347**(3): 565-81.

Venclovas, C. and M. Margelevicius (2005). "Comparative modeling in CASP6 using consensus approach to template selection, sequence-structure alignment, and structure assessment." Proteins **61 Suppl 7**: 99-105.

Watson, J. D., S. Sanderson, et al. (2007). "Towards fully automated structure-based function prediction in structural genomics: a case study." J Mol Biol **367**(5): 1511-22.

Conclusion

This work has focused on signature 3D motifs. These motifs are conserved within a group of proteins and can be used to identify the group. Through the development and application of GASPS, I have identified a large number of these signature 3D motifs that represent conserved functional components within protein structures. I have shown that these 3D motifs can be used to help annotate the functions of proteins but also that signature motifs are not a universally useful tool that can be used for all groups of proteins we may wish to classify. Many groups examined here provided no useful motif. In many cases, this can be attributed to the classification system used to define any particular group. While the group defined by the classification may not all share a motif, a natural sub-group may have. It may be that no perfect classification exists. The accepted classification of biological entities at any level is constantly changing. As long as there are new genomes to sequence, there will likely always be newly discovered protein structures that do not faithfully match any previously discovered group motif.

The distribution of these 3D motifs and the patterns of residues within them also describe how local protein structure evolves. While we see a strong relationship between conserved elements and a protein’s function, a protein’s evolutionary history appears more important for determining local structure than just function alone. We see very few cases of convergent evolution here, where a single function has required the same set of residues in unrelated proteins. More often, we see that a 3D motif and the elements of a protein’s function that the motif represents have been adopted from an ancestral function, even when the overall function has evolved to be different. This phenomenon is identified here by homologous groups of proteins with diverse functions and well-conserved 3D motifs that are directly involved in a protein's function.

As more structures become available, I expect the automated method of GASPS can be used to generate motifs for protein groups that did not have enough structures to be included in this study. The motifs available in GASPSdb can therefore be a constantly growing resource available to the biological community. While I present some work here evaluating and discarding alternative techniques for GASPS, I expect that GASPS could be improved by other methods. GASPS was built to require no similarity in folds so that motifs could be detected even when overall folds could not be aligned. One of the conclusions of this work, however, is that cases of motifs across different folds are very rare. With this knowledge, a faster method could use a structural alignment to first identify conserved regions that can identify the group with high sensitivity, and, if necessary, adjustments can be made to ensure specificity. Such a system may be better able to deal with the added degrees of freedom provided by a method that allows substitutions at specific positions. Another alternative is to build motifs from single atoms, chemical groups, or physical and chemical descriptors instead of residues.

In many ways, the development of this work parallels both the evolution of protein structure and function, as well as the genetic algorithm that provided the majority of the results for my work. All three systems make use of fortuitous occurrences with trial and error, leverage existing resources, and culminate in a successful product. The main fortuitous occurrence (there are plenty of others that were less successful) that contributed to this dissertation was the opportunity to collaborate with the initial superfamily active site templates project, described in Chapter 1. Just as today's proteins recruited functional components that originally may have provided a different overall function, GASPS makes use of many Python functions written for the work of Chapter 1. In fact, my using a genetic algorithm as opposed to other techniques for building motifs, is a result of its development history, when other techniques may be equally, if not more, appropriate (see above). Finally, while I suggest above that the parallel includes a successful end-product of my work, I leave the evaluation of the success of this dissertation to the scoring or fitness functions that my readers bring with them.

Appendix 1: GASPS Package

This appendix contains the text files included in the GASPS software package. This package was distributed as a gzipped, tar-formatted archive to other researchers requesting the source code for GASPS.

ReadMe

```
             ____  _____    _    ____  __  __ _____
            |  _ \| ____|  / \  |  _ \|  \/  | ____|
            | |_) |  _|   / _ \ | | | | |\/| |  _|
            |  _ <| |___ / ___ \| |_| | |  | | |___
            |_| \_\_____/_/   \_\____/|_|  |_|_____|



This package should contain all the python code that is necessary to run GASPS.  In addition you will need the motif searching tool SPASM and MKSPAZ to generate your own libraries.  Together with their manuals, these can be downloaded from:

http://alpha2.bmc.uu.se/usf/spasm.html

Just in case we need to say it: We are not responsible, nor hold any ownership for any of the SPASM and MKSPAZ software available from the above site.

GASPS.py can construct a multiple sequence alignment by running PSI-BLAST if it is so instructed. To use this you will need a copy of the blastpgp program and sequence database available from NCBI:

http://www.ncbi.nlm.nih.gov/BLAST/download.shtml

Again: We are not responsible, nor hold any ownership for any of the NCBI software.

Minimally, once SPASM is installed you are ready to go. GASPS can be controlled via many different command line arguments.  The following is a typical command and makes a reasonable test of your installation. It should run from the GASPS_package directory with the files located in test/.  Any errors should result in a failed execution within the first few minutes.  A sequence of happy SPASM messages to stderr "..Toodle Pip.." indicates that things are probably running properly. Running to completion may take a few hours or more depending on your system's speed.  As configured (--writeTables=1), it will probably use about 30MB or more to store the output of all its SPASM runs, turn this off (--writeTables=0) if you're tight on disk space.


```

```
[~/GASPS_package] % python GASPS.py --pdbFile=test/d2mnr_1.pdb --chain=' ' --filesPath=2mnr.test --tpFile=test/enolase.list --trueLibrary=test/enolase.lib --lengthTrueLibrary=7 --doNotCountQuery=d2mnr_1 --falseLibrary=test/astral_1.65_SF.lib --alignFile=test/d2mnr_1.fasta.psiblast.xml.faln --refRowName=d2mnr_1 --writeTables=1 --useFileNames


For more information on the purpose of the specific command line arguments,  read the GASPS.py file.  Most useful output will appear in 2mnr.test_log.txt, and the files used to run SPASM will all be in 2mnr.test/*. The log.txt file will contain at least one line for each motif attempted and its GASPS score. The winning motif can be located near the end of the file.  Additionally, the residues that appear frequently among the top scoring motifs are given a score that is simply the number of top scoring motifs they appear in. When these residues are described, the potential list of substitutions are listed at each of these positions regardless of wether substitutions were turned on (--noSubs=0).

Running GASPS from any other directory may require configuring your python environment to find my modules located in GASPS_package/polacco/.  If you need help and don't know where to look, try a web search for PYTHONPATH.  For example:

http://www.google.com/search?q=pythonpath.

Additionally, if the spasm and blastpgp binaries (or databases) are not in your shell's search paths, you may have to modify the following lines in GASPS.py to point to the absolute paths of these files:

__spasmBinaryPath = "spasm"
__blastpgpPath = "blastpgp"
__blastDB = "nrdb90"



The files in test/ are typical files that GASPS depends on for a typical run:
files in test/


                     d2mnr_1.fasta Sequence file corresponding to
                                   2mnr.pdb. use:
                                   --generateAlign=d2mnr_1.fasta

    d2mnr_1.fasta.psiblast.xml.faln Multiple sequence alignment already
                                   generated by GASPS.  use:
                                   --align=d2mnr_1.fasta.psiblast.xml.faln

                       d2mnr_1.pdb Structure file as a source of motif
                                   coordinates. use: --pdbFile=d2mnr_1.pdb

```

```

                 astral_1.65_SF.lib A short MKSPAZ-formatted library of
                                    structures, one per SCOP
                                    superfamily. use:
                                    --falseLibrary=astral_1.65_SF.lib

                        enolase.lib A library of 7 representative
                                    structures from the enolase
                                    superfamily. use:
                                    --trueLibrary=enolase.lib

                       enolase.list A list of all structures in the
                                    enolase family. These will be
                                    excluded from 'falseLibrary' to
                                    result in a library where all hits
                                    will truly be false positives. use:
                                     --tpFile=enolase.list




This software (GASPS.py and polacco/*.py) was developed for ongoing research purposes, and mostly my own private use. To make things a bit less confused, I attempted to remove those sections of code that were not directly of use to a functioning GASPS.py, but I easily could have broken something that I did not have the time to test. Should anything not work as you expect, please let me know.

If you use GASPS in your work, please cite:

Polacco BJ, Babbitt PC. Automated discovery of 3D motifs for protein function annotation. Bioinformatics. 2006 Mar 15;22(6):723-30.

We are pretty certain this software does no harm to your system under typical usage. However, we offer no warranties of any kind and so can not be held accountable if it does.

Any comments, questions, suggestions, complaints can be directed my way by email:
polacco@cgl.ucsf.edu

-Ben Polacco
March 30, 2006

```

GASPS.py

```
#! /sw/bin/python

#
#   ____    _    ____  ____  ____
#  / ___|  / \  / ___||  _ \/ ___|  _ __  _   _
```

```
# | |  _  / _ \ \___ \| |_) \___ \ | '_ \| | | |
# | |_| |/ ___ \ ___) |  __/ ___) || |_) | |_| |
#  \____/_/   \_\____/|_|   |____(_) .__/ \__, |
#                                  |_|    |___/
#
#
#
#
# There are many, many options that can be set from the command line.  This results
# in some very long commands.  The following is a typical basic command.
#
# GASPS.py --pdbFile=test/2mnr.pdb --chain=' ' --filesPath=2mnr.test --tpFile=test/enolase_1.65All.list --trueLibrary=test/enolase.lib --lengthTrueLibrary=7 --doNotCountQuery=2mnr --falseLibrary=test/astral_1.65_noMutants_SF_pid0.lib --alignFile=test/2mnr.fasta.psiblast.xml.faln --refRowName=2MNR:_ --writeTables=1
#
# Those arguments explained here:
#
#  --pdbFile=2mnr.pdb                                PDB file to pull coordinates from
#  --chain=' '                                       Which chain in PDB file to use. (A, B, C, ' ' )
#  --filesPath=2mnr.test                             Directory name where most output files should go.
#  --tpFile=enolase_1.65All.list                     List of true positives that should be ignored from the so-called falseLibrary.
#  --trueLibrary=enolase.lib                         Spasm library containing only true positive structures.
#  --lengthTrueLibrary=7                             Number of structures in lengthTrueLibrary; e.g., grep -c PDB enolase.lib
#  --doNotCountQuery=2mnr                            Name of query structure, don't count it towards computing GASPS scores.
#  --falseLibrary=astral_1.65_noMutants_SF_pid0.lib    Spasm library containing the background structures, may include true positives, use
#                                          tpFile to list those that should be ignored in this file.
#  --alignFile=2mnr.fasta.psiblast.xml.faln          Multiple sequence alignment, must contain a sequence corresponding to pdbFile
#  --refRowName=2MNR:_                               Name in multiple sequence alignment of corresponding (pdbFile) sequence
#  --writeTables=1                                   Boolean, should GASPS write out meaningful files showing matches for each motif.
#
#
# To get an idea of other options use: GASPS --help
# To get an idea of what these options control, examine comments in function SetDefaults
# The existence of some of the more puzzling options may simply be a side-effect of my development process.

import string, random, os, os.path, sys, time, math, getopt, copy

```

```
#
# May have to move polacco/*.py to a directory listed in PYTHONPATH
# or modify PYTHONPATH to include parent directory of polacco/*.py
import polacco.Spasm, polacco.MultiAlign, polacco.utils
#

# Trouble locating these, you may have to use absolute path here.
__spasmBinaryPath = "spasm"
__blastpgpPath = "blastpgp"
__blastDB = "nrdb90"


def DescribeMembers (info, openFile):
   kees = info.__dict__.keys()
   kees.sort()
   for key in kees:
      openFile.write( "%30s\t:\t%s\n" % (key, info.__dict__[key]) )
   openFile.flush()

def FileExists (filePath):
   try:
      fp = open (filePath)
      fp.close()
      return 1
   except IOError:
      return 0

def PatternSampled (info, subset):
   directory = os.path.join (info.filesPath, string.join (subset,
"_"))
   doneFile = os.path.join (directory, "spasm.table")
   return FileExists (doneFile) or FileExists (doneFile + ".gz")

def WriteLog (logFile, string, newLine = 1):
   fp = open (logFile, 'a')
   fp.write (string)
   if newLine:
      fp.write ("\n")
   fp.close()

def ChooseConservationCutoff (conservationScores, numWanted,
referenceRow=None, allowedResidues=None):
   cons = []
   for i in range (len (conservationScores)):
      if referenceRow.chars[i] in allowedResidues:
         cons.append (conservationScores[i])

   if numWanted < 1.0 and numWanted > 0.0:
      # User is asking for a fraction,
      # convert it to a number of residues based on number of
allowed residues
      numWanted = numWanted * len (cons)

   numWanted = int (numWanted)
   if numWanted > len (cons):
      print "After removing gaps and unwanted residues:Only  %d
residues to choose from (wanted %d)" % (len (cons), numWanted)
      return 0.0
```

```

   cons.sort()
   cons.reverse()
   return cons[numWanted-1]


def GetAvailableResidues(pdbFile, chain=None, model=1):
   chains = {}
   fp = open (pdbFile)
   lastRes = ''
   lastChain = ''
   index = 0
   models = 0
   curModel = None
   while (1):
      line = fp.readline()
      if (line == ''):
         break


      if line[0:5] == 'MODEL':
         models+=1
         curModel = int(line[11:16])
         continue
      elif line[0:6] == 'ENDMDL':
         if curModel == model:
            break
         curModel = None
         continue
      elif line[0:4] != 'ATOM':
         continue

      if curModel and curModel != model:
         continue

      curChain = line[21]
      if (chain and curChain != chain):
         if not chain in '?*': #special cases, ? is first and * is all
            continue
      res = string.strip( line[22:27])
      atom = line[13:16]

      if (res == lastRes and curChain == lastChain):
         if locatedCA:
            continue
      else:
         locatedCA = 0
         recorded = 0

      if not recorded:
         try:
            c = chains[curChain]
         except KeyError:
            index = 0
            c = chains[curChain] = {}
         c[res] = ()
         recorded = 1
```

```

      if not locatedCA and atom == "CA ":
         locatedCA = 1
         x = float (line[32:38])
         y = float (line[40:46])
         z = float (line[48:55])
         type = line[17:20]

         chains[curChain][res] = (x,y,z,index, type)
         index += 1
      lastChain = curChain
      lastRes = res
   fp.close()
   if chain and not chain in '?*':
      return chains[chain]
   elif chain == '?':
      assert len(chains) == 1
      return chains[chains.keys()[0]]
   else:
      return chains


#simple Needleman-Wunsch to map the alignment sequence on to the structure sequence
def Needleman(s1, s2, scoreMatch=1.0, scoreMismatch=-3.0, scoreGap=-1.0):
   m = []
   for i1 in range(len(s1) + 1):
      m.append((len(s2) + 1) * [ 0.0])

   for i1 in range(len(s1)):
      for i2 in range(len(s2)):
         if s1[i1] == s2[i2]:
            best = m[i1][i2] + scoreMatch
         else:
            best = m[i1][i2] + scoreMismatch
         skip = m[i1][i2+1] + scoreGap
         if skip > best:
            best = skip
         skip = m[i1+1][i2] + scoreGap
         if skip > best:
            best = skip
         m[i1+1][i2+1] = best
   i1 = len(s1)
   i2 = len(s2)
   matchList = []
   while i1 > 0 and i2 > 0:
      best = m[i1-1][i2-1]
      action = 0     # match
      if m[i1][i2-1] > best:
         best = m[i1][i2-1]
         action = 1  # skip i2
      if m[i1-1][i2] > best:
         best = m[i1-1][i2]
         action = 2  # skip i1
      if action == 0:
         matchList.append((i1-1, i2-1))
         i1 = i1 - 1
```

```
         i2 = i2 - 1
      elif action == 1:
         i2 = i2 - 1
      else:
         i1 = i1 - 1
   return matchList


def GetAvailableConservedResidues (info, pdbFile, chain, multiAlign, referenceRow,
                                   minConservation = 0.0, allowedResidues = "FILVPAGMCWYTSQNEDHKR", numResiduesAllowed = -1):
   print "Processing multiple sequence alignment..."
   #first load all residues from the pdbFile
   allResidues = GetAvailableResidues (pdbFile, chain)
   residueNames = allResidues.keys()
   #reconstruct their order
   namesInOrder = []
   typesInOrder = []
   for i in range (len(residueNames)):
      namesInOrder.append ('?')
      typesInOrder.append ('???')

   for name in residueNames:
      #print name
      if len (allResidues[name]) < 5:
         print "Warning:  Trouble reading information from pdb for residue %s" % (name)
         WriteLog (info.logFile, "Warning:  Trouble reading information from pdb for residue %s" % (name))
         continue
      (index, type) = allResidues[name][3:5]
      try:
         namesInOrder[index] = name
      except IndexError, data:
         print data
         print index
         print namesInOrder
         print typesInOrder
         print pdbFile
         print allResidues
         sys.exit(0)
      typesInOrder[index] = type

   #align pdbSequence with referenceRow
   pdbSeq =  (polacco.utils.SeqAA3to1(typesInOrder))
   refChars, refIndexes = referenceRow.GetCharsAndIndexesNoGaps()
   matches = Needleman (pdbSeq, refChars)

   #compute conservation
   vc = polacco.MultiAlign.ValdarConservation (multiAlign)
   conservations = vc.Compute()
   if numResiduesAllowed > 0:
      minConservation = ChooseConservationCutoff (conservations, numResiduesAllowed, referenceRow, allowedResidues)
      print "Using conservation cutoff = %6.4f" % minConservation
   #get possible substitutions per position
   #subs = multiAlign.GetLettersPerColumn ()
```

```
   subs = multiAlign.GetDominantLettersPerColumn(0.1)

   #generate list of user requested residues from user requested motifs.
   #these are forced to be included (with all their substitutions!) regardless of their conservation
   userRequestedResidues = []
   for motif in info.motifs:
      for res in motif:
         if not res in userRequestedResidues:
            userRequestedResidues.append (res)

   #map conservation scores to pdbSeq and return result
   conResidues = {}
   for match in matches:
      name = namesInOrder[match[0]]
      conservation = conservations[refIndexes[match[1]]]

      if name in userRequestedResidues:
         pass
      elif not pdbSeq[match[0]] in allowedResidues:
         continue
      elif conservation < minConservation:
         continue

      if    not refChars[match[1]] in subs[refIndexes[match[1]]]:
         subs[refIndexes[match[1]]].append (refChars[match[1]])

      conResidues[name] = (allResidues[name] + (conservation, polacco.utils.SeqAA1to3(subs[refIndexes[match[1]]])))


   return conResidues

def EucDistance (a, b):
   sumSquares = 0.0
   for i in range (len (a)):
      sumSquares += math.pow(a[i]-b[i], 2)
   return math.sqrt(sumSquares)

def GetDistanceMatrix (allResidueLocations):
   mat = {}
   for res in allResidueLocations.keys():
      mat[res] = {}
      for other in allResidueLocations.keys():
         if res == other:
            continue
         mat[res][other] = EucDistance(allResidueLocations[res][0:3], allResidueLocations[other][0:3])
   return mat


def MatChooseSpatiallyCloseSubset (allResidueLocations, distanceMatrix, numResidues, maxRadius, res = None):
   numResidues = int (numResidues)
   chosen = []
   next = None
```

```
   i = 0
   resNames = allResidueLocations.keys()
   center = None
   while (i < numResidues):
      while (1):
         if not center:
            if not res:
               next = random.choice (resNames)
            else:
               next = res
            if maxRadius != 99.9:
               possibleOthers = [x for x in distanceMatrix[next].keys() if distanceMatrix[next][x] < maxRadius]
            else:
               possibleOthers = distanceMatrix[next].keys()
            if len (possibleOthers) < numResidues-1:
               if res:
                  return None
               else:
                  continue
            center = allResidueLocations[next][0:3]
         else:
            next = random.choice (possibleOthers)
            if EucDistance (center, allResidueLocations[next][0:3]) > maxRadius:
               continue
         if not next in chosen:
            break
      chosen.append (next)
      i += 1
   return chosen

#mostly for debugging:
def DescribePossibilities (distanceMatrix, cutoff, number, info):
   WriteLog (info.logFile, 'From %d residues at distance cutoff = %d requiring %d neighbors' % (len (distanceMatrix.keys()), cutoff, number))
   num = int (number) - 1
   for res in distanceMatrix.keys():
      l = len ([x for x in distanceMatrix[res].values() if x < cutoff])
      if l >= num:
         WriteLog (info.logFile, '%s %s %s' % (res, l, [x for x in distanceMatrix[res].keys() if distanceMatrix[res][x] < cutoff] ))

def GenerateMotifFile (pdbFile, chain, residues, directory, allResidues = None):
   motifPath = os.path.join (directory, "motif.pdb")
   motfp = open (motifPath, "w")
   pdbfp = open (pdbFile)
   resTypes = []
   lastres = ''
   models = 0
   for line in pdbfp:
      if line[0:5] == 'MODEL':
         models = 1
         continue
      if models and line[0:6] == 'ENDMDL':
```

```
         break
      if line[0:4] != 'ATOM':
         continue
      curChain = line[21]
      if chain == '?':
         chain = curChain
      if (curChain != chain):
         continue
      res = string.strip( line[22:27])
      if res in residues:
         motfp.write (line)
         if res != lastres:
            try:
               if allResidues and len (allResidues[res]) > 6:
                  resTypes.append (allResidues[res][6])
               else:
                  resTypes.append ((line[17:20],))
            except KeyError:
               #res not found in allResidues, must be a user supplied motif
               resTypes.append ((line[17:20],))
      lastres = res


   # add remark indicating the allowed substitutions we expect: SPASM and GASPS do not use this!
   motfp.write ("REMARK   *  For note only, spasm does not use this!\n")
   motfp.write ("REMARK   *  restypes:")
   for resType in resTypes:
      motfp.write ("  %s" % string.join (resType, "/"))
   motfp.write ("\n")
   motfp.close()
   pdbfp.close()
   #print resTypes
   return (motifPath, resTypes)



def GenerateSpasmRunFile (motifFile, resTypes, directory, library = "", runPath = 'spasm.com', outPath = 'spasm.out', info = None):
   spasmBinaryPath = __spasmBinaryPath

   maxHits = 100000 # set this arbitrarily high so that SPASM never stops early
   maxRMSD = 3.2
   maxCADiff = 5.0
   maxSCDiff = 3.8
   scOnly = 0
   if info:
      maxRMSD = info.maxRMSD
      maxCADiff = info.maxCADiff
      maxSCDiff = info.maxSCDiff
      scOnly = info.scOnly

   maxResolution = 999.9
   maxResidues = 9999

```

```
   subStrings = []
   for subList in resTypes:
      subStrings.append (string.join (subList, " "))

   substituteString = string.join (subStrings,"\n")

   fp = open (runPath, "w")
   if scOnly:
      spasmRunFileString = polacco.Spasm.runFileStringSTDOUT_scOnly
   else:
      spasmRunFileString = polacco.Spasm.runFileStringSTDOUT

   fp.write (spasmRunFileString % (
                                            spasmBinaryPath,
                                            maxHits,
                                            library,
                                            motifFile,
                                            'rand',
                                            maxRMSD,
                                            maxCADiff,
                                            maxSCDiff,
                                            maxResolution,
                                            maxResidues,
                                            5,
                                            substituteString))
   fp.close()
   os.chmod (runPath, 0755)
   return runPath


def LoadTrueHash (tpFile):
   tpHash = {}
   if tpFile:
      fp = open (tpFile)
      while (1):
         item = string.strip (fp.readline())
         if item == '':
            break
         item = item.upper()
         tpHash[item] = 1
      fp.close()
      print "Loaded %d unique identifiers from %s" %
(len(tpHash.keys()), tpFile)
   return tpHash

def SetupSpasmFiles (info, subset, allResidues):
      directory = os.path.join (info.filesPath, string.join (subset,
"_"))
      if info.scratchPath:
         spasmTrueOutFile = os.path.join (info.scratchPath,
string.join (subset, "_") + "true_spasm.out")
         spasmFalseOutFile = os.path.join (info.scratchPath,
string.join (subset, "_") + "false_spasm.out")
      else:
         spasmTrueOutFile = os.path.join (directory,
"true_spasm.out")
         spasmFalseOutFile = os.path.join (directory,
"false_spasm.out")
```

```


      spasmTableFile = os.path.join (directory, "spasm.table")
      try:
         os.makedirs (directory)
      except OSError, data:
         WriteLog (info.logFile, "Error (ignored) while generating directory: %s" % directory)
         WriteLog (info.logFile, data.strerror)
      if info.noSubs:
         (motif, resTypes) = GenerateMotifFile (info.pdbFile, info.chain, subset, directory, 0)
      else:
         (motif, resTypes) = GenerateMotifFile (info.pdbFile, info.chain, subset, directory, allResidues)
      runFileTrue = os.path.join (directory, "true_spasm.com")
      runFileFalse = os.path.join (directory, "false_spasm.com")
      GenerateSpasmRunFile (motif, resTypes, directory, info.falseLibrary, runFileFalse, spasmFalseOutFile, info)
      GenerateSpasmRunFile (motif, resTypes, directory, info.trueLibrary, runFileTrue, spasmTrueOutFile, info)
      return info, runFileFalse, runFileTrue, spasmTableFile


def DoSpasmRuns (info, runFileFalse, runFileTrue, spasmTableFile, writeOutFile = 0):
         fpFalseSpasm = os.popen ("csh %s" % runFileFalse)
         falseSearch = polacco.Spasm.SpasmSearch (1)
         falseSearch.titleFromFileName = info.useFileNames
         falseSearch.ParseSpasmHits (fpFalseSpasm)
         fpFalseSpasm.close()

         fpTrueSpasm = os.popen ("csh %s" % runFileTrue)
         trueSearch = polacco.Spasm.SpasmSearch (1)
         trueSearch.titleFromFileName = info.useFileNames
         trueSearch.ParseSpasmHits (fpTrueSpasm)
         fpTrueSpasm.close()

         returnAsString = 1
         tableFileString = polacco.Spasm.Convert2SpasmSearchesToSortedAndScoredTable (trueSearch, falseSearch,
                                           spasmTableFile, info.trueHash,info.useDistanceRmsd, returnAsString,
                                           writeOutFile, info.trueSkipHash, info.falseSkipHash)
         return tableFileString

def ScoreMotifs (info, population, allResidues, scores = {}):
   for subset in population:
      info, runFileFalse, runFileTrue, spasmTableFile = SetupSpasmFiles (info, subset, allResidues)

      WriteLog (info.logFile, "guess %3d %40s" % (info.round, string.join(subset, "_")), 0)

      if not info.testing:
```

```
         tableFileString = DoSpasmRuns(info, runFileFalse, runFileTrue, spasmTableFile, info.writeTables)
         if not tableFileString:
            WriteLog (info.logFile, " WARNING!: No table file string generated for guess %3d %40s; score set to 0.0" % (info.round, string.join(subset, "_")))
            print ( "WARNING!: No table file string generated for guess %3d %40s; score set to 0.0" % (info.round, string.join(subset, "_")))
            score = 0.0

         elif info.rocArea:
            score = polacco.Spasm.ComputeAreaFromTableFile (spasmTableFile, info.maxFalse, tableFileString, info.useDistanceRmsd)
         else:
            score = polacco.Spasm.ComputeSeparationScoreFromTableFile3 (spasmTableFile, info.maxFalse,
               info.maxRMSD, info.lengthTrueLibrary, info.sepScoreImportance, info.useDistanceRmsd, tableFileString)
      else:
         score = random.random ()
      scores[tuple(subset)] = score

      WriteLog (info.logFile,"  %8.4f" % ( score ))

   return scores

def GetResTypesFromMotifFile (motifFile):
   fp = open (motifFile, "r")
   restypes = []
   while (1):
      #"REMARK   1  restypes:"
      line = fp.readline()
      if line == '':
         break
      if line[0:6] == 'REMARK' and line[12:21] == "restypes:":
         for res in line[21:].strip().split(' '):
            res = res.strip()
            if len(res) >= 3:
               restypes.append (res.split('/'))
   fp.close()
   return restypes

def GetResTypesFromSpasmRunFile (runFile):
   resList = polacco.utils.aa3to1.keys()
   fp = open (runFile, "r")
   restypes = []
   while (1):
      line = fp.readline()
      if line == '':
         break
      words = line.strip().split (" ")
      if words[0] in resList:
         restypes.append (words)
   fp.close()
   return restypes
```

```

def TestMotif (directory, trueLibrary, falseLibrary):
   testTrueOutFile = os.path.join (directory,
   "testTrue_spasm.out")
   testFalseOutFile = os.path.join (directory,
"testFalse_spasm.out")

   testTableFile = os.path.join (directory, "test.table")
   motifFile = os.path.join (directory, "motif.pdb")

   #load allowed residue types from motifFile
   resTypes = GetResTypesFromMotifFile (motifFile)
   if not resTypes:
      print "WARNING!  restypes not found in motif file!"
      resTypes = GetResTypesFromSpasmRunFile (os.path.join
(directory, "false_spasm.com"))
      if not resTypes:
         print "Failed again reading from run file!"
   print resTypes
   runFileFalse = os.path.join (directory, "testFalse_spasm.com")
   runFileTrue = os.path.join (directory, "testTrue_spasm.com")

   GenerateSpasmRunFile (motifFile, resTypes, directory,
falseLibrary, runFileFalse, testFalseOutFile)
   GenerateSpasmRunFile (motifFile, resTypes, directory,
trueLibrary, runTrueFalse, testTrueOutFile)
   print runFileFalse
   os.spawnlp (os.P_WAIT, "sh", "sh", runFileFalse)
   print runFileTrue
   os.spawnlp (os.P_WAIT, "sh", "sh", runFileTrue)

   polacco.Spasm.Convert2SpasmFilesToSortedAndScoredTable
(testTrueOutFile, testFalseOutFile, testTableFile)
   return testTableFile


def IsSuperSet (super, other):
   for item in other:
      if not item in super:
         return 0
   else:
      return 1

def RemoveSameScoringSupersets (info, motifScores, motifHash,
upForRemoval):
   toRemove = []
   for mot in upForRemoval:
      score = motifScores[mot]
      otherMots = motifHash[score]
      for otherMot in otherMots:
         if otherMot == mot:
            continue
         if IsSuperSet (mot, otherMot):
            WriteLog (info.logFile, "Removing the same scoring
superset: %s (%s : %d)" % (mot, otherMot, score))
            toRemove.append (mot)
            break
   for mot in toRemove:
```

```
      score = motifScores[mot]
      del (motifScores[mot])
      motifHash[score].remove (mot)
      if len (motifHash[score]) == 0:
         del (motifHash[score])

def GetTopScorers (info, number, motifScores, previousTop):
   topScorers = {}
   motHash = {}
   #reverse the score hash:
   for mot in motifScores.keys():
      try:
         motHash[motifScores[mot]].append(mot)
      except KeyError:
         motHash[motifScores[mot]] = [mot]

   RemoveSameScoringSupersets (info, motifScores, motHash, previousTop.keys())

   scores = motHash.keys()
   scores.sort()
   scores.reverse()

   for score in scores:
      if score == 0.0:
         break
      motifs = motHash[score]
      if len(topScorers) >= number:
         break
      if len(topScorers)==0 or len(topScorers)+len(motifs)<=number*2:
         for mot in motifs:
            topScorers[mot] = score
      else:
         break
   return topScorers

def MakeRandomGuessesWithCoverage (info,distanceMatrix, allResidues, numGuesses, population = []):
   if numGuesses==0:
      return population
   allRes = allResidues.keys()
   lenAllRes = len(allRes)
   if lenAllRes > numGuesses:
      skip = lenAllRes/numGuesses
   else:
      skip = 1

   for i in range (0, lenAllRes, skip):
      res = allRes[i]
      subset = MatChooseSpatiallyCloseSubset (allResidues, distanceMatrix, info.numResidues, info.maxNeighborhood, res)
      if not subset:
         continue
      subset.sort()
      if subset in population:
         continue

```

```
      if PatternSampled (info, subset):
         continue
      population.append (subset)

   return population

def MakeRandomGuesses (info, distanceMatrix, allResidues, numGuesses, population = []):
   totalSize = len (population) + numGuesses
   while (len (population) < totalSize):
      for i in range (10000):  #TODO set up a better check to make sure there are choices left before continuing the loop.
         subset = MatChooseSpatiallyCloseSubset (allResidues, distanceMatrix, info.numResidues, info.maxNeighborhood)
         subset.sort()
         if subset in population:
            continue
         if PatternSampled (info, subset):
            continue
         population.append (subset)
         break
      else:
         WriteLog (info.logFile ,"Could not find a random guess not already tried after 10000 tries!")
         break
   return population


def MakeMutations (survivors, allResidues, info, numMutations, population):
   resNames = allResidues.keys()
   totalSize = len (population)  + numMutations

   allMutations = []
   for parent in survivors:
      for pres in parent:
         for mres in resNames:
            if mres in parent:
               continue
            newMotif = list (parent)
            newMotif.remove (pres)
            newMotif.append (mres)
            newMotif.sort()
            allMutations.append (newMotif)
   while (allMutations and len (population) < totalSize):
      newMotif = random.choice (allMutations)
      allMutations.remove (newMotif)
      if newMotif in population:
         continue
      if newMotif in survivors:
         continue
      if PatternSampled (info, newMotif):
         continue
      population.append (newMotif)
   return population


```

```
def MakeInsertions (survivors, allResidues, info, numInsertions, population):
   resNames = allResidues.keys()
   totalSize = len (population) + numInsertions

   allInsertions = []
   for parent in survivors:
      if len (parent) >= info.maxResidues:
         continue
      for res in resNames:
         if res in parent:
            continue
         newMotif = list (parent)
         newMotif.append (res)
         newMotif.sort()
         allInsertions.append (newMotif)

   while (allInsertions and len (population) < totalSize):
      newMotif = random.choice (allInsertions)
      allInsertions.remove (newMotif)
      if newMotif in population:
         continue
      if newMotif in survivors:
         continue
      if PatternSampled (info, newMotif):
         continue
      population.append (newMotif)
   return population

def MakeDeletions (survivors, info, numDeletions, population):
   totalSize = len (population) + numDeletions
   allDeletions = []
   for parent in survivors:
      if len (parent) <= info.minResidues:
         continue
      for res in parent:
         newMotif = list (parent)
         newMotif.remove (res)
         allDeletions.append (newMotif)  # allDeletions may have duplicates, but only a finite number so its okay

   while (allDeletions and len (population) < totalSize):
      newMotif = random.choice (allDeletions)
      allDeletions.remove (newMotif)
      if newMotif in population:
         continue
      if newMotif in survivors:
         continue
      if PatternSampled (info, newMotif):
         continue
      population.append (newMotif)
   return population

def MakeRecombinations (parents, info, numRecombinations, population):
   totalSize = len (population) + numRecombinations
   attempts = 0
   while (len (population)  < totalSize and attempts < 10000):
```

```
      attempts = attempts + 1
      if not parents:
         break
      #first choose parents
      mom = random.choice (parents)
      tmp = parents[:]
      if len (tmp) > 1:
         tmp.remove(mom)
      dad = random.choice (tmp)
#     choose contributions
      all = list(mom)
      for res in dad:
         if res in mom:
            all.remove(res)
      all = all + list(dad)
      newMotif = []
      maxLen = min (info.maxResidues, len(all))
      newLen = random.randrange (3, maxLen+1)
      while (len (newMotif) < newLen):
         newMotif.append (random.choice(all))
         all.remove (newMotif[-1])

      newMotif.sort()
      if newMotif in population:
         continue
      if newMotif in parents:
         continue
      if PatternSampled (info, newMotif):
         continue
      population.append (newMotif)
   if (attempts >= 10000): #lazy but sufficient:
      WriteLog (info.logFile, "WARNING: Couldn't find a recombination that hasn't been tried in 10000 tries!" )
   return population

def EvolveNextPopulation (survivors, info, allResidues, distanceMatrix):
   nextPop = []
   l = len (nextPop)
   if info.motifs:
      nextPop.extend (info.motifs)
      info.motifs = []
      WriteLog (info.logFile, "**User specified starting motifs**")
      PrintPopulation (info, nextPop[l:])
      l = len (nextPop)

   if len (survivors) < info.popFromPrevious/2:  # no population bottlenecks
      nextPop = MakeRandomGuessesWithCoverage (info, distanceMatrix, allResidues, info.populationSize, nextPop)
      WriteLog (info.logFile, "**Random Guesses With Coverage")
      PrintPopulation (info, nextPop[l:])


      l = len (nextPop)
      nextPop = MakeRandomGuesses (info, distanceMatrix, allResidues, info.populationSize - len(nextPop), nextPop)
      WriteLog (info.logFile, "**Random Guesses")
```

```
      PrintPopulation (info, nextPop[l:])


   else:
      nextPop = MakeRandomGuesses (info, distanceMatrix,allResidues, info.popFromRandom, nextPop)
      WriteLog (info.logFile, "**Random Guesses")
      PrintPopulation (info, nextPop[l:])

      l = len (nextPop)
      nextPop = MakeMutations (survivors, allResidues, info, info.popMutations, nextPop)
      WriteLog (info.logFile, "**Mutations")
      PrintPopulation (info, nextPop[l:])

      l = len (nextPop)
      nextPop = MakeInsertions (survivors, allResidues, info, info.popInsertions, nextPop)
      WriteLog (info.logFile, "**Insertions")
      PrintPopulation (info, nextPop[l:])

      l = len (nextPop)
      nextPop = MakeDeletions (survivors, info, info.popDeletions, nextPop)
      WriteLog (info.logFile, "**Deletions")
      PrintPopulation (info, nextPop[l:])

      l = len (nextPop)
      nextPop = MakeRecombinations (survivors, info, info.populationSize - len(nextPop), nextPop)
      WriteLog (info.logFile, "**Recombinations")
      PrintPopulation (info, nextPop[l:])
   return nextPop


def PrintPopulation (info, population):
   for mot in population:
      WriteLog (info.logFile, string.join (mot, '_') )

def PrintMotifScores (info, motifScores, finalStats=0):
   scorePairs = []
   total =  0.0
   maxScore = None
   minScore = 2.0
   meanScore = 0.0
   count = 0

   for mot in motifScores.keys():
      count += 1
      total = total + motifScores[mot]
      maxScore = max (maxScore, motifScores[mot])
      minScore = min (minScore, motifScores[mot])
      scorePairs.append ( (motifScores[mot], string.join (mot, '_') ) )

   if not count:
      WriteLog (info.logFile, "WARNING: No top scorers found to describe!")
```

```
      return

   meanScore = total/count
   scorePairs.sort()
   #loop again to print, and calculate variance
   total = 0.0
   for pair in scorePairs:
      total += math.pow (pair[0]-meanScore, 2)
      if not finalStats:
         WriteLog (info.logFile,"topScorer %d %40s %8.4f" % (info.round, pair[1], pair[0]))
   if count > 2:
      variance = total/(count - 1)
   else:
      variance = 0.0
   if finalStats:
      label = "finalTopScoreStats"
   else:
      label = "topScoreStats"

   WriteLog (info.logFile, "%s %d %6.4f %6.4f %6.4f %10.8f" % (label, info.round, meanScore, maxScore, minScore, math.sqrt(variance)))

def GetTopScorer (motifScores):
   scorePairs = []
   for mot in motifScores.keys():
      scorePairs.append ( (motifScores[mot], string.join (mot, '_') ) )
   topPair = max (scorePairs)
   return topPair



def SummarizeResiduesInMotifScores (info, motifScores, allResidues, finalSummary = 0):
   ress = {}
   for mot in motifScores.keys():
      for res in mot:
         try:
            ress[res] += 1
         except KeyError:
            ress[res] = 1
   pairs = []
   for res in ress.keys():
      pairs.append ( (ress[res], res) )
   pairs.sort()
   pairs.reverse()
   if not finalSummary:
      for score,res in pairs:
         if len (allResidues[res]) > 6:
            resTypes = (allResidues[res][6])
         else:
            resTypes = []
         subString = string.join (resTypes, ",")

         WriteLog (info.logFile, "resScore %d %8s.%s %3d # %s" % (info.round, res, allResidues[res][4], score, subString))
```

```
   else:

      summaryString = ""
      for score,res in pairs:
         summaryString += "%s.%s(%02d); "% (res, allResidues[res][4], score)
      WriteLog (info.logFile, "finalResidues " + summaryString)


def PrintFinalSummary (info, motifScores, allResidues):
   #first print top score stats:
   PrintMotifScores (info, motifScores, 1)
   #now single line for final residues
   SummarizeResiduesInMotifScores (info, motifScores, allResidues, 1)



# The meat of GASPS work is performed here.

def DoGASpasm (info):
   try:
      os.makedirs (os.path.split (info.logFile)[0])
   except OSError:
      pass #most likely directory already exists.

   WriteLog (info.logFile, "\n\nGASpasm started at %s" % time.ctime(time.time()))
   # First get information from multi align file if appropriate
   if info.alignFile:
      ma = polacco.MultiAlign.MultiAlign([])
      WriteLog (info.logFile, "Opening multialign file at %s" % info.alignFile)
      fp = open (info.alignFile)
      if (string.upper (info.alignFormat) == 'FASTA'):
         ma.read_fasta (fp)
      elif (string.upper (info.alignFormat) == 'CLUSTAL'):
         ma.read_clustal (fp, 0) # 0 sets to not force strict clustal: first line "CLUSTAL..." is optional
      else:
         print "Unrecognized align format: %s " % info.alignFormat
         sys.exit(0)
      fp.close()
      WriteLog (info.logFile, "Finished align file at %s" % time.ctime(time.time()))

      if info.alignRange:
         (start,stop) = info.alignRange.split(':')
         if start:
            start = int (start)
         else:
            start = ma.FIRST_POSITION
         if stop:
            stop = int (stop)
         else:
            stop = ma.FIRST_POSITION + ma.length()

```

```
         WriteLog (info.logFile, "Restricting alignemnt to %s and %s" % (start, stop))
         ma = ma.getMultiAlignBlock (start, stop)

      refRow = ma.get_row_by_name (info.refRowName)
      ma.protectedRows.append (refRow)
      if info.numSeqsInAlign > 0:
         WriteLog (info.logFile, "Shrinking Multiple alignment")
         ma.ShrinkByRemovingRedundancy_Efficient (info.numSeqsInAlign)

      allResidues = GetAvailableConservedResidues (info, info.pdbFile, info.chain, ma, refRow, info.minConservation, info.allowedResidues, info.numTopConservedResidues)
   else:
      allResidues = GetAvailableResidues (info.pdbFile, info.chain)

   distanceMatrix = GetDistanceMatrix (allResidues)
   #mostly for debugging purposes:
   DescribePossibilities (distanceMatrix, info.maxNeighborhood, info.numResidues, info)

   topX = {}
   motifScores = {}
   for round in range (info.numRounds):
      info.round = round
      WriteLog (info.logFile, "Round %3d started at %s" % (round, time.ctime(time.time())) )
      population = EvolveNextPopulation (topX.keys(), info, allResidues, distanceMatrix)
      motifScores = ScoreMotifs (info, population, allResidues, motifScores)
      topX = GetTopScorers (info, info.popFromPrevious, motifScores, topX)
      WriteLog (info.logFile, "New Top Scorers:")
      PrintMotifScores (info, topX)
      WriteLog (info.logFile, "Residue scores:")
      SummarizeResiduesInMotifScores (info, topX, allResidues)

   PrintFinalSummary (info, topX, allResidues)
   if info.xValidate:
      score, motif_ = GetTopScorer (topX)
      directory = os.path.join (info.filesPath, motif_)
      ValidateDirectory (info, directory)

   if info.writeTables:
      score, motif_ = GetTopScorer (topX)
      tableFile = os.path.join (info.filesPath, motif_, "spasm.table.gz")
      localName = info.filesPath + motif_ + "_table.gz"
      cmd = 'cp %s %s' % (tableFile, localName)
      os.system(cmd)

   WriteLog (info.logFile, "Finished at %s" % (time.ctime(time.time())))


def ValidateDirectory (info, directory):
```

```
   testTrueOutFile = os.path.join (directory, "testTrue_spasm.out")
   testFalseOutFile = os.path.join (directory, "testFalse_spasm.out")

   testTableFile = os.path.join (directory, "test.table")
   motifFile = os.path.join (directory, "motif.pdb")
   #load allowed residue types from motifFile
   resTypes = GetResTypesFromMotifFile (motifFile)
   if not resTypes:
      print "WARNING!  restypes not found in motif file!"
      resTypes = GetResTypesFromSpasmRunFile (os.path.join (directory, "false_spasm.com"))
      if not resTypes:
         print "Failed again reading from run file!"
   runFileFalse = os.path.join (directory, "testFalse_spasm.com")
   runFileTrue = os.path.join (directory, "testTrue_spasm.com")

   GenerateSpasmRunFile (motifFile, resTypes, directory, info.xFalseLibrary, runFileFalse, testFalseOutFile)
   GenerateSpasmRunFile (motifFile, resTypes, directory, info.trueLibrary, runFileTrue, testTrueOutFile)

   fpFalseSpasm = os.popen ("csh %s" % runFileFalse)
   falseSearch = polacco.Spasm.SpasmSearch (1)
   falseSearch.ParseSpasmHits (fpFalseSpasm)
   fpFalseSpasm.close()

   fpTrueSpasm = os.popen ("csh %s" % runFileTrue)
   trueSearch = polacco.Spasm.SpasmSearch (1)
   trueSearch.ParseSpasmHits (fpTrueSpasm)
   fpTrueSpasm.close()


   trueSkipHash = LoadTrueHash (info.xTrueSkipFile)

   returnAsString = 1
   writeOutFile = 1 #always write out this file, the most interesting one!
   tableFileString = polacco.Spasm.Convert2SpasmSearchesToSortedAndScoredTable (trueSearch, falseSearch,
                             testTableFile, info.trueHash, info.useDistanceRmsd, returnAsString,
                             writeOutFile, trueSkipHash)
   if info.rocArea:
      score = polacco.Spasm.ComputeAreaFromTableFile (testTableFile, info.maxFalse, tableFileString)
   else:
      lengthTrueLibrary = 1
      score = polacco.Spasm.ComputeSeparationScoreFromTableFile3 (testTableFile, info.maxFalse,
         info.maxRMSD, lengthTrueLibrary, info.sepScoreImportance, info.useDistanceRmsd, tableFileString)

   WriteLog (info.logFile,"Cross-Validate Result %s  %8.4f" % (directory , score ))

```

```
def SetDefaults(info):
    info.chain = ' '
    info.numResidues = 5   # first guesses motif size, min and max specify that allowed during optimization
    info.maxResidues = 10  # spasm won't display ca-ca or sc-sc matrices if this is any higher than 10
    info.minResidues = 3
    info.maxNeighborhood = 12 # in angstroms, initial guess motifs are built from residues within this distance
    info.alignFile = None  #if alignment exists already, read it here.
    info.alignFormat = 'FASTA' # format of above, alternative is CLUSTAL
    info.minConservation = 0.6 # conservation necessary for inlc
    info.filesPath = None
    info.allowedResidues = 'FILVPAGMCWYTSQNEDHKR'
    info.logFile = None
    info.tpFile = None
    info.falseSkipFile = None #these specify which items in the true or fales libraries should be skipped.
    info.trueSkipFile = None

    info.populationSize = 36
    info.popFromPrevious = 16
    info.popFromRandom = 0
    info.popInsertions = 8
    info.popMutations = 12
    info.popDeletions = 8
    info.noSubs = 1  # true or false indicating wether to turn off substitutions (beware double negative!)
    info.maxFalse = 5 # cut off for computing ROC scores
    info.numRounds = 50 # number of rounds to complete before stopping
    info.testing = 0
    info.rocArea = 0 #binary indicating what kind of scoring to use (rocArea, vs separation score)
    info.numSeqsInAlign = -1 #if greater than zero this determines the size of the multi-align to use
    info.numTopConservedResidues = 100 #if greater than 0 this determines the number of positions in the multialign to choose as conserved residues
                                       #if less than 1 it specifies fraction to accept, if greater than 1 it represents the number of residues to accept
    info.alignRange = ':' #specifies a range of columns to limit the multialign

    info.trueLibrary  = None
    info.falseLibrary = None

    info.validateDirectory = None

    info.lengthTrueLibrary = None
    info.sepScoreImportance = 0.1

    info.useDistanceRmsd = 0

    info.motifs = []
```

```
   info.scratchPath = ''

   info.xValidate = 0 #perform cross validation at the end of a completed run.
                  #reuse tp library, but give different exclude list
                  #must give new fp library.
   info.xTrueSkipFile = None
   info.xFalseLibrary = None

   info.doNotCountQuery = None # used to pass the name of the query in the the spasm library
                               # so that matches to itself can be ignored.

   info.writeTables=0     #write out spasm output tables from each run. Eats disk space.
   info.useFileNames=0    #use the file name given in the spasm library to describe the matched structure.
                          # the alternative is to use the four character pdb-style code given in the spasm library.
                          # if turned on, this will use "d1qcrd2" from /pdbstyle-1.63/qc/d1qcrd2.ent
                          # used by polacco.Spasm.SpasmHit.ReadOpenFile()

   info.generateAlignment = None #file to use as input to psiblast for generation of alignment


   info.maxRMSD = 3.2 #thresholds to be passed to spasm
   info.maxCADiff = 5.0
   info.maxSCDiff = 3.8

   info.scOnly = 0 #use spasm in sidechain-only mode; ignore alpha carbons




def PrintUsage(short, long):
   for i in range (len(short)):
      print '%s     %s' % (short[i], long[i])

def SetUpSkipHashes(info):

   if info.trueSkipFile:
      info.trueSkipHash = LoadTrueHash (info.trueSkipFile)
      WriteLog (info.logFile, "loaded %d items to skip from %s" % (len (info.trueSkipHash.keys()), info.trueSkipFile))
   else:
      info.trueSkipHash = {}
   if info.doNotCountQuery:
      info.trueSkipHash[string.upper(info.doNotCountQuery)] = 1
   if info.falseSkipFile:
      info.falseSkipHash = LoadTrueHash (info.falseSkipFile)
      WriteLog (info.logFile, "loaded %d items to skip from %s" % (len (info.falseSkipHash.keys()), info.trueSkipFile))
   else:
```

```
      info.falseSkipHash = {}

def GenerateAlignment (info):
   import polacco.BlastXML

   psiBlastFile = info.generateAlignment + ".psiblast.xml"
   info.alignFile = psiBlastFile + ".faln"
   print "Looking for " + info.alignFile
   if FileExists (info.alignFile):
      print "Found align file, not repeating psiblast."
      return

   cmd = "%s -d %s -i %s -o %s -m7 -j2" % (__blastpgpPath,
__blastDB, info.generateAlignment, psiBlastFile)
   print cmd
   os.system (cmd)

   polacco.BlastXML.GetAlignmentFromPsiBlastFile (psiBlastFile,
info.alignFile)

   os.remove (psiBlastFile)


# dummy class mostly to allow me to easily store any number of
configuration variables
class struct:
   pass

def main():
   info= struct()
   #display the next line unwrapped for an easy mapping from short
option to long option, or simply do GASPS.py -h
   shortList =   ['h',    'p:',       'H:',     'r:',
'n:',               'a:',         'A:',          'F:',
'c:',               'o:',         'R:',               'l:',
't:',     'i:',         'P:',      'T',        'O:'      ,'m:'
,'s:'          ,'C:'                      ,'N:'        , 'G:'
, 'U:'          , 'X:'           , 'v:'                , 'b:'
, 'S:'                 , 'D:'              , 'M:'     , 'z:'
, 'k:'           , ''              , 'x'        , ''              ,
''              , ''            ,''                , ''      , ''
, ''            , '']
   longOptions = ['help', 'pdbFile=', 'chain=', 'numResidues=',
'maxNeighborhood=', 'alignFile=', 'refRowName=', 'alignFormat=',
'minConservation=', 'filesPath=', 'allowedResidues=',
'logFile=','tpFile=','iterations=','popSize=','testing',
'rocArea=','maxFalse=','numSeqsInMA=','numTopConservedResidues=','no
tTpFile=', 'alignRange=', 'trueLibrary=', 'falseLibrary=',
'validateDirectory=', 'lengthTrueLibrary=', 'sepScoreImportance=',
'useDistanceRmsd=', 'motifs=', 'scratchPath=', 'trueSkipFile=',
'falseSkipFile=', 'xValidate', 'xTrueSkipFile=', 'xFalseLibrary=',
'maxResidues=','doNotCountQuery=', 'noSubs', 'writeTables=',
'useFileNames', 'generateAlignment=', 'maxRMSD=', 'maxCADiff=',
'maxSCDiff=', 'scOnly']

   shortOptions = string.join (shortList, '')
   opts, args = getopt.getopt (sys.argv[1:], shortOptions,
longOptions)
```

```

   SetDefaults (info)

   for o,a in opts:
      if o in ('-h', '--help'):
         PrintUsage (shortList, longOptions)
         sys.exit(0)
      elif o in ('-p', '--pdbFile'):
         info.pdbFile = a
      elif o in ('-H', '--chain'):
         if a == 'space':
            a = ' '
         info.chain = a
      elif o in ('-r', '--numResidues'):
         info.numResidues = int (a)
      elif o in ('-n', '--maxNeighborhood'):
         info.maxNeighborhood = float (a)
      elif o in ('-a', '--alignFile'):
         info.alignFile = a
      elif o in ('-A', '--refRowName'):
         info.refRowName = a
      elif o in ('-F', '--alignFormat'):
         info.alignFormat = a
      elif o in ('-c', '--minConservation'):
         info.minConservation   = float (a)
      elif o in ('-o', '--filesPath'):
         info.filesPath  = a
      elif o in ('-R', '--allowedResidues'):
         if a == "NOTBORING":
            info.allowedResidues = "GSTCMPDNEQKRHFYW"
         else:
            info.allowedResidues = string.upper (a)
      elif o in ('-l', '--logFile'):
         info.logFile  = a
      elif o in ('-t', '--tpFile'):
         if a != 'none':
            info.tpFile = a
      elif o in ('-i', '--iterations'):
         info.numRounds = int(a)
      elif o in ('-P', '--popSize'):
         info.populationSize = int(a)
      elif o in ('-T', '--testing'):
         print "TESTING, no spasm runs will be attempted. Scores chosen randomly!!!!"
         info.testing = 1
      elif o in ('-L', '--library'):
         pass
      elif o in ('-O', '--rocArea'):
         info.rocArea = int (a)
      elif o in ('-m', '--maxFalse'):
         info.maxFalse = int (a)
      elif o in ('-s', '--numSeqsInMA'):
         info.numSeqsInAlign = int (a)
      elif o in ('-C', '--numTopConservedResidues'):
         info.numTopConservedResidues = float (a)
      elif o in ('-N', '--notTpFile'):
         info.notTpFile = a
      elif o in ('-G', '--alignRange'):
```

```
         info.alignRange = a
      elif o in ('-U', '--trueLibrary'):
         info.trueLibrary = a
      elif o in ('-X', '--falseLibrary'):
         info.falseLibrary = a
      elif o in ('-v', '--validateDirectory'):
         info.validateDirectory = a
      elif o in ('-b', '--lengthTrueLibrary'):
         info.lengthTrueLibrary = int(a)
      elif o in ('-S', '--sepScoreImportance'):
         info.sepScoreImportance = float (a)
      elif o in ('-D', '--useDistanceRmsd'):
         info.useDistanceRmsd = int (a)
      elif o in ('-k', '--trueSkipFile'):
         info.trueSkipFile = a
      elif o in ('--falseSkipFile',):
         info.falseSkipFile = a
      elif o in ('-x', '--xValidate'):
         info.xValidate = 1
      elif o  == '--xTrueSkipFile':
         info.xTrueSkipFile = a
      elif o == '--xFalseLibrary':
         info.xFalseLibrary = a
      elif o == '--maxResidues' :
         info.maxResidues = int (a)
      elif o == '--doNotCountQuery':
         info.doNotCountQuery = a
      elif o in ('-M', '--motifs'):
         motifs = a.split (",")
         for mot in motifs:
            mot = mot.split("_")
            mot.sort()
            info.motifs.append (mot)
      elif o in ('-z', '--scratchPath'):
         info.scratchPath = a
         try:
            os.makedirs (a)
         except OSError, data:
            print ("Error (ignored) while generating scratchPath: %s" % a)
            print (data.strerror)

      elif o == '--noSubs':
         print "ATTENTION: No substitutions will be allowed."
         info.noSubs = 1
      elif o == '--writeTables':
         info.writeTables = int (a)
      elif o == '--useFileNames':
         info.useFileNames = 1
      elif o == '--generateAlignment':
         info.generateAlignment = a

      elif o == '--maxRMSD':
         info.maxRMSD = float (a)
      elif o == '--maxCADiff':
         info.maxCADiff = float (a)
      elif o == '--maxSCDiff':
         info.maxSCDiff = float (a)
```

```
      elif o == '--scOnly':
         info.scOnly = 1

      else:
         print "Unrecognized option: %s  : %s, use '-h' for list of possible arguments" % (o,a)
         sys.exit(0)

   if info.motifs:
      print ( "Loaded %d motifs from input" % len (info.motifs))


   if not info.logFile:
      info.logFile = info.filesPath+"_log.txt"


   if info.tpFile:
      info.trueHash = LoadTrueHash (info.tpFile)
   else:
      info.trueHash = {}
   SetUpSkipHashes(info)

   print " Settings from command line and defaults: "
   DescribeMembers (info, sys.stdout)

   if info.generateAlignment:
      GenerateAlignment(info)

   if info.doNotCountQuery:
      info.lengthTrueLibrary -= 1

   if info.validateDirectory:
      print TestMotif (info.validateDirectory, info.trueLibrary, info.falseLibrary)
   else:

      DoGASpasm (info)

if __name__ == "__main__":
   main()


```

polacco/BlastXML.py

```
import polacco.XML, polacco.MultiAlign
import string, sys


#      ____  _           _  __  ____  __ _
#     | __ )| | __ _ ___| |_\ \/ /  \/  | |      _ __  _   _
#     |  _ \| |/ _` / __| __|\  /| |\/| | |     | '_ \| | | |
#     | |_) | | (_| \__ \ |_ /  \| |  | | |___ _| |_) | |_| |
#     |____/|_|\__,_|___/\__/_/\_\_|  |_|_____(_) .__/ \__, |
#                                               |_|    |___/

```

```

# Despite it's name, it works equally well with the XML output of both blastall and blastpgp.


class PsiBlastXMLFile (polacco.XML.XML_tree) :

   def __init__ (self, openFileIn):
      self.maxEValue = 0
      self.minHitOverlapFraction = 0.0

      polacco.XML.XML_tree.__init__(self, openFileIn)
      self.queryLength = self.GetQueryLength()


   def SetMaxEValue (self, maxEValue):
      self.maxEValue = float(maxEValue)
   def SetQueryLength (self, queryLength):
      self.queryLength = int (queryLength)
   def SetMinHitOverlapFraction (self, minHitOverlapFraction):
      self.minHitOverlapFraction = float (minHitOverlapFraction)

   def GetHits (self, iteration = -1):
      hits = self.rootNode.subNodes['BlastOutput_iterations'][-1].subNodes['Iteration'][iteration].subNodes['Iteration_hits'][-1].subNodes['Hit']
      return hits

   def GetQueryLength (self):
      return int (self.rootNode.subNodes['BlastOutput_query-len'][-1].value)

   # returns tuples of id, accession, hitDef, evalue
   def GetSimpleHits (self, iteration = -1):
      simpleHits = []
      hits = self.GetHits(iteration)

      for hit in hits:
         accession = hit.subNodes['Hit_accession'][-1].value
         id = hit.subNodes['Hit_id'][-1].value
         hitDef = hit.subNodes['Hit_def'][-1].value

         #now pick best evalue from all hsps
         hsps = hit.subNodes['Hit_hsps'][-1].subNodes['Hsp']
         eValue = max ( [ hsp.subNodes['Hsp_evalue'][-1].value for hsp in hsps] )
         simpleHits.append ( (id, accession, hitDef, eValue) )

      return simpleHits

   def GetMultiAlignment (self, iteration = -1):
      hits = self.GetHits (iteration)
      multiAlign = None

      queryName = self.rootNode.subNodes['BlastOutput_query-def'][-1].value

      for hit in hits:
```

```
         accession = hit.subNodes['Hit_accession'][-1].value
         id = hit.subNodes['Hit_id'][-1].value
         hitDef = hit.subNodes['Hit_def'][-1].value

         hsps = hit.subNodes['Hit_hsps'][-1].subNodes['Hsp']
         #I don't want to assume these are already sorted by evalue, so sort them by evalue
         tempToSort = [ (hsp.subNodes['Hsp_evalue'][-1].value, hsp) for hsp in hsps]
         tempToSort.sort()
         tempToSort.reverse()
         hsps = [row[1] for row in tempToSort]

         #determine which hsps are worth keeping (IMHO)
         #keep the most significant that do not overlap with any others on either the query or match sequence
         goodHsps = []
         def _QueryOverLap (hsp1, hsp2):
            if hsp1.subNodes['Hsp_query-to'][-1] < hsp2.subNodes['Hsp_query-from'][-1]:
               return 0
            elif hsp1.subNodes['Hsp_query-from'][-1] > hsp2.subNodes['Hsp_query-to'][-1]:
               return 0
            else:
               return 1
         def _MatchOverLap (hsp1, hsp2):
            if hsp1.subNodes['Hsp_hit-to'][-1] < hsp2.subNodes['Hsp_hit-from'][-1]:
               return 0
            elif hsp1.subNodes['Hsp_hit-from'][-1] > hsp2.subNodes['Hsp_hit-to'][-1]:
               return 0
            else:
               return 1

         for hsp in hsps:
            for goodHsp in goodHsps:
               if _QueryOverLap(hsp, goodHsp) and _MatchOverLap (hsp, goodHsp):
                  break
            else:
               goodHsps.append (hsp)

         hsps = goodHsps

         if len(hsps) > 1:
            print "More than one hsp found and used for %s %s" % (accession, id)
            #print summaries of overlaps.
            for hsp in hsps:
               sys.stdout.write ("query:")
               for i in range (0, hsp.subNodes['Hsp_query-from'][-1], 5):
                  sys.stdout.write (".")
               for i in range (hsp.subNodes['Hsp_query-from'][-1], hsp.subNodes['Hsp_query-to'][-1], 5):
                  sys.stdout.write ("Q")
```

```
               sys.stdout.write ('\n')

               sys.stdout.write ("hit  :")
               for i in range (0, hsp.subNodes['Hsp_hit-from'][-1], 5):
                  sys.stdout.write (".")
               for i in range (hsp.subNodes['Hsp_hit-from'][-1], hsp.subNodes['Hsp_hit-to'][-1], 5):
                  sys.stdout.write ("H")
               sys.stdout.write ('\n')

         #hsp = hsps[0]
         i = 0
         for hsp in hsps:

            #make sure e value is significant
            if float(hsp.subNodes['Hsp_evalue'][-1].value) > self.maxEValue:
               continue
            #make sure we are aligning to a significant fraction of the query
            alignQueryLength = int(hsp.subNodes['Hsp_query-to'][-1].value) - int(hsp.subNodes['Hsp_query-from'][-1].value)
            if float(alignQueryLength)/self.queryLength < self.minHitOverlapFraction:
               continue

            hitName = accession
            if i > 0:
               hitName = hitName + ".%d" % i
            i+=1

            if not multiAlign:
               parentRow = polacco.MultiAlign.AlignRow ( queryName , int(hsp.subNodes['Hsp_query-from'][-1].value), list (hsp.subNodes['Hsp_qseq'][-1].value))
               multiAlign = polacco.MultiAlign.MultiAlign ([parentRow])

            master = polacco.MultiAlign.AlignRow ( queryName , int(hsp.subNodes['Hsp_query-from'][-1].value), list(hsp.subNodes['Hsp_qseq'][-1].value))
            #slave = polacco.MultiAlign.AlignRow  (hitDef[0:50], int(hsp.subNodes['Hsp_hit-from'][-1].value), list(hsp.subNodes['Hsp_hseq'][-1].value))
            slave = polacco.MultiAlign.AlignRow  (hitName, int(hsp.subNodes['Hsp_hit-from'][-1].value), list(hsp.subNodes['Hsp_hseq'][-1].value))

            try:
               multiAlign.addPair (master, slave)
            except "AlignmentOutOfRange":
               raise "AlignmentOutOfRange"

      print "Success reading alignment from (psi)blast file!"
      return multiAlign, parentRow


```

```




def GetAlignmentFromPsiBlastFile (fileName, outFileName = None):
   fp = open (fileName)

   pbFile = PsiBlastXMLFile (fp)
   fp.close()
   if not outFileName:
      outFileName = fileName + ".faln"

   pbFile.SetMaxEValue (1.0e-10)
   #pbFile.SetQueryLength (355)
   pbFile.SetMinHitOverlapFraction (0.5)

   ma,parentRow = pbFile.GetMultiAlignment()
   del (pbFile)

   ma.protectedRows.append(parentRow)
   ma.ShrinkByRemovingRedundancy_Efficient(50)
   ma.RemoveGappedColumns()
   ma.DashifyGapCharacters()
   fpout = open (outFileName, "w")
   ma.simple_print (fpout)
   fpout.close()
   return outFileName


if __name__ == '__main__':
   GetAlignmentFromPsiBlastFile (sys.argv[1])





def test():
   fp = open ("longtest.xml")
   pbtree = PsiBlastXMLFile (fp)

   ma = pbtree.GetMultiAlignment()
```

polacco/Data.py

```
#######################################################
#
#         ____        _
#        |  _ \  __ _| |_ __ _   _ __  _   _
#        | | | |/ _` | __/ _` | | '_ \| | | |
#        | |_| | (_| | || (_| |_| |_) | |_| |
#        |____/ \__,_|\__\__,_(_) .__/ \__, |
#                               |_|    |___/
#
#
#
```

```
#      Things were done to save typing time, not necessarily program running time
#########################################################
import string

global PET91_matrix
PET91_matrix = None

def GetPET91_matrix():
   global PET91_matrix
   if PET91_matrix:
      return PET91_matrix

   aaOrder = "ARNDCQEGHILKMFPSTWYV"
   PET91_matrix = {}

   # PET91 matrix for 120 PAM (Jones, Thornton and Taylor)
   temp = {}
   temp["A"] = string.split ("   6  -3  -1  -2  -3  -3  -2   0  -4  -1  -4  -4  -2  -6   0   2   2  -7  -7   1")
   temp["R"] = string.split ("  -3   8  -2  -4  -1   2  -3  -1   2  -6  -5   4  -4  -8  -2  -2  -3  -1  -5  -6")
   temp["N"] = string.split ("  -1  -2   8   3  -3  -1  -1  -1   2  -4  -6   1  -4  -6  -3   2   1  -8  -2  -5")
   temp["D"] = string.split ("  -2  -4   3   8  -7  -1   5   0  -1  -7  -8  -2  -7  -9  -5  -2  -3 -10  -4  -5")
   temp["C"] = string.split ("  -3  -2  -3  -7  14  -6  -8  -3  -2  -5  -5  -6  -5  -2  -5   0  -3   0   2  -3")
   temp["Q"] = string.split ("  -3   2  -1  -1  -6   9   2  -4   4  -6  -3   2  -4  -7   0  -3  -3  -6  -4  -6")
   temp["E"] = string.split ("  -2  -3  -1   5  -8   2   8  -1  -3  -7  -8   0  -6 -10  -5  -3  -4  -8  -7  -5")
   temp["G"] = string.split ("   0  -1  -1   0  -3  -4  -1   8  -4  -6  -8  -4  -6  -9  -4   0  -3  -3  -8  -4")
   temp["H"] = string.split ("  -4   2   2  -1  -2   4  -3  -4  11  -6  -4  -1  -5  -2  -1  -2  -3  -6   4  -6")
   temp["I"] = string.split ("  -1  -6  -4  -7  -5  -6  -7  -6  -6   7   2  -6   3  -1  -5  -3   0  -7  -5   5")
   temp["L"] = string.split ("  -4  -5  -6  -8  -5  -3  -8  -8  -4   2   7  -6   3   2  -2  -3  -4  -4  -4   1")
   temp["K"] = string.split ("  -4   4   1  -2  -6   2   0  -4  -1  -6  -6   8  -4 -10  -4  -3  -2  -6  -7  -6")
   temp["M"] = string.split ("  -2  -4  -5  -7  -5  -4  -6  -6  -5   3   3  -4  10  -2  -4  -3   0  -6  -6   2")
   temp["F"] = string.split ("  -6  -8  -6  -9  -2  -7 -10  -9  -2  -1   2  -9  -2  11  -5  -3  -6  -3   5  -2")
   temp["P"] = string.split ("   0  -2  -3  -5  -5   0  -5  -4  -1  -5  -2  -4  -5  -5   9   1   0  -8  -6  -4")
   temp["S"] = string.split ("   2  -2   2  -2   0  -3  -3   0  -2  -3  -3  -3  -3  -3   1   5   2  -5  -3  -3")
   temp["T"] = string.split ("   2  -3   1  -3  -3  -3  -4  -2  -3   0  -4  -2   0  -6   0   2   6  -7  -6   0")
   temp["W"] = string.split ("  -8  -1  -8  -9   0  -6  -9  -3  -6  -7  -4  -6  -6  -3  -8  -5  -7  17  -2  -6")
   temp["Y"] = string.split ("  -7  -5  -2  -4   2  -4  -7  -8   4  -5  -4  -7  -6   5  -7  -3  -6  -2  12  -6")
   temp["V"] = string.split ("   1  -6  -5  -5  -3  -6  -5  -4  -6   5   1  -6   2  -2  -4  -3   0  -6  -6   7")
```

```
   #temp["B"] = string.split ("   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0")

   for aaRow in aaOrder:
      PET91_matrix[aaRow] = {}
      for i in range (len (aaOrder)):
         PET91_matrix[aaRow][aaOrder[i]] = int(temp[aaRow][i])

   #now do specials


   return PET91_matrix


```

polacco/MultiAlign.py

```
#! /sw/bin/python


#    __  __       _ _   _    _    _ _
#   |  \/  |_   _| | |_(_)  / \  | (_) __ _ _ __    _ __  _   _
#   | |\/| | | | | | __| | / _ \ | | |/ _` | '_ \  | '_ \| | | |
#   | |  | | |_| | | |_| |/ ___ \| | | (_| | | | |_| |_) | |_| |
#   |_|  |_|\__,_|_|\__|_/_/   \_\_|_|\__, |_| |_(_) .__/ \__, |
#                                     |___/        |_|    |___/



from string import *
import sys, copy, math, string



def is_gap(char):
   if (char == '-' or char == '.' or char == '?'):
      return 1
   else:
      return 0

def is_ambiguous (char):
   if (char in 'XBUxbu'):
      return 1
   else:
      return 0


class MultiAlign:
   def __init__(self, rows = None):
      if not rows:
         self.rows = []
      else:
         self.rows = rows
      self.FIRST_POSITION = 1 #how should the API call the first position (0 or 1)
      self.protectedRows = [] #these are rows that are of special interest, see ShrinkByRemovingRedundancy_Efficient
```

```

   def LoadFromFastaFile (self, fastaFile):
      fp = open (fastaFile, "r")
      self.read_fasta (fp)
      fp.close()


   def read_fasta(self, fasta_file):
      if len (self.rows) > 0:
         print "WARNING: Blindly adding new fasta alignment to current alignment!"

      name = ''
      lines = []

      for line in fasta_file.readlines():
         if line[0] in (">"):
            #we're done with previous row
            if name:
               #first join lines, then split and join to remove spaces, then repeat to remove \n, then make list
               chars = list (join(split(join (split(join (lines, ''), " "),''), "\n"),""))
               self.rows.append(AlignRow (   name, offset, chars))
            lines = []
            slash_split = split (line[1:-1], '/')
            if (len (slash_split) > 1):
               name = join(slash_split[:-1], '')
               try:
                  offset = int (split (slash_split[-1], '-')[0])
               except ValueError:
                  offset = 1

            else:
               offset = 1
               name = slash_split[0]
         else:
            lines.append(line)
      else:
         #if everything went well we have to add the last sequence.
         if lines:
            chars = list (join(split(join (split(join (lines, ''), " "),''), "\n"),""))
            self.rows.append(AlignRow (   name, offset, chars))


   def LoadFromClustalFile (self, clustalFile, strictCLUSTAL = 1):
      print "Loading alignment from %s" % clustalFile
      fp = open (clustalFile, 'r')
      self.read_clustal (fp, strictCLUSTAL)
      fp.close()



   def read_clustal (self, clustal_file, strictCLUSTAL = 1):
      if len(self.rows) > 0:
         print "WARNING: Blindly adding new fasta alignment to current alignment! Current row names:"
```

```
         print [row.name for row in self.rows]
      partsHash = {}
      inorder = []
      fileStarted = 0
      while (1):
         line = clustal_file.readline()
         if line == '':
            break
         words = line.split()
         if (len (words) == 0):
            continue

         elif (not fileStarted):
            if words[0] == "CLUSTAL":
               fileStarted = 1
               continue
            if strictCLUSTAL:
               continue
            else:
               fileStarted = 1

         elif (words[0][0] in ":.*"):
            if len (partsHash.keys()) == 0:
               print "WARNING:  unexpected location of conservation line, possibly illegal first character for sequence name?"
               print line
            #looks like a line indicating conservation, skip it
            continue

         try:
            partsHash[words[0]].append (string.join (words[1:], ''))
         except KeyError:
            inorder.append (words[0])
            partsHash[words[0]] = [string.join (words[1:], '')]

      length = -1
      #error checking: make sure file was started.  and make sure we have same number of parts for all
      for key  in inorder:
         parts = partsHash[key]
         if length < 0:
            length = len (parts)
         elif len (parts) != length:
            print "Current: " + key
            print "Parts mismatch: %d with %d" % (len(parts), length)
            print inorder
            print parts
            raise "Bad Alignment Read!"
         chars = list (string.join(parts, ""))
         #see if we can get the offset from the name (key)
         slash_split = split (key, '/')
         if (len (slash_split) > 1):
            name = join(slash_split[:-1], '')
            try:
               offset = int (split (slash_split[-1], '-')[0])
            except ValueError:
               offset = 1
```

```
         else:
            offset = 1
            name = slash_split[0]
         self.rows.append(AlignRow (   name, offset, chars))


   # very simple FASTA format
   def simple_print(self, outfile):
      for row in self.rows:
         outfile.write (">%s\n"%(row.description() ))
         outfile.write ("%s\n" % join(row.chars, '')  )

   # roughly clustal format
   def PrettyPrint (self, outfile, columns = 60, nameWidth = 20):
      minLength = self.min_length()
      nameFormatString = '%%-%ds' % nameWidth

      i = 0
      while (i <  minLength):
         end = i+ columns
         for row in self.rows:
            outfile.write (nameFormatString %
row.name.split()[0][0:nameWidth])
            outfile.write ("      %s\n" % string.join
(row.chars[i:end], ''))
         i = end
         if i >= minLength:
            break
         outfile.write ("\n\n")


   def get_row_by_name (self, name):
      for row in self.rows:
         if row.name == name:
            return row
      else:
         print "row not found among: "
         print map((lambda row: row.name), self.rows)
         return 0

   def remove_row_by_name(self, name):
      for row in self.rows:
         if row.name == name:
            self.rows.remove(row)
            break
      else:
         raise ValueError


   def get_column (self, position):
      column = {}
      if position >= self.FIRST_POSITION:
         for row in self.rows:
            column[row] =  (row.chars[position-self.FIRST_POSITION])
      elif position < 0:
         for row in self.rows:
            column[row] =  (row.chars[position])
      else:
```

```
         raise "Illegal MultiAlign Position"

      return column

   def get_column_aslist (self, position):
      column = []
      if position >= self.FIRST_POSITION:
         for row in self.rows:
            column.append(row.chars[position-self.FIRST_POSITION])
      elif position < 0:
         for row in self.rows:
            column.append(row.chars[position])
      else:
         raise "Illegal MultiAlign Position"

      return column


   def insert_gap (self, position):
      self.insertColumnIndex ( position)

      if position == -1:
         for row in self.rows:
            if row.chars[-1] == '.':
               row.chars.append ('.')
            else:
               row.chars.append('-')

      else:
         position = position - self.FIRST_POSITION
         for row in self.rows:
            if position == 0 or row.chars[position - 1] == '.':
               row.chars[position:position] = ['.']
            else:
               row.chars[position:position] = ['-']

   def max_length (self):
      if len (self.rows) == 0:
         return 0
      return max (map (lambda r : len (r.chars), self.rows))

   def min_length (self):
      if len (self.rows) == 0:
         return 0
      return min (map (lambda r : len (r.chars), self.rows))

   #only use if you KNOW that minlength and maxlength are the same
   def length (self):
      minim = self.min_length()
      maxim = self.max_length()
      assert (minim == maxim)
      return maxim


   def getMultiAlignBlock (self, start_column, stop_column):
      subrows = []
      for row in self.rows:
```

```
         subrows.append (row.subRow (start_column - self.FIRST_POSITION,
                               stop_column - self.FIRST_POSITION +1)) #+1 includes the stop_column

      return MultiAlign (subrows)

   def MAPositionFromSeqPosition (self, refRow, rowPosition):
      numChars = refRow.num_chars_at_offset (rowPosition )

      return numChars + self.FIRST_POSITION


   def ComputePIDTable (self):
      pidTable = {}
      for seqRow in self.rows:
         pidTable[seqRow] = {}
         for seqCol in self.rows:
            try:
               pidTable[seqRow][seqCol] = pidTable[seqCol][seqRow]
            except KeyError:
               pidTable[seqRow][seqCol] = seqRow.PercentIdentity (seqCol)
      return pidTable

   def ComputePIDList (self, rows = None):
      if not rows:
         rows = self.rows
      pidList = []
      done = []
      for seq1 in rows:
         done.append(seq1)
         for seq2 in rows:
            if seq2 in done:
               continue
            pidList.append ([seq1.PercentIdentity (seq2), seq1, seq2])
      return pidList

   def GetHighestPIDPair (self, pidList):
      max = 0.0
      topPair = None

      for pair in pidList:
         if (pair[1] in self.protectedRows and pair[2] in self.protectedRows):
            continue
         if pair[0] > max:
            max = pair[0]
            topPair = pair
      if topPair == None:
         print "Warning: All pairs are protected!"

      return topPair

   # At least it's more efficient than my last one which is now deleted. No other
   # claims are made to its efficiency!
```

```
   def ShrinkByRemovingRedundancy_Efficient (self, maxSeqs):
      if maxSeqs >= len (self.rows):
         print "No shrinking needed (%d, %d)" % (maxSeqs, len (self.rows))
         return
      #compute initial pidlist
      #print "Computing initial PIDList"
      keepRows = []
      for row in self.protectedRows:
         if row in self.rows and not row in keepRows:
            keepRows.append (row)


      i = 0
      for row in self.rows:
         if len (keepRows) >= maxSeqs:
            break
         if not row in keepRows:
            keepRows.append (row)
         i = i + 1


      pidList = self.ComputePIDList(keepRows)

      #find highest identity, and rows that give it
      topPidPair = self.GetHighestPIDPair (pidList)
      #print topPidPair

      #for each row not yet in keep, find if it is more similar than highest identity in keep to any
      #if it is, then skip it
      #if it is not then add it, and its pidList
      #then remove one or the other row of highest scoring pair
      removed = [] # we have to keep a running tally of these to remove when we're don, because we can't change self.rows while we're iterating over it
      for candidate in self.rows[i:]:
         if candidate in self.protectedRows:
            continue
         candPidList = []

         for row in keepRows:
            pid = row.PercentIdentity (candidate)
            candPidList.append ( [pid,candidate, row] )
            if pid >= topPidPair[0]:
               #print "Removing row: %s because it is %6.4f identical to %s compared to running high of %6.4f (%s and %s) " % (candidate.name, pid, row.name, topPidPair[0], topPidPair[1].name, topPidPair[2].name)
               removed.append (candidate)
               break
         else: #it was not more similar than topPercentIdentity to anything else, then we need to add it
            #first remove other
            length = [0,0,0]

            length[1] = topPidPair[1].numCharsNoGaps()
            length[2] = topPidPair[2].numCharsNoGaps()
```

```

            #they can't both be in protectedRows because the GetHighestPIDPair function prevents it
            if topPidPair[1] in self.protectedRows:
               keep = 1
            elif topPidPair[2] in self.protectedRows:
               keep = 2
            elif length[1] >= length[2] :
               keep = 1
            else:
               keep =2

            if keep == 1:
               rem = 2
            else:
               rem = 1


            removed.append (topPidPair[rem])
            #print "Removing row: %s (%d chars) to keep %s with (%d chars); similarity = %6.4f" % (topPidPair[rem].name, length[rem], topPidPair[keep].name,length[keep], topPidPair[0])

            #now combine pidList and remove old entries
            pidList = pidList + candPidList
            toRemove = []
            for pair in pidList:
               if pair[1] == topPidPair[rem] or pair[2] == topPidPair[rem]:
                  toRemove.append(pair)
            for r in toRemove:
               pidList.remove (r)

            #now update keepRows
            keepRows.remove (topPidPair[rem])
            keepRows.append (candidate)

            #find new highest identity, and rows that give it
            topPidPair = self.GetHighestPIDPair (pidList)


      print "Removing %d rows to keep %d rows that have at most %6.4f sequence identity (%s)" % (len (removed), len(keepRows), topPidPair[0], self.rows[0].name)
      #now do actual removing from self
      #print [r.name for r in removed]
      for r in removed:
         #print "Removing %s" % r.name
         self.rows.remove (r)

      #done?

   def RemoveGappedColumns(self):
      toRemove = []

      for i in range (self.length()):
         for j in range (len (self.rows)):
            if not is_gap (self.rows[j].chars[i]):
```

```
               break
         else:
            toRemove.append(i)

      toRemove.reverse()
      for i in toRemove:
         self.RemoveColumnBySimpleIndex (i)

   def RemoveColumnBySimpleIndex (self, index):
      for i in range (len (self.rows)):
         self.rows[i].chars[index:index+1] = []

   def DashifyGapCharacters (self):
      for row in self.rows:
         row.DashifyGapCharacters()

   def GetDominantLettersPerColumn (self, minFraction = 0.1):
      seqLetters = []
      numRows = len (self.rows)
      for i in range (self.length()):
         columnLetters = {}
         domLetters = []
         for row in self.rows:
            if is_gap (row.chars[i]):
               continue
            try:
               columnLetters[row.chars[i]]+=1.0
            except KeyError:
               columnLetters[row.chars[i]] = 1.0
         for char in columnLetters.keys():
            if columnLetters[char]/numRows > minFraction:
               domLetters.append (char)
         seqLetters.append(domLetters)
      return seqLetters
   def addPair (self, master, slave):
      return self.addMultiAlign (MultiAlign ([master, slave]))

#################################################################
#  The next four functions deal with adding or aligning two multiple
#  alignments to each other.  These alignments  must share one row
# with the same name and with an identical overlapping region.  No aa
# mismatches are allowed.  anchor refers to that row in the self alignment
# tether refers to that row in the otherAlign.
##################################################################

#  Only generates a map of columns in one align to columns in the other
#  No alignments (except temporary copies) should be modified
   def MapColumns2OtherAlign (self, otherAlign):
      (tether, anchor) = self.FindTetherAndAnchor(otherAlign)

      #do all work on a copy of anchor and tether
      anchor = AlignRow (anchor.name, anchor.offset, anchor.chars)
      tether = AlignRow (tether.name, tether.offset, tether.chars)

```

```
      #set up dummy clones of MultiAligns to keep track of columnIndeces
      otherAlign2 = MultiAlign ([tether])
      otherAlign2.index = copy.deepcopy(otherAlign.columnIndex)
      self2 = MultiAlign ([anchor])
      self2.index = copy.deepcopy (self.columnIndex)

      map =  self.align2Aligns (self2, otherAlign2, tether, anchor)


      return (map, self2)

# This does the work of inserting gaps and extensions necessary so that 2 alignments
# can be directly compared or added by simply stacking

   def align2Aligns (__self, firstAlign, otherAlign, tether, anchor):
      #print "Before dealing with offsets"
      #pretty_print ( [join(anchor.chars,''),  join(tether.chars, '') ], 60)

      #we're about to rely on offset values (a bad idea), so fix the anchor's offset
      #print anchor.offset, tether.offset
      anchor.fixOffset (tether)
      #print anchor.offset, tether.offset
      #first deal with cases where the left end of tether begins after anchor
      if tether.offset > anchor.offset:
         split = anchor.num_chars_at_offset (tether.offset)
         if split > len (anchor.chars):
            sys.stderr.write ("alignment out of range, shares two non overlapping regions of %s\n" % (anchor.name))
            raise "AlignmentOutOfRange"
         i = split
         while  i:
            otherAlign.insert_column(otherAlign.FIRST_POSITION, '.')
            i = i - 1
         tether.chars[0:split] = anchor.chars[0:split]
         tether.offset = anchor.offset

      #now deal with cases where the beginning of row is to the right of beginning of master
      if anchor.offset > tether.offset:
         split = tether.num_chars_at_offset (anchor.offset)
         if split > len (tether.chars):
            sys.stderr.write ("alignment out of range, shares two non overlapping regions of %s\n" % (anchor.name))
            raise "AlignmentOutOfRange"
         i = split
         while i:
            firstAlign.insert_column (firstAlign.FIRST_POSITION, '.')
            i = i - 1

         anchor.chars[0:split] = tether.chars[0:split]
         anchor.offset = tether.offset
```

```

      #print "After dealing with offsets..."
      #pretty_print ( [join(anchor.chars,''),  join(tether.chars, '') ], 60)


      i=0
      minlength = min (len (anchor.chars), len (tether.chars))
      while (i < minlength):
         empty = ('.-')
         #print "mas: %s row: %s" % (master.chars[i], row.chars[i])
         if (anchor.chars[i] == tether.chars[i]):
      #     print "=",
            i = i + 1
            continue
         elif (anchor.chars[i] in empty) and (tether.chars[i] in empty):
      #     print "g",
            tether.chars[i] = anchor.chars[i] = '-'
            i = i + 1
            continue

         elif (tether.chars[i] in empty) and (anchor.chars[i] not in empty):
      #     print "^",
            firstAlign.insert_column (firstAlign.FIRST_POSITION + i, tether.chars[i])
         elif (anchor.chars[i] in empty) and (tether.chars[i] not in empty):
      #     print "v",
            otherAlign.insert_column (otherAlign.FIRST_POSITION + i, anchor.chars[i])

         elif (tether.chars[i] == 'X'):
      #     print "x",
            tether.chars[i] = anchor.chars[i]
         elif (anchor.chars[i] == 'X'):
      #     print "X",
            anchor.chars[i] = tether.chars[i]

         else:  # (row.chars[i] != master.chars[i]):
            pretty_print ( [join(anchor.chars,''), join(tether.chars, '') ], 60)
            print "WARNING!: tether sequence in foreign alignment does not match corresponding anchor sequence in firstAlign alignment! (%s %s)" % (anchor.chars[i], tether.chars[i])
            firstAlign.simple_print(sys.stdout)
            print "\notherAlign:\n"
            otherAlign.simple_print(sys.stdout)
            raise "exception"
         i = i + 1
         minlength = min (len (anchor.chars), len (tether.chars))

      #pretty_print ( [join(anchor.chars,''),  join(tether.chars, '') ], 60)

      #now deal with the trailing end
      diff =  len(anchor.chars) - len (tether.chars)
```

```
      if diff > 0:
         for c in anchor.chars[-diff:]:
            otherAlign.insert_column (-1, '.')
            tether.chars[-1] = c
      elif diff < 0:
         for c in tether.chars[diff:]:
            firstAlign.insert_column (-1, '.')
            anchor.chars[-1] = c
         #pretty_print ( [join(row.chars,''),  join(master.chars, ''), join(slave.chars, '')  ], 60)

      #pretty_print ( [join(anchor.chars,''),  join(tether.chars, '') ], 60)

      assert (len(anchor.chars) == len(tether.chars)), (len(anchor.chars) ,len (tether.chars))
      assert (join(anchor.chars) == join (tether.chars)), "\n%s\n%s" % (join(anchor.chars, ''), join(tether.chars, ''))

   #this does the work of finding the actual rows that are the tether and anchor.  It chooses
   # the first it finds, not necessarily the best.
   def FindTetherAndAnchor (self, otherAlign, taName = None):
      otherNames = [row.name for row in otherAlign.rows]
      anchor = 0
      for row in self.rows:
         if taName and taName != row.name:
            continue
         if row.name in otherNames:
            anchor = row
            for r in otherAlign.rows:
               if r.name == anchor.name:
                  tether = r
                  break
            break
      else:
         #corresponding master row not found, give warning
         raise "Error finding an anchor and tether %s" % (otherNames),  [row.name for row in self.rows]
      return (tether, anchor)

   # This will add otherAlign to the self alignment.
   def insert_column (self, position, char):
      if (len(char) > 1):
         raise "insert too long", char
      if position == -1:
         for row in self.rows:
            row.chars.append(char)
      else:
         position = position - self.FIRST_POSITION
         for row in self.rows:
            row.chars[position:position] = [char]

   def addMultiAlign (self, otherAlign, taName = None):
      (tether, anchor) = self.FindTetherAndAnchor (otherAlign, taName)
      #print "Using %s (%s) as tether and anchor sequences" % (tether.name, anchor.name)
```

```
      self.align2Aligns (self, otherAlign, tether, anchor)
      #otherAlign.rows.remove(tether)
      self.rows.extend(otherAlign.rows)
      self.rows.remove (tether)

#
#---------------------

#-------------------
#
# used by MultiAlign

class AlignRow:
   def __init__(self, name, offset, chars):
      self.name = name
      self.offset = offset
      self.chars = chars

   def fixOffset (self, referenceRow):
      minMatch = 20 # the reference row must minimally overlap by this many characters.
                    # the larger it is the faster it is with obvious problems if its too large
      selfChars = self.GetCharsNoGaps()
      reference = referenceRow.GetCharsNoGaps()

      correction = minMatch - len (selfChars)
      sub1Start = -correction
      sub1End = len (selfChars)
      sub2Start = 0
      sub2End = minMatch

      # find alignment by looking for matching substrings
      # "slide" one sequence across the other and look for matching substrings where substrings
      # are the overlapping regions.
      while (correction < len(reference) - minMatch):
         sub1 = selfChars[sub1Start:sub1End]
         sub2 = reference[sub2Start:sub2End]
         if sub1 == sub2:
            #print correction
            #print sub1
            #print sub2
            break

         #update for next iteration
         if sub1Start > 0:
            sub1Start = sub1Start - 1
         else:
            sub2Start = sub2Start + 1
         if sub1End + correction == len(reference):
            sub1End  -= 1
         else:
            sub2End = sub2End + 1
         correction = correction + 1

      self.offset = self.offset + correction - (self.offset - referenceRow.offset)
```

```
    def GetCharsNoGaps (self):
       ch = []
       for char in self.chars:
          if not is_gap(char):
             ch.append (char)
       return ch

    def GetCharsAndIndexesNoGaps (self):
       ch = []
       indexes = []
       for i in range (len (self.chars)):
          if not is_gap(self.chars[i]):
             ch.append (self.chars[i])
             indexes.append (i)
       return ch, indexes

    def num_chars_at_offset (self, offset):
       num=0
       if offset < self.offset and offset > 0:
          return offset-self.offset
       offset = offset - self.offset

       for char in self.chars:
          if not is_gap(char):
             if offset == 0:
                return num
             offset = offset - 1
          num+=1
       else:
          return len (self.chars) + 1

    def numCharsNoGaps(self):
       count = 0
       for char in self.chars:
          if not is_gap(char):
             count = count + 1
       return count


    def initial_gap_chars(self):
       count = 0
       for char in self.chars:
          if is_gap(char):
             count+=1
          else:
             break
       return count

    def terminal_gap_chars(self):
       count = 0
       pointer = -1
       try:
          while (1):
             if is_gap(self.chars[pointer]):
                count = count + 1
```

```
               pointer = pointer - 1
            else:
               break
      except IndexError:
         pass
      return count

   def description(self):
      title = split (self.name, "\n")[0]
      title = split (title, " ")[0]
      desc = "%s/%03d" % (title, self.offset)
      return desc

   def subRow (self, start_index, stop_index):
      return AlignRow (self.name, self.offset + start_index, self.chars[start_index:stop_index])


   def PercentIdentity (self, other):
      seq1 = self.chars
      seq2 = other.chars
      numAligned = 0.0
      ids = 0.0
      acceptable = "ACDEFGHIKLMNPQRSTVWY"

      for i in range (len (seq1)):
         if seq1[i] not in acceptable and seq2[i] not in acceptable:
            continue
         #if is_gap (seq1[i]) or is_gap (seq2[i]):
         #  continue
         #if is_ambiguous (seq1[i]) or is_ambiguous (seq2[i]):
         #  continue
         numAligned = numAligned + 1.0
         if seq1[i] == seq2[i]:
            ids = ids + 1.0
      if numAligned == 0:
         print "Warning: polacco.Mulitalign.AlignRow.PercentIdentity, can't compare two seqeunces because no meaningful characters align.  pid set to 0."
         return 0
      return ids/numAligned

   def DashifyGapCharacters (self):
      for i in range(len(self.chars)):
         if is_gap (self.chars[i]):
            self.chars[i] = '-'



class ValdarConservation:
   def __init__(self, multiAlign):
      self.align = multiAlign
      self.vMat = None

   def Compute (self, multiAlign = None):
      if not multiAlign:
         multiAlign = self.align

```

```
      if not self.vMat:
         import polacco.Data
         self.SetValdarMutMatrix (polacco.Data.GetPET91_matrix())
      #first calculate sequence distance matrix
      self.ComputeSequenceDistanceMatrix()
      #second calculate sequence weights
      self.ComputeSequenceWeights()
      #calculate the normalizing denominator
      self.ComputeTotalWeightProducts()
      #finally calculate the conservation
      conScores = []
      numSeqs = len (self.align.rows)
      for i in range (self.align.length()):
         score = 0.0
         for j in range (numSeqs):
            seqji = self.align.rows[j].chars[i].upper()
            for k in range (j+1, numSeqs):
               seqki = self.align.rows[k].chars[i].upper()
               if is_gap (seqji) or is_gap (seqki) or is_ambiguous (seqji) or is_ambiguous(seqki):
                  score = score + 0.0
               else:
                  score += (self.seqWeights[self.align.rows[j]] * self.seqWeights[self.align.rows[k]]) * self.vMat[seqji][seqki]
         conScores.append(score/self.sumProducts)
      self.conScores = conScores
      return conScores


   def SetValdarMutMatrix (self, mutMatrix):
      self.vMat = {}
      aas = mutMatrix.keys()
      #first find min and max
      minimum = maximum = None
      for aaRow in aas:
         minimum = min (minimum, min (mutMatrix[aaRow].values())) or min (mutMatrix[aaRow].values())
         maximum = max (maximum, max (mutMatrix[aaRow].values()))
      range = float(maximum - minimum)
      for aaRow in aas:
         self.vMat[aaRow] = {}
         for aaCol in aas:
            self.vMat[aaRow][aaCol] = (mutMatrix[aaRow][aaCol] - minimum)/range
      return self.vMat

   def ComputeSequenceDistanceMatrix(self):
      self.seqDistances = {}
      for seqRow in self.align.rows:
         self.seqDistances[seqRow] = {}
         for seqCol in self.align.rows:
            try:
               self.seqDistances[seqRow][seqCol] = self.seqDistances[seqCol][seqRow]
            except KeyError:
               self.seqDistances[seqRow][seqCol] = self.SeqDistance (seqRow.chars, seqCol.chars)

```

```
   def SeqDistance (self, seq1, seq2):
      acceptable = "ACDEFGHIKLMNPQRSTVWY"
      numAligned = 0
      sumScores = 0.0
      for i in range (len (seq1)):
         if seq1[i] not in acceptable or seq2[i] not in acceptable:
            continue
         #if is_gap (seq1[i]) or is_gap (seq2[i]):
         #  continue
         #if is_ambiguous (seq1[i]) or is_ambiguous (seq2[i]):
         #  continue

         sumScores = sumScores + self.vMat[seq1[i]][seq2[i]]
         numAligned = numAligned + 1
      if numAligned == 0:
         print "Warning: polacco.Mulitalign.Valdarconservation.SeqDistance, can't compare two seqeunces because no meaningful characters align.  Distance set to 1."
         return 1
      return 1 - (sumScores/numAligned)

   def ComputeSequenceWeights(self):
      self.seqWeights = {}
      count = len (self.align.rows) - 1
      for seq in self.align.rows:
         self.seqWeights[seq] = 0.0
         for other in self.align.rows:
            if seq is other:
               continue
            self.seqWeights[seq] = self.seqWeights[seq] + self.seqDistances[seq][other]
         self.seqWeights[seq] = self.seqWeights[seq] / count

   def ComputeTotalWeightProducts (self):
      self.sumProducts = 0.0
      for j in range (len (self.align.rows)):
         for k in range (j+1, len (self.align.rows)):
            self.sumProducts += (self.seqWeights[self.align.rows[j]] * self.seqWeights[self.align.rows[k]])

```

polacco/Spasm.py

```
#! /sw/bin/python
import string, sys, os.path, math, StringIO, gzip

########################################################
#    ____
#   / ___| _ __   __ _ ___ _ __ ___    _ __  _   _
#   \___ \| '_ \ / _` / __| '_ ` _ \  | '_ \| | | |
#    ___) | |_) | (_| \__ \ | | | | |_| |_) | |_| |
#   |____/| .__/ \__,_|___/_| |_| |_(_) .__/ \__, |
#          |_|                         |_|    |___/
########################################################
class SpasmSearch:
   def __init__(self, lowMemory = 0):
```

```
      self.hits = []
      self.caMatrix = None
      self.scMatrix = None
      self.lowMemory = lowMemory
      self.titleFromFileName = 0

   def ParseSpasmHits (self, openfile, loadInMemory = 1, outFile = None):
      line = openfile.readline()
      hit = None
      while (1):
         if line == '':
            break
         #read until we find a " ==> HIT  : (1CLX) " line
         if len (line) < 8 or (line[:8] != " ==> HIT" and line[:8] != " ==> TRY"):
            if len (line) > 6 and line[1:6] == 'ERROR':
               print "Error found in current file"
               print line
               if (hit):
                  print "File: " + hit.fileName
            line = openfile.readline()
            continue
         else:
            hit = SpasmHit(self)
            line = hit.ReadOpenFile (line, openfile)
            if loadInMemory:
               self.hits.append (hit)
            else:
               self.hits = (hit)
               if outFile:
                  self.WriteOutTable (outFile)

      if loadInMemory and outFile:
         self.WriteOutTable (outFile)

   def GetBestMatchesPerHit (self, useDistanceRmsd=0):
      #get best match (residues) for each hit (pdb)
      matches = []
      for hit in self.hits:
         matches.append (hit.GetBestRMSDMatch(useDistanceRmsd))
         #save a reference to the hit
         matches[-1].hit = hit

      if useDistanceRmsd:
         indexedMatches = [(m.distanceRmsd, m) for m in matches]
      else:
         indexedMatches = [(m.rmsd, m) for m in matches]
      indexedMatches.sort()
      matches = [m[1] for m in indexedMatches]


#     def __matchSort (a,b):
#        if a.rmsd < b.rmsd:
#           return -1
#        elif a.rmsd > b.rmsd:
#           return 1
#        else:
```

```
#           return 0
#     matches.sort (__matchSort)
      return matches


   def WriteOutTable (self, openFile):
      #every class writes out its prefix on the line.
      prefix = ""
      for hit in self.hits:
         hit.WriteOutTable (openFile, prefix)


   def WriteOutScoredSortedTable (self, openFile, trueHash, useFileName = 0, notTrueHash = None):
      matches = self.GetBestMatchesPerHit()

      if not notTrueHash:
         notTrueHash = {}

      for match in matches:
         if useFileName:
            name = os.path.splitext(os.path.basename(match.hit.fileName))[0]
         else:
            name = match.hit.title
         try:
            trueHash[name]
            pre = '1'  # a true positive
         except KeyError:
            try:
               notTrueHash[name]
               pre = '*'
            except KeyError:
               pre = "0" # a false positive
         match.WriteOutTable (openFile, match.hit.GetTablePrefix(pre))


   #use to convert spasm to table with a small memory footprint
   def ConvertToTable (self, openfile, outOpenFile):
      doNotLoad = 0
      self.ParseSpasmHits (openfile, doNotLoad, outOpenFile)



class SpasmHit:
   def __init__(self, search):
      self.search = search
      self.fileName = None
      self.lowMemory = search.lowMemory
      self.bestRMSD = 99999.9
      self.titleFromFileName = self.search.titleFromFileName


   def ReadOpenFile (self, titleLine, fp):
      if self.titleFromFileName:
         self.title = "noFileName!" #yet, will be set later
      else:
```

```
         self.title = titleLine.split(":")[1].strip()
         self.title.strip
         if self.title[0] == '(': # strip () if necessary
            self.title = self.title[1:-1]

      self.matches = []
      line = fp.readline()
      while (1):
         if len (line) > 8 and line[:8] == " MATCH #":
            match = SpasmMatch(self.search)
            line = match.ReadOpenFile (line, fp)

            if self.lowMemory: #only store the best scoring match here
               if self.matches and match.rmsd > self.bestRMSD:
                  match = None
                  continue
               else:
                  self.bestRMSD = match.rmsd
                  self.matches = []
            self.matches.append (match)
         elif line == '':
            break

         elif len (line) > 5 and (line[:5] == " File"):
            self.fileName = line.split()[2]
            if self.fileName[0] == "(":
               self.fileName = self.fileName[1:-1]
            if self.titleFromFileName:
               #want d1qcrd2 from  /pdbstyle-1.63/qc/d1qcrd2.ent
               self.title = string.upper (os.path.basename(self.fileName))
               #remove extension:
               try:
                  self.title = self.title [:string.rindex(self.title, ".")]
               except ValueError:
                  #no periods, leave title alone
                  pass

            line = fp.readline()

         elif len (line) > 8 and (line[:8] == " ==> HIT" or line[:8] == " ==> TRY"):
            break
         else:
            #blank lines and others that we will ignore?
            line = fp.readline()

      return line

   def WriteOutTable (self, openFile, pre):
      prefix = self.GetTablePrefix (pre)

      for match in self.matches:
         match.WriteOutTable (openFile, prefix)

   def GetTablePrefix (self, pre):
```

```
      return pre + " %s %s" % (self.title, self.fileName)

   def GetBestRMSDMatch(self, useDistanceRmsd = 0):
      best = None
      if useDistanceRmsd:
         for match in self.matches:
            if not best or match.distanceRmsd < best.distanceRmsd:
               best = match
      else:
         for match in self.matches:
            if not best or match.rmsd < best.rmsd:
               best = match
      return best


class SpasmMatch:
   def __init__(self, search):
      self.id = None
      self.rmsd = None
      self.scRmsd = None
      self.caRmsd = None
      self.distanceRmsd = None
      self.numatoms = None
      self.scMatrix = None
      self.caMatrix = None
      self.search = search
      self.chain = ''

   def ReadOpenFile (self, idLine, fp):
      idwords = idLine.split()
      self.id = idwords[2]

      assert (idwords[4] == "RMSD")
      self.rmsd = float(idwords[5])
      self.numatoms = int (idwords[8])

      #load the diff matrices
      line = fp.readline()
      while (1):
         words = line.split()
         if len(words) > 3 and words[3] == "matrix":
            matrix = SpasmMatrix()
            if words[1] == "SC":
               if words[0] == "Target":
                  if self.search.scMatrix == None:
                     self.search.scMatrix = matrix
                  else:
                     matrix.dummy = 1
               else:
                  self.scMatrix = matrix
            elif words[1] == "CA":
               if words[0] == "Target":
                  if self.search.caMatrix == None:
                     self.search.caMatrix = matrix
                  else:
                     matrix.dummy = 1
               else:
                  self.caMatrix = matrix
```

```
            line = matrix.ReadOpenFile (fp)
         elif line == '' or (len(words) > 1 and (words[0] == "MATCH" or words[0] == "==>")):
            break
         else: #skip blank lines and other unexpected things
            line = fp.readline()
      if not self.search.scMatrix:
         print "WARNING! Did not read a scMatrix for : %s" % (self.id)
      if not self.search.caMatrix:
         print "WARNING! Did not read a caMatrix for : %s" % (self.id)

      #calculate distance rmsd
      if self.scMatrix and self.search.scMatrix:
         self.scRmsd = self.scMatrix.CalculateRMSD (self.search.scMatrix)
      if self.caMatrix and self.search.caMatrix:
         self.caRmsd = self.caMatrix.CalculateRMSD (self.search.caMatrix)

      if self.scRmsd != None and self.caRmsd != None:
         self.distanceRmsd = (self.scRmsd + self.caRmsd)/2
      else:
         self.distanceRmsd = self.scRmsd
      if self.distanceRmsd == None:
         self.distanceRmsd = self.caRmsd
      if self.distanceRmsd == None:
         self.distanceRmsd = 9.99

      return line

   def CalcDiffs (self):
      self.scDiffMatrix = SpasmMatrix()
      self.caDiffMatrix = SpasmMatrix()

      if self.scMatrix and self.search.scMatrix:
         self.scDiffMax = self.scDiffMatrix.SetAsDifference(self.search.scMatrix, self.scMatrix)
      else:
         self.scDiffMax = 0.0

      if self.caMatrix and self.search.caMatrix:
         self.caDiffMax = self.caDiffMatrix.SetAsDifference(self.search.caMatrix, self.caMatrix)
      else:
         self.caDiffMax = 0.0

   def WriteOutTable (self, outFile, pre):
      self.CalcDiffs()
#     if self.distanceRmsd != None:
#        calcRmsd = self.distanceRmsd
#     else:
#        calcRmsd = -1.0
      calcRmsd = self.distanceRmsd

```

```
      outFile.write ("%6s %3s %3s %5.2f %5.2f %3d %4.1f %4.1f RES: " %
                     (pre, self.id, string.join(self.caMatrix.GetChains(), ''),
                         self.rmsd, calcRmsd, self.numatoms, self.caDiffMax, self.scDiffMax))

      outFile.write (self.caMatrix.OneLineResidues())

      outFile.write ("\n")

class SpasmMatrix:
   def __init__(self):
      self.dummy = 0

   def ReadOpenFile (self, fp):
      if not self.dummy:
         self.rows = []
         self.residues = []
      numRows = -1
      totalRows = 0
      while (numRows < totalRows):
         line = fp.readline()
         if not self.dummy:
            res = SpasmResID (line[1:11])
            row = map (float,  line[12:].split() )
            self.rows.append (row)
            self.residues.append (res)

         if not totalRows:  #we're on the first row, so set number of rows
            totalRows = len(line[12:].split())
            numRows = 1
         else:
            numRows = numRows + 1

      return fp.readline()

   def GetChains (self):
      chains = []
      for res in self.residues:
         if res.chain in chains:
            continue
         else:
            chains.append (res.chain)
      if len (chains) == 0:
         chains.append ('-')
      return chains

   def SetAsDifference (self, reference, other):
      assert(len (reference.rows) == len (other.rows))
      self.rows = []
      self.residues = other.residues
      maxDiff  = -1
      for i in range (len (other.rows)):
         row = []
         for j in range (len (other.rows)):
            diff = other.rows[i][j] - reference.rows[i][j]
```

```
            maxDiff = max (maxDiff, abs(diff))
            row.append (diff)
         self.rows.append (row)
      return maxDiff

   def OneLineResidues  (self):
      return string.join (map (lambda a: "%s_%s_%s"%(a.type, a.chain, a.position), self.residues), " ")

   def CalculateRMSD (self, otherMatrix):
      i = 0
      sumSquareDevs=0
      numSquareDevs=0
      matSize = len (self.rows)
      assert (matSize == len (otherMatrix.rows))
      for i in range (0, matSize):
         for j in range (i+1, matSize):
            dev = self.rows[i][j] - otherMatrix.rows[i][j]
            sumSquareDevs = sumSquareDevs +  (dev * dev)
            numSquareDevs = numSquareDevs + 1
      return math.sqrt (sumSquareDevs/numSquareDevs)


class SpasmResID:
   def __init__ (self, idString):
      #string is "HIS A 339" or "HIS_A_339"
      self.type = string.strip (idString[:3])
      if idString[4] != " ":
         self.chain = string.strip (idString[4])
      else:
         self.chain = string.strip ('-')
      if idString[5] == '_':
         self.position = string.strip (idString[6:])
      else:
         self.position = string.strip (idString[5:])


def dummy___runFileStrings ():
   pass

##################################################################
# Formalized contents of a spasm *.com run file:
#
#
#
# use like:
# spasmRunFileString % (
#           spasmBinaryPath,
#           maxhits,
#           libpath,
#           motifPath,
#           fourLetterCode,
#           maxRMSD,
#           maxCADiff,
#           MaxSCDiff,
#           maxResolution,
#           maxResidues,
#           substitutionsAllowed,   # 5 for user defined
```

```
#           substitutionsString)    #multi line
#
#
####################################################################
runFileStringSTDOUT = """#automatically generated by Spasm.py
%s maxhits %d<<EOI
%s
%s
%s
%f
%f
%f
%f
%d
%d
%s
n
n
n
y
n
n
1
n
n
n
n
n
EOI
"""


runFileStringSTDOUT_scOnly = """#automatically generated by Spasm.py
%s maxhits %d<<EOI
%s
%s
%s
%f
%f
%f
%f
%d
%d
%s
n
n
n
y
n
n
2
n
n
n
n
n
EOI
"""
```

```
def Convert2SpasmFilesToSortedAndScoredTable (trueSpasm, falseSpasm, tableFile, trueHash = None, useDistanceRmsd=0, returnString = 0):
   fp = open (trueSpasm)
   trueSearch = SpasmSearch (1)
   trueSearch.ParseSpasmHits (fp)
   fp.close()

   fp = open(falseSpasm)
   falseSearch = SpasmSearch (1)
   falseSearch.ParseSpasmHits (fp)
   fp.close()


   return Convert2SpasmSearchesToSortedAndScoredTable (trueSearch, falseSearch, tableFile, trueHash, useDistanceRmsd, returnString)


def Convert2SpasmSearchesToSortedAndScoredTable (trueSearch, falseSearch, tableFile, trueHash = None, useDistanceRmsd=0,
                                                 returnString = 0, writeFile=1, trueSkipHash=None, falseSkipHash=None):
   if not trueHash:
      trueHash = {}
   if not trueSkipHash:
      trueSkipHash = {}
   if not falseSkipHash:
      falseSkipHash = {}

   if returnString:
      stringFile = StringIO.StringIO()
      fp = stringFile
   else:
      fp = open (tableFile, "w")

   trueMatches = trueSearch.GetBestMatchesPerHit(useDistanceRmsd)
   if falseSearch:
      falseMatches = falseSearch.GetBestMatchesPerHit(useDistanceRmsd)
   else:
      falseMatches = []
   trueIndex = 0
   for fm in falseMatches:
      while (1):
         if useDistanceRmsd:
            if not (trueIndex < len (trueMatches) and trueMatches[trueIndex].distanceRmsd < fm.distanceRmsd):
               break
         else:
            if not (trueIndex < len (trueMatches) and trueMatches[trueIndex].rmsd < fm.rmsd):
               break

         if trueSkipHash.has_key (trueMatches[trueIndex].hit.title):
            pre = "*"
         else:
            pre = "1"

```

```
         trueMatches[trueIndex].WriteOutTable (fp, trueMatches[trueIndex].hit.GetTablePrefix(pre))
         trueIndex += 1

      name = fm.hit.title

      #check that the false match isn't one that is marked as true (lets me not have to remove these from false library)
      if trueHash.has_key (name):
         continue
#
#     try:
#        trueHash[name]
#        continue
#     except KeyError:
#        pass

      if falseSkipHash.has_key (name):
         pre = "#"
      else:
         pre = "0"
      fm.WriteOutTable (fp, fm.hit.GetTablePrefix(pre))
   #finally write any remaining true positives, important in cases where no false positives were hit
   while trueIndex < len (trueMatches):
      if trueSkipHash.has_key (trueMatches[trueIndex].hit.title):
         pre = "*"
      else:
         pre = "1"

      trueMatches[trueIndex].WriteOutTable (fp, trueMatches[trueIndex].hit.GetTablePrefix(pre))
      trueIndex += 1

   if returnString:
      compressFileName = tableFile + ".gz"
      fp = gzip.open (compressFileName, "w")
      if writeFile:
         fp.write (stringFile.getvalue())
      else:
         #still have to maintian file for how gasps checks which motifs its tried already!
         fp.write (":-)")
      fp.close
      return stringFile.getvalue()
   else:
      fp.close()

def ComputeAreaFromTableFile (filePath, maxFalse, fileString = '', useDistanceRMSD = 0):
   if fileString:
      fp = StringIO.StringIO (fileString)
   else:
      fp = open (filePath)
   numFalse = 0
   scoreHash = {}
   #first load the lines we need in to scoreHash
   while (numFalse < maxFalse):
```

```
      line = fp.readline()
      if line == '':
         break
      if line[0] == '#':
         continue
      elif line[0] == '*':
         continue
      elif line[0] == '0':
         numFalse = numFalse + 1
      if useDistanceRMSD:
         score = float (line.split()[6])
      else:
         score = float (line.split()[5])
      try:
         scoreHash[score].append (line[0])
      except KeyError:
         scoreHash[score] = [line[0]]
   fp.close()
   numFalse = 0
   numTrue = 0
   area = 0
#  print scoreHash
   #next calculate area one false positive at a time.
   # if one score has several hits, false postives are counted before true positives.
   scores = scoreHash.keys()
   scores.sort()
#  print scores
   for score in scores:
      hits = scoreHash[score]
      newTrue = 0
      for hit in hits:
         if hit == '1':
            newTrue = newTrue + 1
         elif hit == '0':
            numFalse = numFalse + 1
            area = area + numTrue
      numTrue = numTrue + newTrue

      if numFalse > maxFalse:
         break

   #if we don't have enough false positives assume all other hits would be false.
   while (numFalse < maxFalse):
      print "Assuming a false positive"
      area = area + numTrue
      numFalse = numFalse + 1

   return area

def ComputeSeparationScoreFromTableFile3 (filePath, maxFalse, maxRMSD, maxTrue, sepScoreImportance, useDistanceRmsd, fileString = ''):

   if fileString:
      fp = StringIO.StringIO (fileString)
   else:
```

```
      fp = open (filePath)
   numFalse = 0
   trueScores = []
   falseScores = []
   #read the relevant scores for true and false positives
   while (numFalse < maxFalse):
      line = fp.readline()
      if line == '':
         break
      if line[0] == '#':
         continue
      elif line[0] == '*':
         continue
      elif line[0] == '0':
         numFalse += 1
         if useDistanceRmsd:
            falseScores.append(float(line.split()[6]))
         else:
            falseScores.append(float(line.split()[5]))
      elif line[0] == '1':
         if useDistanceRmsd:
            trueScores.append (float(line.split()[6]))
         else:
            trueScores.append (float(line.split()[5]))

   #get score to assign to false positives that don't show up in list.
   if maxRMSD:
      dummyFalseScore = maxRMSD
   else:
      dummyFalseScore = 0
      if trueScores:
         dummyFalseScore = trueScores[-1]
      if falseScores:
         dummyFalseScore = max (dummyFalseScore, falseScores[-1])
   dummyFalseScore += 0.01

   #fill out false positive list
   while len(falseScores) < maxFalse:
      falseScores.append (dummyFalseScore)

   #remove trueScores that could just as easily have come after the last false positive because they score equivalently
   while (falseScores and trueScores and trueScores[-1] == falseScores[-1]):
      del (trueScores[-1])


   #calculate medianFP, relies on falseScores being sorted already
   if len(falseScores) % 2:
      #odd number just grab central scores
      medianFP = falseScores[len(falseScores)/2]
   else:
      #even number, take center of two central scores
      medianFP = falseScores[len(falseScores)/2] + falseScores[len(falseScores)/2-1]
      medianFP = medianFP/2

```

```

   #calculate meanFP while we also calculate rocArea
   meanFP = 0.0
   rocArea = 0.0
   for fs in falseScores:
      meanFP += fs/maxFalse
      for ts in trueScores:
         if ts < fs:
            rocArea += 1.0
   #normalize to 1
   rocArea = rocArea/(maxFalse*maxTrue)


   if trueScores:
      #calculate medianTP
      if len (trueScores ) % 2:
         medianTP = trueScores[len(trueScores)/2]
      else:
         medianTP = trueScores[len(trueScores)/2] + trueScores[len(trueScores)/2-1]
         medianTP = medianTP/2

      sepScore = (medianFP - medianTP)/medianFP  # max 1, min 0
      if sepScore < 0:
         sepScore = 0


      maxSepScore = (maxRMSD - 0.0) * len (trueScores) #potentially undesirable; should use max number of trueScores instead?

      #normalize:
      sepScore = sepScore * sepScoreImportance

   else:
      sepScore = 0

   return sepScore + rocArea

```

polacco/XML.py

```

#   __  ____  __ _
#   \ \/ /  \/  | |    _ __  _   _
#    \  /| |\/| | |   | '_ \| | | |
#    /  \| |  | | |___ _| |_) | |_| |
#   /_/\_\_|  |_|_____(_) .__/ \__, |
#                        |_|    |___/
#

# A VERY simple XML module.  This was developed specifically to read XML files as
# output by ncbi blastall or blastpgp programs, and processed by BlastXML.py
# Many xml features may be poorly dealt with, if at all, if they do not occur in
```

```
# the output of the ncbi programs. Consider yourself warned if you try to use this
# as a full-featured XML parser.



import string
class XML_node:
   def __init__ (self):
      self.name = None
      self.subNodes = {}
      self.variables = {} #not common in blast output if any, these are name value pairs within the first < >
      self.value = None
      self.parentNode = None

   # builtins to get at the value :
   def __int__(self):
      return int (self.value)
   def __str__(self):
      return self.value
   def __float__(self):
      return float (self.value)


   def keys(self):
      return self.subNodes.keys()

   def LoadFromOpenFile (self, openFileIn, startTag):
      words = startTag[1:-1].split()

      #get name from opening tag
      self.name = words[0]

      #get variableStrings -- this can be elaborated if anybody uses them
      self.variableStrings = words[1:]
      if self.variableStrings:
         for vs in self.variableStrings:
            name,value = vs.split("=")
            self.variables[name] = value[1:-1] #remove thequotes

      #read value if any
      valueBuffer = []
      char = ''
      while char != '<':
         valueBuffer.append (char)
         char = openFileIn.read(1)
         if char == "":
            raise "Unexpected end of XML file %s" % (self.name)
      self.value = string.join (valueBuffer, '')

      #now start reading subnodes
      tagType = ''
      while tagType != '/':
         tagType, tag = readXMLTag (openFileIn, char)
         char = ''
         if tagType == '':
```

```
            nextNode = XML_node()
            nextNode.LoadFromOpenFile (openFileIn, tag)
            nextNode.parentNode = self
            try:
               self.subNodes[nextNode.name].append (nextNode)
            except KeyError:
               self.subNodes[nextNode.name] = [nextNode]
         elif tagType != '/':
            raise "Unexpected tag type %s in middle of file (%s)" %
(tagType, self.name)

      # tagType and tag should correspond to the closing tag for the
current node.  verify it!
      closingName = tag[2:-1]
      if closingName != self.name:
         raise "closingName doesn't match self.name: %s != %s" %
(closingName, self.name)


def readXMLTag (openFileIn, firstChar = ''):
   tagBuffer= [firstChar]
   type = ''

   #first get opening carrot if we need to
   if tagBuffer[-1] == '<':
      started = 1
   else:
      char = ''
      while (char != '<'):
         char = openFileIn.read(1)
         if char =="":
            raise "Unexpected end of XML file."
      tagBuffer.append (char)

   #get special first characters
   char = openFileIn.read(1)
   if char in '/?!':
      type = char
   else:
      type = ''

   tagBuffer.append (char)
   done = 0
   #get rest of tag
   while (not done):
      char = openFileIn.read(1)
      if char == '>':
         done = 1
      if char in '\n\r':
         continue
      tagBuffer.append (char)

   tag = string.join (tagBuffer, '')
   #print tag

   return type, tag


```

```
class XML_tree:
   def __init__ (self, openFileIn):
      self.rootNode = None
      self.xmlVersionString = None
      self.doctype_string = None

      if not openFileIn:
         return

      while(1):
         char = openFileIn.read(1)
         if char == '<':
            type,tag = readXMLTag (openFileIn, char)
         if type != '':
            if type == '?':
               self.xmlVersionString = tag[2:-2]
            elif type == '!':
               self.doctype_string = tag[2:-1]
            else:
               raise "Unexpected tag type at start of file: %s" % type

         else:  #found the start of the root node
            self.rootNode = XML_node()
            self.rootNode.LoadFromOpenFile (openFileIn, tag)
            break




def test():
   f = open ("longtest.xml")
   xtree = XML_tree (f)
```

polacco/utils.py

```
###################################################################3
#                    _                            _   _ _
#        _ __   ___ | | __ _  ___ ___ ___  _   _| |_(_) |___
#       | '_ \ / _ \| |/ _` |/ __/ __/ _ \| | | | __| | / __|
#       | |_) | (_) | | (_| | (_| (_| (_) | |_| | |_| | \__ \
#       | .__/ \___/|_|\__,_|\___\___\___(_)__,_|\__|_|_|___/
#       |_|
####################################################################

import string



# This function will read a sequence in FASTA format up to and including the '>'
# character signifying the start of the next sequence.  If I knew how to peek
# at this of put it back on the stream I would.  If the first non-white character
```

```
# in the file is not '>' this function assumes that line is the name of the sequence.
# This never returns the first '>' on a line.  If for some reason you have two, then you can expect one.

def GetNextFASTASeq  (seqFile):
   firstChar = " "
   while firstChar in string.whitespace:
      firstChar = seqFile.read (1)
      if firstChar == '':
         return ('', '')
   if firstChar != '>':
      name = string.rstrip (firstChar + seqFile.readline())
   else:
      name = string.strip (seqFile.readline())
   parts = []
   firstChar = seqFile.read(1)
   while firstChar not in ( '>', ''):
      parts.append (string.strip (firstChar + seqFile.readline()))
      firstChar = seqFile.read(1)
   seq = string.join (parts, '')
   return (name, seq)


def WriteFastaSeq (fileName, seqName, sequence):
   fp = open (fileName, "w")
   fp.write (">%s\n%s\n" % (seqName, sequence))
   fp.close()


aa3to1 = {
          "PHE" : 'F',
          "ILE" : 'I',
          "LEU" : 'L',
          "VAL" : 'V',
          "PRO" : 'P',
          "ALA" : 'A',
          "GLY" : 'G',
          "MET" : 'M',
          "CYS" : 'C',
          "TRP" : 'W',
          "TYR" : 'Y',
          "THR" : 'T',
          "SER" : 'S',
          "GLN" : 'Q',
          "ASN" : 'N',
          "GLU" : 'E',
          "ASP" : 'D',
          "HIS" : 'H',
          "LYS" : 'K',
          "ARG" : 'R',
          "GAP" : '-',
          # non standard that we might run in to:
          "MSE" : 'M',
          "MME" : 'M',
          "???" : 'X'

           }
```

```
aa1to3 = {
          'F' : "PHE",
          'I' : "ILE",
          'L' : "LEU",
          'V' : "VAL",
          'P' : "PRO",
          'A' : "ALA",
          'G' : "GLY",
          'M' : "MET",
          'C' : "CYS",
          'W' : "TRP",
          'Y' : "TYR",
          'T' : "THR",
          'S' : "SER",
          'Q' : "GLN",
          'N' : "ASN",
          'E' : "GLU",
          'D' : "ASP",
          'H' : "HIS",
          'K' : "LYS",
          'R' : "ARG",
          '-' : "GAP"
      }

def AA3to1 (aaa):
   return aa3to1[string.upper(aaa)]

def AA1to3 (a):
   return aa1to3[string.upper (a)]

def SeqAA3to1 (aaas):
   as = []
   for aaa in aaas:
      as.append (AA3to1 (aaa))
   return as

def SeqAA1to3 (as):
   aaas = []
   for a in as:
      try:
         aaas.append (AA1to3 (a))
      except KeyError:
         aaa = a + a + a
         aaas.append (aaa)
   return aaas
```

test/astral_1.65_SF.lib (partial)

```
! Created by MKSPAZ V. 040618/2.3.3 at Thu Jan 27 10:13:05 2005 for ben
!
PRO A6M_
PDB /Users/ben/tmp_NoBackup/pdbstyle-1.65/a6/d1a6m__.ent
RES   1.00
CMP
VAL     1  -3.526 15.758 14.900 -4.746 16.634 16.149
```

```
LEU      2  -0.689 14.190 16.862 1.731 14.740 15.905
SER      3  -1.487 12.495 20.143 -1.027 10.685 20.278
GLU      4  0.324 13.366 23.335 -0.274 15.439 25.046
GLY      5  2.196 10.084 23.022 2.196 10.084 23.022
GLU      6  3.317 10.981 19.508 1.056 10.123 17.224
TRP      7  4.502 14.431 20.597 2.977 17.247 18.818
GLN      8  6.475 12.812 23.418 7.581 10.572 24.982
LEU      9  8.296 10.604 20.915 7.311 8.231 20.426
VAL     10  9.019 13.628 18.670 7.839 14.702 17.503
LEU     11  10.311 15.860 21.464 8.655 17.799 21.829
HIS     12  12.315 13.068 23.090 13.323 10.588 24.655
...
GLY A  74  1.747 -16.568 2.419 1.747 -16.568 2.419
ARG A  75  4.859 -15.859 0.292 3.334 -18.781 -0.441
VAL A  76  6.216 -12.816 -1.561 6.009 -11.548 -0.047
GLU A  77  9.002 -11.725 -3.902 9.730 -10.534 -6.922
ARG A  78  12.003 -10.462 -1.910 15.682 -10.880 -2.598
SER A  79  11.298 -6.719 -1.500 9.563 -6.804 -2.269
END
!
! total residues 204251
! total proteins   1247
```

test/d2mnr_1.fasta

```
>d2mnr_1 c.1.11.2 (133-359) Mandelate racemase {Pseudomonas putida}
RESOLUTION: 1.900000
pvqaydshsldgvklateravtaaelgfravktkigypaldqdlavvrsirqavgddfgimvdynqsl
dvpaaikrsqalqqegvtwieeptlqhdyeghqriqsklnvpvqmgenwlgpeemfkalsigacrlam
pdamkiggvtgwirasalaqqfgipmsshlfqeisahllaatptahwlerldlagsvieptltfeggn
avipdlpgvgiiwrekeigkylv
```

test/d2mnr_1.fasta.psiblast.xml.faln (partial)

```
>d2mnr_1/001
PVQAYD----S-H---S-LDGVKLATE---RA-VTA---A---EL---------GFRAV-KT-----
KI------G------------------YPA---------L------------------DQ--------
-------DL----------------AVVR-----------------SIRQAV----------GDDF--
G---I-----MVD-Y------------------NQS-L------D-------V-P-------AA-
IKRSQAL-QQ---E-------G---VT------W--IEE--PT-LQ----HD----YEGHQ------
R------I-QSKL---N-----V-P-----VQM-GE-NW--L---GP-------------------E-
EMFKA-LS-IGAC----RL--AMPDAMKIGGVTGWIRASALAQ--Q--FG---I-P-M------S-S-
----H----------LF------Q-------E------IS-------AHL-----LA---------
AT-----PT------A----------H----W----LE----R-------------L-----------
-------DL----A----------G-----SV--I-----E--P--------------T------
LTFE-G-G--N--AV----I-P--D--LP--GVGIIWREKEIGKYLV
>886661/143
---AWT----L-A---S-GDTARDIAE---AE-QML---E---AR---------RHRIF-KL-----
KI------G------------------ANP---------L------------------EQ--------
-------DL----------------KHVV-----------------AIKKAL----------GERA--
S---V-----RVD-V------------------NQY-W------D-------E-S-------QA-
IRGCRVL-GD---N-------G---ID------L--IEQ--PI-SR----VN----RSGQI------
R------L-NQRS---L-----A-P-----IMA-DE-SI--E---SV-------------------E-
DAFSL-AA-DGAA----SV--FALKIAKNGGPRAVLRTAQIAE--A--AG---I-A-L------Y-G-
----G----------TM------L-------E------GS-------VGT-----LA---------
```

```
SA-----HA------FLTLRQLTWDTE----L----FG----P-------------L-----------
-------LL----T----------E-----DI--V-----T--E--------------R------
PQYR-D-F--H--LH----I-P--R--TP--GLGLTLDEERLARFR-
...
```

test/d2mnr_1.pdb (partial)

```
HEADER    SCOP/ASTRAL domain d2mnr_1 [29245]      21-NOV-03   0000
REMARK  99
REMARK  99 ASTRAL ASTRAL-version: 1.65
REMARK  99 ASTRAL SCOP-sid: d2mnr_1
REMARK  99 ASTRAL SCOP-sun: 29245
REMARK  99 ASTRAL SCOP-sccs: c.1.11.2
REMARK  99 ASTRAL Source-PDB: 2mnr
REMARK  99 ASTRAL Source-PDB-REVDAT: 31-JAN-94
REMARK  99 ASTRAL Region: 133-359
REMARK  99 ASTRAL ASTRAL-SPACI: 0.52
REMARK  99 ASTRAL ASTRAL-AEROSPACI: 0.52
REMARK  99 ASTRAL Data-updated-release: 1.61
ATOM    954  N   PRO   133      26.117  28.195  19.354  1.00 16.42
2MNR1142
ATOM    955  CA  PRO   133      26.550  27.070  20.183  1.00 17.39
2MNR1143
ATOM    956  C   PRO   133      25.646  25.856  19.887  1.00 17.79
2MNR1144
ATOM    957  O   PRO   133      24.432  26.050  19.690  1.00 15.27
2MNR1145
ATOM    958  CB  PRO   133      26.423  27.575  21.607  1.00 16.88
2MNR1146
ATOM    959  CG  PRO   133      26.217  29.084  21.431  1.00 18.48
2MNR1147
ATOM    960  CD  PRO   133      25.391  29.164  20.163  1.00 16.15
2MNR1148
ATOM    961  N   VAL   134      26.203  24.632  19.793  1.00 14.37
2MNR1149
ATOM    962  CA  VAL   134      25.376  23.431  19.550  1.00 13.11
2MNR1150
ATOM    963  C   VAL   134      25.512  22.524  20.760  1.00 10.74
2MNR1151
ATOM    964  O   VAL   134      26.639  22.219  21.178  1.00 10.65
2MNR1152
ATOM    965  CB  VAL   134      25.822  22.626  18.293  1.00 14.98
2MNR1153
ATOM    966  CG1 VAL   134      24.813  21.481  18.039  1.00 15.66
2MNR1154
ATOM    967  CG2 VAL   134      25.879  23.546  17.071  1.00 18.17
2MNR1155
```

test/enolase.lib (partial)

```
! Created by MKSPAZ V. 040618/2.3.3 at Wed Jul 20 10:14:20 2005 for
ben
!
PRO ONEA
PDB scopc.1.11/d1onea1.pdb
RES   1.80
```

```
CMP
SER A 142  3.954 -21.471 10.866 3.132 -23.070 10.341
PRO A 143  6.317 -20.493 9.738 6.437 -19.927 7.955
TYR A 144  8.244 -18.231 11.988 8.330 -19.865 15.458
VAL A 145  9.753 -15.108 10.378 8.597 -13.779 9.428
LEU A 146  13.368 -14.050 10.947 14.866 -16.299 11.239
PRO A 147  14.071 -10.342 10.694 14.559 -10.674 12.458
VAL A 148  16.213 -8.369 8.330 15.891 -7.404 6.671
PRO A 149  18.854 -6.741 10.645 19.911 -8.202 10.014
PHE A 150  19.145 -2.899 10.152 16.286 -2.033 8.574
```

test/enolase.list

```
d1onea1
d1kkoa1
d1fhua1
d1muca1
d1ec7a1
d2mnr_1
d1jpma1
```

Appendix 2: GASPSdb CGI scripts

This appendix contains the scripts written in the python programming language that ran on the UCSF Resource for Biological Visualization and Informatics web server. The GASPSdb file contained the vast majority of the code to run the backend and interface with the database. The jsonMotif file's sole responsibility was to describe a requested motif so that the pages delivered by GASPSdb could show a motif in a popup window.

GASPSdb

```
#!/usr/local/bin/python2.4



import cgi, urllib
import sys, tempfile, os, string, stat
sys.path.insert(0, "/mol/sfld/gaspsdb/py.packages")
import polacco.Rigor

def LibraryLookup():
   libs = {"scop3" :"/mol/sfld/gaspsdb/lib/scop3.emp.25and40.rig",
           "scop4" :"/mol/sfld/gaspsdb/lib/scop4.emp.25and40.rig",
           "GO"    :"/mol/sfld/gaspsdb/lib/go.emp.25and40.rig",
           "goScop":"/mol/sfld/gaspsdb/lib/goScop.emp.25and40.rig"}
   return libs

def DrawMainForm():
   print"""
   <!-- DrawForm -->
   <p> This search relies on <a href="/references.html">RIGOR</a> which
   is freely provided by its creator, Gerard Kleywegt, for use by private
   individuals, schools, academics, and not-for-profit institutions.
   <strong>The use of this search by others is not allowed.</strong> Contact us
   if you wish to use our motifs, and contact Gerard Kleywegt if you wish to
   use <a href="/references.html">RIGOR</a>.

   <table border=1>
   <tr><th> Coordinates in PDB format:</th><th>Select a library</th><th>Click to start search</th></tr>
   <form action="./GASPSdb" enctype="multipart/form-data" method="post">
   <input type="hidden" name="do" value="rigor">
   <tr>
```

```
   <td>Enter a PDB code (e.g., '2mnr')<input type = "text" name="pdb" size = "6">
   <br>or select a file to upload:
   <br><input type="file" name="fileUpload" size="20"></td>
   <td>
   <input name="lib" type="radio" value="scop3"> SCOP Superfamilies
   <br> <input name="lib" type="radio" value="scop4"> SCOP Families
   <br> <input name="lib" type="radio" value="GO"> Gene Ontology (GO)
   <br> <input name="lib" type="radio" value="goScop"> SCOP Superfamilies / GO
   </td
   <td>
   <input type="submit" value="Send">
   </td></tr>
   </form>
   </table>
   """


def DrawKeyWordSearchForm(instructions=True):
   if instructions:
      print"""
      <p> To look for a specific group, enter keywords below. Separate words are automatically joined with 'AND'.
      """
   print"""
   <p><form action="./GASPSdb" method="get">
   <input type="hidden" name="do" value="keySearch">
   <input name="keywords" type="text" size="40">
   <button type="submit" id="Search">Search</button>
   </form>
   """
def DrawStructSearchForm (instructions = True):
   if instructions:
      print"""
      <p> To find groups that contain a structure, enter its PDB id below. Multiple IDS are automatically joined with 'OR'.
      """
   print"""
   <p><form action="./GASPSdb" method="get">
   <input type="hidden" name="do" value="structSearch">
   <input name="structs" type="text" size="40">
   <input name="hideMotifs" type="hidden" value="TRUE">
   <button type="submit" id="Search">Search</button>
   </form>
   """

def DrawSearchPage ():
      print "<h2> Find motifs that match your structure.</h2>"
      DrawMainForm()
      print "<hr>"
      print "<h2> Find groups by key word.</h2>"
      DrawKeyWordSearchForm()
      print "<hr>"
      print "<h2> Find groups and motifs by PDB ID</h2>"
      DrawStructSearchForm()

```

```
def GetLocalPDBFile(pdbCode):
   pdbCode = string.lower (pdbCode)
   fp = None
   if len (pdbCode) == 4:
      path = "/databases/mol/pdb/%s/pdb%s.ent" % (pdbCode[1:3], pdbCode)
      try:
         fp = open (path)
      except IOError:
         fp = None
   else:
      print "<p> PDB codes must be four characters long, '1one', for example."
   if not fp:
      print "<p> There was an error retrieving the coordinates for pdb %s. Please try to upload the coordinates." % pdbCode
      return None
   else:
      return fp

def GetFile(form):
   pdbFile = None
   pdbCode = form.getfirst ("pdb", "")
   if pdbCode:
      pdbFile =  GetLocalPDBFile (pdbCode)
      fileName = pdbCode
   if not pdbFile:
      try:
         uploadedFile = form["fileUpload"]
         fileName = uploadedFile.filename
         pdbFile = uploadedFile.file
      except KeyError:
         print "<p> No file uploaded."
         return None
   lib = form.getfirst ("lib", "")
   if pdbFile:
      #set up a temporary file to write the uploaded file to:
      fd, localFileName = tempfile.mkstemp (prefix= fileName + "_",suffix = "_"+lib,  dir="/var/tmp/gaspsdb/rigorRuns")
      lineCount = 0
      atomCount = 0
      model = 0
      endmdl = 0
      for line in pdbFile:
         if line[0:6] == "ATOM  ":
            atomCount+=1
         lineCount +=1
         if line[0:6] == "MODEL ":
            model += 1
         if model and line[0:6] == "ENDMDL":
            endmdl = 1
         if not (model and endmdl):
            os.write (fd, line)
      os.close (fd)
      os.chmod (localFileName, stat.S_IROTH | stat.S_IRUSR | stat.S_IRGRP )
      if atomCount:
```

```
         print "<p>Received %d lines describing %d atoms from %s" %
(lineCount, atomCount, fileName)
      if model and endmdl:
         print "<p>Uploaded file contained %d models, only the first
model will be searched." % (model)
      if atomCount < 100:
         print "<p><b>Warning:</b> The uploaded file (%s) seemed to
only contain %d atom records. This may not produce usable results."
% (uploadedFile.filename, atomCount)
      return localFileName
   else:
      print "<p> fileUpload did not appear to be a file."
      return None

def HTMLRedirectHead_DoRigor (code):
   print """
   <html>
   <meta  HTTP-EQUIV="REFRESH" content="0;
url=http://babbittlab.ucsf.edu/cgi-
bin/GASPSdb?do=retrieveResults&code=%s">
   </html>
   """ % code

def DrawFormatResultsForm (code = None, first=1, last=100):
   print """
   <form action="./GASPSdb" enctype="application/x-www-form-
urlencoded" method="get">
   <table width=600 border=1>
   <tr><th> Click to format the results.</th><th> To view results
from a previous run, enter its code below.</th><tr>
   <input type="hidden" name="do" value="retrieveResults">
   <tr><td align="center"><input type="submit" value="Get
Results!"></td>
   <td> <input type="text" name="code" size="40" value="%s"></td>
   </tr>
   <tr>
      <td colspan="2" align="center">
      <strong>Options:</strong>
      </td>
   </tr>
   <tr>
    <td> Which hits:</td>
    <td><table>
      <tr><td>First: <input type="text" name="first" size="8"
value="%d"></textarea></td><td>Last: <input type="text" name="last"
size="8" value="%d"><small> Enter 'all' to show all.</small> </tr>
      </table>
    </td>
   </tr>
   </table>
   </form>
   """ % (code, first, last)


def DoRigor(form):
   localUploadFile = GetFile(form)
   if not localUploadFile:
#     HTMLHead()
```

```
#     DrawNavBar (form)
      print "<p> There was an error storing your uploaded file. Can't run rigor."
      return

   lib = form.getfirst ("lib")
   if not lib:
      print "<p> You must select a motif library."
      return
   libraryPath = LibraryLookup()[form.getfirst("lib")]
   rigorPath = "/mol/sfld/gaspsdb/al_rigor"
   #files that will be written to by Rigor.py
   rigorErr = localUploadFile + ".err"
   rigorOut = localUploadFile + ".out"
   rigorRun = localUploadFile + ".run"
   pid = polacco.Rigor.RigorRun (localUploadFile, libraryPath, rigorPath, rigorErr,
                                rigorOut, rigorRun, useSubprocess=True)
   #print "<p> rigor run started with pid %d." % pid
   print "<p> Searching for matching motifs...<p> This run has been given the code: <b>", os.path.split(localUploadFile)[1], "</b>"
   DrawFormatResultsForm (os.path.split (localUploadFile)[1])
#  HTMLRedirectHead_DoRigor (os.path.split (localUploadFile)[1])
#  DrawNavBar(form)
#  print "Starting RIGOR run with code %s" % (os.path.split (localUploadFile)[1])

class TooManyHits (Exception):
   pass






def FixGroupName (group):
   #two choices, the second char is a "." then this is a scop id
   #(i.e., a.1.2.3) no need to fix
   #otherwise the first is a number of a go id and needs a couple of zeros prepended to it
   if group[1] == ".":
      return group
   elif group[0] in "0123456789":
      return "000" + group



def DrawRigorHitsTable (hits, lib, code,first=0, last=None):
   scores = hits.keys()
   scores.sort()

   if first > 1:
      print '<p><strong>NOTE: Hits before hit #%d are not shown below.</strong>' % first

   ToolTipScripts()
   ToolTipDiv()
```

```

   print '<p><table class="smallTable">'

   if lib in ("scop3", "scop4", "goscop"):
      domainHead = "Domain"
   elif lib =="go":
      domainHead = "Chain"
   else:
      domainHead = "Structure"

   print
'<th>#</th><th>E</th><th>RMSD</th><th>Group</th><th>%s</th><th>G-
score</th><th>Residues</th><th>Matches</th><th></th>' % domainHead
   rowSwitcher = TableRowGenerator()
   count = 0
   try:
      for s in scores:
         evText = "%4.1e" % s
         if s<1e-4:
            evText = Green (evText)
         elif s<5e-2:
            evText = Yellow(evText)
         else:
            evText = Red(evText)
         for h in hits[s]:
            motParts = h[0].split()
            if len (motParts) == 3:
               motGroup, motDomain, motScore = motParts
            elif len (motParts) == 2:
               motGroup, motDomain = motParts
               motScore = ""
            else:
               motGroup, motDomain, motScore = "* * *".split()

            if motScore and False: #currently broken so skip it for
now
               opString = h[5]
               imageLink = '<a
href="./MatchImage?code=%s&op=%s&group=%s&struct=%s"
target="_blank"> Image</a>'% (code, opString, motGroup, motDomain)
            else:
               imageLink = ''
            motGroup = FixGroupName(motGroup)
            groupName = motGroup
            structName = motDomain

            motGroup = '<a href
="./GASPSdb?do=describeGroup&group=%s" target="_blank">%s</a>' %
(motGroup, motGroup)
            if lib in ("scop3", "scop4"):
   #           motGroup = '<a
href="http://scop.berkeley.edu/search.cgi?ver=1.65&key=%s"
target="_blank">%s</a>' % (motGroup, motGroup)
               motDomain = '<a
href="http://scop.berkeley.edu/search.cgi?ver=1.65&amp;key=%s"
target="_blank">%s</a>' % (motDomain, motDomain)
            elif lib =="GO":
```

```
   #           motGroup = '<a href="http://www.godatabase.org/cgi-bin/amigo/go.cgi?action=replace_tree&search_constraint=terms&query=GO:%07d" target="_blank">%s</a>' % (int(motGroup), motGroup)
               pdbID = motDomain[0:4]
               chain = motDomain[4]
               motDomain = '<a href="http://www.pdb.org/pdb/navbarsearch.do?inputQuickSearch=%s" target="_blank">%s</a>%s' % (pdbID, pdbID, chain)

            elif lib == "goScop":
               scop = string.join (motGroup.split(".")[0:3], ".")
               go = int (motGroup.split (".")[3])
   #           motGroup = """ <a href="http://scop.berkeley.edu/search.cgi?ver=1.65&key=%s" target="_blank">%s </a>
   #                          <a href="http://www.godatabase.org/cgi-bin/amigo/go.cgi?action=replace_tree&search_constraint=terms&query=GO:%07d" target="_blank"> %07d</a>
   #           """ % (scop, scop, go, go)
               pdbID = motDomain[0:4]
               chain = motDomain[4]
               motDomain = '<a href="http://www.pdb.org/pdb/navbarsearch.do?inputQuickSearch=%s" target="_blank">%s</a>%s' % (pdbID, pdbID, chain)

   #        if altColor:
   #           print'<tr class="row2">'
   #           altColor = False
   #        else:
   #           print'<tr class="row1">'
   #           altColor = True
#           residues=[]
#           resTypes = h[3].split(",")
#           resNames = h[4].split(",")
#           for i in range (len (resTypes)):
#              residues.append (resNames[i]+resTypes[i])
#           resString = string.join (residues, "&res=")

            count +=1
            if count >= first:
               print TableRow (TableData(count, evText, h[1]) +
                               ToolTipGroupName_td(motGroup, groupName, structName=structName) +
                               TableData(motDomain, motScore, h[3], h[4], imageLink),
                             rowSwitcher.next())
            if last and count >= last:
               raise TooManyHits
      print "</table>"

   except TooManyHits:
      print "</table>"
      print "<strong> List of hits truncated after hit number %d </strong><br>Modify 'first' and 'last' below to see other hits" % last
      DrawFormatResultsForm(code, first, last)

```

```

def RetrieveResults (form):
   code = form.getfirst ("code")
   if not code:
      print "<p> Please enter a valid run code."
      DrawFormatResultsForm()
      return

   last = form.getfirst ("last")
   if not last:
      last = "100"
   try:
      last = int (last)
   except ValueError:
      last = 0

   first=form.getfirst ("first")
   if not first:
      first = "0"
   try:
      first = int (first)
   except ValueError:
      first = 0


   rigorErr = os.path.join ("/var/tmp/gaspsdb/rigorRuns/", code + ".err")
   try:
      ferr = open (rigorErr)
   except IOError:
      print "<p> %s does not appear to be a valid code." % code
      DrawFormatResultsForm (code)
      return
   for line in ferr:
      #if line[0:4] == "STOP":# ... Toodle pip ... statement executed"
      if line[0:18] =="... Toodle pip ...":
         ferr.close()
         break
   else:
      ferr.close()
      print "<p> Run %s not yet completed. Please try again in a few seconds." % code
      DrawFormatResultsForm (code)
      return


   rigorOut = os.path.join ("/var/tmp/gaspsdb/rigorRuns/", code + ".out")
   fp = open (rigorOut)
   hits,errorMessages = polacco.Rigor.SimpleRigorTranslate (fp, silent=1)
   fp.close()

   lib = string.lower(code.split("_")[-1])
   pdb = code.split ("_")[0]
   if lib == "scop3":
      library = "SCOP superfamilies"
```

```
   elif lib == "scop4":
      library = "SCOP families"
   elif lib == "go":
      library = "GO annotations"
   elif lib == "goscop":
      library = "SCOP superfamilies subdivided by GO annotations"
   else:
      library = "selected motifs"
   print ("<h2> Matches to  %s among motifs from %s</h2>(code: %s)" % (pdb, library, code) )

   if errorMessages:
      print "<hr><p> <b> Warning:</b> The following error message(s) was encountered while completing the rigor run."
      for message in errorMessages:
         print "<p>", message
      print "<hr>"
   DrawRigorHitsTable (hits, lib, code, first, last)

def DrawDescribeForm ():
   DrawBrowseChoices(title=False)
   #print "<p>  l a z y ."

def GetCursorFromDatabase():
   import MySQLdb
   db = MySQLdb.connect (“””#### marked out for security ######”””)
   c = db.cursor()
   return c

def LibrarySort (toSort, index = None):
   if index != None:
      toSort = [ (string.split(l[index], "."), l) for l in toSort]
   else:
      toSort = [ (string.split(l, "."),l) for l in toSort]

   for item in toSort:
      for i in  range (len (item[0]) ):
         try:
            item[0][i] = int(item[0][i])
         except ValueError:
            pass
   toSort.sort()
   return [item[1] for item in toSort]


def DrawBrowseChoices (title=True):
   if title:
      print"""
      <div align="center" id="titleDiv">
      <h1> Browse the Motifs </h1>
      </div>
      """
   print"""
      <p>
      You can browse the motifs generated on
      <ul>
```

```
         <li><a href="/cgi-bin/GASPSdb?do=browse&amp;class=scop&amp;depth=4">SCOP families</a></li>
         <li><a href="/cgi-bin/GASPSdb?do=browse&amp;class=scop&amp;depth=3">SCOP superfamilies</a></li>
         <li><a href="/cgi-bin/GASPSdb?do=browse&amp;class=go">GO annotations</a></li>
         <li><a href="/cgi-bin/GASPSdb?do=browse&amp;class=goscop">SCOP superfamilies subdivided by GO annotations</a></li>
      </ul>
      """
   DrawKeyWordSearchForm()

def KeySearch (form):
   c = GetCursorFromDatabase()
   keyWords = form.getfirst ("keywords")
   if not keyWords:
      DrawKeyWordSearchForm()
      return
   words = string.split (keyWords)
   whereList = ["description like '%%%s%%'"%word for word in words]
   where = string.join (whereList ," AND ")

   query = 'select g.name, g.description from groups g where %s'%where
   c.execute (query)
   groups = c.fetchall()


   print "<h2>Groups matching %s</h2>" % keyWords
   if not groups:
      print "No Groups Found!"
   else:
      print "Motif details can be viewed by clicking a group name."

      print "<table>"
      print TableHeader ("Group", "# Motifs", "Top G-Score", "Description")
      for (groupName, groupDesc) in groups:
         c.execute ('select struct, gScore from motifs where groupName = "%s" order by gScore' % groupName)
         motifs = c.fetchall()
         motCount = len (motifs)
         if motifs:
            topScore = motifs[-1][1]
         else:
            topScore = "NA"
         groupName = '<a href ="./GASPSdb?do=describeGroup&group=%s" >%s</a>' % (groupName, groupName)
         print TableRow ( TableData (groupName, motCount, topScore, groupDesc) )
      print "</table>"
   DrawKeyWordSearchForm()

def StructSearchGroups (form):
   imagePath = "/images/"
```

```
   c = GetCursorFromDatabase()
   structs = form.getfirst ("structs")
   if not structs:
      DrawStructSearchForm()
      return
   hideMotifs = form.getfirst ("hideMotifs")
   if hideMotifs and string.upper (hideMotifs) == "TRUE":
      hideMotifs = True
   else:
      hideMotifs = False

   if hideMotifs:
      toggle = "FALSE"
   else:
      toggle = "TRUE"
   toggleMotifsURL = "./GASPSdb?" + urllib.urlencode
({"do":"structSearch", "structs":structs, "hideMotifs":toggle})
   structs = string.split (structs)
   ignore = []
   for struct in structs:
      if len (struct) != 4:
         print "Structure %s excluded from search, pdb identifiers
should be at least four characters long."
         ignore.append (struct)
   for struct in ignore:
      structs.remove (struct)

   whereList = ["s.name like '%%%s%%'"%struct for struct in structs]
   where = string.join (whereList, " OR ")
   query = 'select g.name, g.description, s.name from groups g inner
join group_struct gs  on g.name = gs.groupName inner join structs s
on gs.structName = s.name where %s' %where
   #print query
   c.execute (query)
   groups = c.fetchall()

   print "<h2>Groups containing motifs from pdbs: %s</h2>" %
string.join (structs, " OR ")
   if not groups:
      print "No Groups Found!"
   else:
      print "Motif details can be viewed by clicking a group name."

      print '<table class="smallTable">'
      if hideMotifs:
         print TableHeader ("Group", "Description", "Structure", '<a
href = "%s">Show Motifs</a>'%toggleMotifsURL)
      else:
         print TableHeader ("Group", "Description", "Structure",
"G-Score", "Motif", "Image")
      tableRows=[]
      rowSwitcher = TableRowGenerator()
      for (groupName, groupDesc, matchStruct) in groups:
         #get the motifs for each structure
         c.execute ('select id, gScore from motifs where groupName =
"%s" and struct="%s"' % (groupName, matchStruct) )
         motifs = c.fetchall()
```

```
         groupNameLinked = '<a href ="./GASPSdb?do=describeGroup&group=%s" >%s</a>' % (groupName, groupName)
         if motifs:
            for motRow in motifs:
#
#              #get the residues for each motif
#              c.execute ('select pdb, chain, name, resType from motifs m inner join motif_residue mr on m.id = mr.motifID inner join residues r on mr.resID = r.id where m.id = %d order by name' % motRow[0] )
#              residues = c.fetchall()
#              try:
#                 resToSort = [ (int(resRow[2]), resRow) for resRow in residues]
#                 resToSort.sort()
#                 residues = [row[1] for row in resToSort]
#              except ValueError:
#                 pass
#              resTable = '\n<table class="residue">' + string.join ([TableRow(TableData (resRow[3], resRow[2], resRow[1]), resRow[3]) for resRow in residues], "\n")    +" </table>"
               imageName =  imagePath + groupName+"_"+matchStruct+".r3d.png" #group_struct.r3d.png
               imageTag = '<img src="'+imageName+'" alt="Motif Image" width="240" height="192">'
               try:
                  gScore =  float (motRow[1]) #gScore
               except TypeError:
                  gScore = 0.0  #Acdtually these should simply be skipped
                             # but leave them in so I have a chance of finding them!
                             #check http://babbittlab.ucsf.edu/cgi-bin/GASPSdb?do=describeGroup&group=00008236
               #first item in each item enables sorting on that item (gScore)
               if hideMotifs:
                  tableRows.append ( (gScore, TableData (groupNameLinked, groupDesc, matchStruct, "")))
               else:
                  resTable = GetResTableByMotifID (c, motRow[0])
                  tableRows.append ( (gScore, TableData (groupNameLinked, groupDesc, matchStruct, "%5.3f"%gScore, resTable, imageTag)))
         else:
            if hideMotifs:
               tableRows.append ( (0.0, TableData (groupNameLinked, groupDesc, matchStruct, "")))
            else:
               tableRows.append ( (0.0, TableData (groupNameLinked, groupDesc, matchStruct, "", "", "No Motif Generated")))
            #if not motifs are found, still report a matching group....
      for row in tableRows:
         print TableRow (row[1], rowSwitcher.next())
      print "</table>"

```

```python
def Browse (form):
   c = GetCursorFromDatabase()

   minGScore = float (form.getfirst ("minGScore", "0.0"))

   classification = form.getfirst ("class")

   if not classification:
      DrawBrowseChoices()
      return

   where = 'where r.classification = "%s"' % classification
   depth = form.getfirst ("depth")

   if depth:
      try:
         depth = int(depth)
      except ValueError:
         depth = 3
      where = where + " and r.depth = %d" %depth

   orderBy = ""
   orderBy = form.getfirst ("orderBy", "")
   if orderBy:
      if orderBy == 'gScore':
         orderBy = 'm.gScore'
      else:
         orderBy = ''
   if orderBy:
      orderBy = "order by " + orderBy

   showMotifs= form.getfirst ("showMotifs", '')
   c.execute ('select g.name, g.description, m.id, m.struct, m.gScore, m.numMotifs from groups g inner join runs r on g.runID = r.id inner join topMotifs m on g.name = m.groupName %s %s' % (where, orderBy) )
   groups = c.fetchall()

   if not groups:
      print "<p>Invalid classification</p><hr>"
      DrawBrowseChoices()
      return


   #sort groups in an intuitive manner:
   if not orderBy:
      groups = LibrarySort (groups, 0)

   classification = string.lower (classification)
   if classification == "scop":
      if depth == 3:
         description = "SCOP Superfamilies"
```

```
      elif depth == 4:
         description = "SCOP Families"
      else:
         description = "SCOP groups"
   elif classification == "go":
      description = "GO Annotations"
   elif classification == "goscop":
      description = "SCOP Superfamilies subdivided by GO annotations"
   else:
      description = "Selected Groups"

   ToolTipScripts()
   ToolTipDiv()
   print "<h2>Browsing %s</h2>" % description
   print "<strong>Hover</strong> over a group name to view a sample motif."
   print "<br><strong>Click</strong> a group name to view all motifs for a group."

   print "<table>"
   print TableHeader ("Group", "# Motifs", "Top G-Score", "Description")
   for (groupName, groupDesc, motifID, struct, topScore, motCount) in groups:
#     c.execute ('select struct, gScore, id from motifs where groupName = "%s" order by gScore' % groupName)
#     motifs = c.fetchall()
#     motCount = len (motifs)
#     if motifs:
#        topScore = motifs[-1][1]
#        motifID = motifs[-1][2]
#     else:
#        topScore = "NA"
#        motifID = ""
#     if topScore == "NA" or topScore < minGScore:
#        continue
      groupNameHTML = '<a href ="./GASPSdb?do=describeGroup&group=%s" >%s</a>' % (groupName, groupName)
      if showMotifs:
         groupDesc = '<table><th colspan=2>'+groupDesc+"</th>" + TableRow(TableData(GetResTableByMotifID (c, motifID), GetImageTag (groupName, struct))) + "</table>"
      print TableRow ( ToolTipGroupName_td (groupNameHTML, groupName, motifID) + TableData (motCount, topScore, groupDesc) )
   print "</table>"


def GetImageTag (groupName, struct):
   imagePath = "/images/"
   imageName =  imagePath + groupName+"_"+struct+".r3d.png" #group_struct.r3d.png
   imageTag = '<img src="'+imageName+'" alt="Motif Image" width="240" height="192">'
   return imageTag


```

```
def ToolTipGroupName_td (groupNameHTML, groupName, motifID=None, structName=None):
   if not motifID:
      motifID=""
   if not structName:
      structName=""
   td = """
   <td id = "%s" onmouseover="getGroupDataWithTO(this, '%s', '%s');" onmouseout="cancelGroupData();">
   %s
   </td>
   """ % (groupName, motifID, structName, groupNameHTML)
   return td

def ToolTipDiv ():
   print """
   <div style="position:absolute;" id="popup" bgcolor="CFCFCF">
      <table id="popupTable"  bgcolor="CFCFCF" border="0" cellspacing="0" cellpadding="0">
      <thead>
      <tr><th id ="popupStatus" bgcolor = "CFCFCF"></th>
           <th id = "popupHeader" bgcolor = "CFCFCF" colspan="2" align="left"></th></tr>
      </thead>
      <tbody id = "popupTableBody"></tbody>
      </table>
   </div>
   """

def ToolTipScripts():
   print """
   <script SRC="/js/MochiKit.js" TYPE="text/javascript"></script>
   <script type = "text/javascript">
   var lastDeferred;
   var lastTO;

   function getGroupDataWithTO (element, motifID, structName){
      _doit = function(){
         getGroupData (element, motifID, structName);
         }
      lastTO = setTimeout ( "_doit();", 500);
      }

   function getGroupData(element, motifID, structName){
      var url = "/cgi-bin/jsonMotif?group=" + escape (element.id) + "&motifID=" + escape (motifID) + "&struct=" + (escape(structName));
      var d = loadJSONDoc (url);
      d.addCallback (partial (showMotifTable, element, motifID) );
      lastDeferred = d;
      showToolTip (element, status="loading...");
      }

   function cancelGroupData(){
      document.getElementById ("popupHeader").innerHTML = "";
      document.getElementById ("popupStatus").innerHTML = "";
      if (lastDeferred) {
         lastDeferred.cancel();
         lastDeferred = null;
```

```
         }
      if (lastTO){
         clearTimeout(lastTO);
         }
      clearData();
      }

   function showToolTip (element, status){
      clearData();
      setOffsets(element);
      document.getElementById ("popupStatus").innerHTML = status;
      document.getElementById ("popupTableBody").innerHTML= '<TR><td width= "400" height="200" align="center"></td></TR>'
      }


   function showMotifTable (element, motifID, result){
      document.getElementById ("popupStatus").innerHTML = "";
      document.getElementById ("popupTableBody").innerHTML = result["html"];
      document.getElementById ("popupHeader").innerHTML = result["group"]
   }

   function clearData(){
      document.getElementById ("popupTableBody").innerHTML = "";
      document.getElementById ("popup").style.border = "none";

      document.getElementById ("popupTable").setAttribute('cellPadding',0);
      document.getElementById ("popupTable").setAttribute('cellSpacing',0);
      }


   function calculateOffset (field, attr) {
      var offset = 0;
      while (field){
         offset += field[attr];
         field = field.offsetParent;
         }
      return offset;
      }

   function onTop (element) {
      var height = 250;
      var scrollTop = document.body.scrollTop;
      var top = element.offsetHeight + calculateOffset (element, "offsetTop");
      return  ( (top - height) > scrollTop);
      }


   function setOffsets(element){

      var end = element.offsetWidth + calculateOffset (element, "offsetLeft");
```

```
      var top = element.offsetHeight + calculateOffset (element, "offsetTop");
      dataDiv  = document.getElementById ("popup");
      dataDiv.style.bgcolor = "CFCFCF";
      dataDiv.style.border = "black 1px solid";
      dataDiv.style.left = end + 15 + "px";
      var ot = onTop(element)
      if (ot)
         {dataDiv.style.top = top - 250 + "px";}
      else
         {dataDiv.style.top = top + 5 + "px";}

      document.getElementById ("popupTable").setAttribute ('cellPadding', 2);
      document.getElementById ("popupTable").setAttribute ('cellSpacing', 2);

      }
   </script>
   """

def GetResTableByMotifID (c, motifID):
   #get the residues for each motif
#  c.execute ("""SELECT r.pdb, r.chain, r.name, r.resType, l.shortLigandName, b.type, o.name
#           FROM motifs m
#           INNER JOIN motif_residue mr on m.id = mr.motifID
#           INNER JOIN residues r on mr.resID = r.id
#           LEFT JOIN ligandInts l on r.id = l.motResId
#           LEFT JOIN (SELECT * FROM bridges WHERE type="disulfide") b on r.id = b.motResId
#           LEFT JOIN residues o on o.id = b.otherResId
#           WHERE m.id = %d ORDER BY r.name""" % motifID )
   c.execute ("""SELECT r.pdb, r.chain, r.name, r.resType, l.shortLigandName, b.type, o.name
            FROM motifs m
            INNER JOIN motif_residue mr on m.id = mr.motifID
            INNER JOIN residues r on mr.resID = r.id
            LEFT JOIN ligandInts l on r.id = l.motResId
            LEFT JOIN bridges b on r.id = b.motResId
            LEFT JOIN residues o on o.id = b.otherResId
            WHERE m.id = %d ORDER BY r.name""" % motifID )
   residues = c.fetchall()
   try:
      resToSort = [ (int(resRow[2]), resRow) for resRow in residues]
      resToSort.sort()
      residues = [row[1] for row in resToSort]
   except ValueError:
      pass

   resTableRows = []
   for resRow in residues:
      if resRow[4]:
         ligand = '~' + resRow[4]
      else:
         ligand = ''
      if resRow[5] == 'disulfide':
         bridge = "SS-"
```

```
         if resRow[6]:
            bridge = bridge + resRow[6]
      elif resRow[5] == 'salt':
         bridge = "+-"
         if resRow[6]:
            bridge = bridge + resRow[6]
      else:
         bridge = ''
      resTableRows.append ( TableRow(TableData (resRow[3], resRow[2], resRow[1], ligand, bridge), resRow[3]) )

   return '\n<table class="residue">' + string.join (resTableRows, "\n")    +" </table>"





def DescribeGroup (form):
   imagePath = "/images/"
   c = GetCursorFromDatabase()
#  import MySQLdb
#  c = db.cursor()

   # get group id
#  c.execute ('select id from groups where name = "%s";' % groupName)
#  groupIDs = c.fetchall()
#  if not groupIDs:
#     print" <p> Group name %s not recognized. Please enter a valid group name."
#     DrawDescribeFrom()
#     return
#  #ignore the possibility of two or more groups with identical names.
#  groupID = int(groupIDs[0][0])

   # don't be confused by the dummy for loop.  It's basically  a trick
   # to make easier coding.  Basically a failure at any if statements sends you to
   # the same else statement.
   for i in (1,):
      groupName = form.getfirst ("group")
      if groupName:
         c.execute ('select description from groups where name = "%s"' % groupName)
         groupDesc = c.fetchone()
         if groupDesc:
            break
   else:
      print "<p> Please enter a valid group name, or browse the groups below."
      DrawDescribeForm ()
      return

   #get all structures in a single group
```

```
   c.execute ('select s.name, s.pdb, s.chain, s.description, s.species, s.ec from structs s inner join group_struct gs on s.name = gs.structName where gs.groupName="%s"'% groupName)
   structures = c.fetchall()

   print '<h2> Group: %s;  %s </h2>' % (groupName, groupDesc[0])
   print '<table class="smallTable">'
   print TableHeader ("Structure", "G-score", "Motif", "Image")
   tableRows = []
   rowSwitcher = TableRowGenerator()
   for structRow in structures:
      # get the motifs if any for each structure
      c.execute ('select id, gScore from motifs where groupName = "%s" and struct="%s"' % (groupName, structRow[0]) )
      motifs = c.fetchall()
      for motRow in motifs:
         resTable = GetResTableByMotifID (c, motRow[0])
         imageName =  imagePath + groupName+"_"+structRow[0]+".r3d.png" #group_struct.r3d.png
         imageTag = '<img src="'+imageName+'" alt="Motif Image" width="240" height="192">'
         try:
            gScore =  float (motRow[1]) #gScore
         except TypeError:
            gScore = 0.0  #Acdtually these should simply be skipped
                       # but leave them in so I have a chance of finding them!
                       #check http://babbittlab.ucsf.edu/cgi-bin/GASPSdb?do=describeGroup&group=00008236

         tableRows.append ( (gScore, TableData ( DescribeStruct(structRow[0], structRow[3], structRow[4], structRow[5]), "%5.3f"%gScore, resTable, imageTag)))
   tableRows.sort()
   tableRows.reverse()

   for row in tableRows:
      print TableRow (row[1], rowSwitcher.next())

   print "</table>"

def HotLinkStructName (structName):

   #two choices  1a4ma and d1a4ma1
   if len (structName) == 7:
      return '<a href="http://scop.berkeley.edu/search.cgi?ver=1.65&amp;key=%s" target="_blank">%s</a> <a href = "http://www.ebi.ac.uk/thornton-srv/databases/cgi-bin/pdbsum/GetPage.pl?pdbcode=%s&template=protein.html&l=1" target="pdbSum">pdbSum</a>' % (structName, structName, structName[1:5])
   elif len (structName) == 5:
      return '<a href="http://www.pdb.org/pdb/explore/explore.do?structureId=%s" target="_blank">%s</a>%s' % (structName[0:4], structName[0:4], structName[4])

```

```
def DescribeStruct (structName, description="", species="", ec=""):
   if len (structName) ==7:
      pdbID= structName[1:5]
      chainID = structName[5]
      astralName = structName

   elif len (structName) == 5:
      pdbID = structName[0:4]
      chainID = structName[4]
      astralName = None

   if description:
      descriptionLine = TableRow ("<td colspan=2>"+Bold(description)
+ "</td>")
   else:
      descriptionLine = ""
   if species:
      speciesLine = TableRow (TableData(Bold('Organism'), species))
   else:
      speciesLine=""

   if ec:
      ecLine = TableRow (TableData (Bold('EC'), ec))
   else:
      ecLine = ""

   pdbLine =  TableRow (TableData (Bold('pdb'), '<a
href="http://www.pdb.org/pdb/explore/explore.do?structureId=%s"
target="_blank">%s</a> chain %s' % (pdbID, pdbID, chainID)))
   otherLinks = TableRow (TableData (Bold('links'),  """<a href =
"http://www.ebi.ac.uk/thornton-srv/databases/cgi-
bin/pdbsum/GetPage.pl?pdbcode=%s&template=protein.html&l=1"
target="pdbSum">pdbSum</a>
                                                            <br> <a
href="http://scop.berkeley.edu/search.cgi?key=%s"
target="scop">SCOP</a> """ % (pdbID, pdbID)))
   if astralName:
      scopLine = TableRow (TableData (Bold('scop'), '<a
href="http://scop.berkeley.edu/search.cgi?ver=1.65&amp;key=%s"
target="_blank">%s</a>' % (astralName, astralName)))
   else:
      scopLine = ""

   return '<table class="smallTable">' + descriptionLine + ecLine +
speciesLine + pdbLine +  scopLine +  otherLinks + "</table>"


def TableRowGenerator():
   switch = False
   while True:
      switch = not switch
      if switch:
         yield ("row1")
      else:
         yield ("row2")


def TableRow (text, type = None):
```

```
   if type:
      first = "<tr class=\"%s\">" %type
   else:
      first = "<tr>"
   return first + text + "</tr>"


def TableData (*cells):
   cells = ["%s" % s for s in cells]
   return '<td>' + string.join (cells, '</td><td>')  + '</td>'
def TableHeader (*cells):
   return '<th>' + string.join (cells, '</th><th>') + '</th>'

def Bold (text):
   return '<b>' + text+"</b>"
def Green (text):
   return '<font color="green">'+text+"</font>"
def Yellow (text):
   return '<font color="orange">'+text+"</font>"
def Red (text):
   return '<font color="red">'+text+"</font>"



def HTMLHead():
   print """<html><HEAD>
   <link rel=STYLESHEET type="text/css" href="/style.css">
   <title>GASPS Motif Database</title>
   </HEAD>
   <body>
   """

def DrawNavBar (form):
   print"""
   <div id="header" class="header">
   <map name="ucsfnosearch">

      <area shape="rect" alt="UCSF home page" coords="38,3,84,27" href="http://www.ucsf.edu/">
      <area shape="rect" alt="UCSF home page" coords="93,11,288,19" href="http://www.ucsf.edu/">
      <area shape="rect" alt="About UCSF" coords="306,11,368,19" href="http://www.ucsf.edu/about_ucsf/">
      <area shape="rect" alt="UCSF Medical Center" coords="387,11,498,19" href="http://www.ucsfhealth.org/" >

   </map>

   <table border="0" width="100%" cellpadding="0" cellspacing="0" summary="table used for layout purposes only">
   <tr bgcolor="#666666">
     <td><img src="/graphics/ucsfgraynosearch.gif" alt="UCSF navigation bar" width="537" height="30" border="0" usemap="#ucsfnosearch" ></td>
     <td style="padding-top:5px; padding-right: 10px; vertical-align:middle; text-align:right;">
     <a href="/"><img src="/graphics/GASPSdb_small.jpg" alt="GASPSdb" border="0" style="vertical-align: center;"></a>
```

```
     &nbsp;&nbsp;<a href="http://www.rbvi.ucsf.edu"><img src="/graphics/RBVI_logo_small.png" alt="RBVI" border="0"></a>
     </td>
   </tr>
   </table>

   <!-- GASPSDB Nav Bar  -->
   <table border="0" width="100%" cellpadding="0" cellspacing="0" >
      <tr bgcolor="#661166" style="text-align:center; color: white; font-family: arial,sans-serif; font-weight: bold;" >
         <td><a href="/" class="bigbutton">Main</a></td>
         <td><a href="/cgi-bin/GASPSdb?do=drawForm" class="bigbutton">Search</a></td>
         <td><a href="/cgi-bin/GASPSdb?do=retrieveResults" class="bigbutton">Get Results</a></td>
         <td><a href="/cgi-bin/GASPSdb?do=browse" class="bigbutton">Browse Motifs</a></td>
         <td><a href="/downloads.html" class = "bigbutton">Downloads</a></td>
         <td align="right">
            <table cellpadding="5px" >
               <tr>
               <td><a href="/help.html" class="smallbutton">Help</a></td>
               <td><a href="/references.html" class="smallbutton">References</a></td>
               </tr>
            </table></td>
      </tr>
   </table>
   </div>

   <!-- Main body of page starts here  -->

   <div class = "body">
   """

def HTMLTail():
   print "</div></body></html>"
   sys.exit(0)

def main():

   print "Content-Type: text/html"     # HTML is following
   print                               # blank line, end of headers

   form = cgi.FieldStorage()

   debug = form.getfirst ("debug", "")
   if debug == "mince":
      import cgitb; cgitb.enable()


   do = form.getfirst ("do", "")
   if not do:
      do = "drawForm"

#  if do=="rigor":
```

```
#     DoRigor(form)
#  else:
#     #main switch function:
   HTMLHead()
   DrawNavBar (form)
   if do == "drawForm" or do == "search":
      DrawSearchPage()
#     DrawMainForm()
#  elif do == "search":
   elif do == "rigor":
      DoRigor(form)
   elif do == "retrieveResults":
      RetrieveResults(form)
   elif do == "describeGroup":
      DescribeGroup (form)
   elif do == "describeMotif":
      DescribeMotif (form)
   elif do == "browse":
      Browse(form)
   elif do == "keySearch":
      KeySearch(form)
   elif do == "structSearch":
      StructSearchGroups (form)

   else:
      print "<p> Unrecognized do command", do, ". Try again."
      print "<hr>"
      DrawMainForm()
   HTMLTail()



if __name__ == "__main__":
   main()
```

jsonMotif

```
#!/usr/local/bin/python2.4

import cgi, string

imagePath = "/images/"

def GetCursorFromDatabase():
   import MySQLdb
   db = MySQLdb.connect ("""#### marked out for security #####""")
   c = db.cursor()
   return c

def TableRow (text, type = None):
   if type:
      first = "<tr class=\"%s\">" %type
   else:
      first = "<tr>"
   return first + text + "</tr>"

```

```

def TableData (*cells):
   cells = ["%s" % s for s in cells]
   return '<td>' + string.join (cells, '</td><td>')  + '</td>'

def GetResTableByMotifID (c, motifID):
   #get the residues for each motif
#  c.execute ("""SELECT r.pdb, r.chain, r.name, r.resType, l.shortLigandName, b.type, o.name
#           FROM motifs m
#           INNER JOIN motif_residue mr on m.id = mr.motifID
#           INNER JOIN residues r on mr.resID = r.id
#           LEFT JOIN ligandInts l on r.id = l.motResId
#           LEFT JOIN (SELECT * FROM bridges WHERE type="disulfide") b on r.id = b.motResId
#           LEFT JOIN residues o on o.id = b.otherResId
#           WHERE m.id = %d ORDER BY r.name""" % motifID )
   c.execute ("""SELECT r.pdb, r.chain, r.name, r.resType, l.shortLigandName, b.type, o.name
            FROM motifs m
            INNER JOIN motif_residue mr on m.id = mr.motifID
            INNER JOIN residues r on mr.resID = r.id
            LEFT JOIN ligandInts l on r.id = l.motResId
            LEFT JOIN bridges b on r.id = b.motResId
            LEFT JOIN residues o on o.id = b.otherResId
            WHERE m.id = %d ORDER BY r.name""" % motifID )
   residues = c.fetchall()
   try:
      resToSort = [ (int(resRow[2]), resRow) for resRow in residues]
      resToSort.sort()
      residues = [row[1] for row in resToSort]
   except ValueError:
      pass

   resTableRows = []
   for resRow in residues:
      if resRow[4]:
         ligand = '~' + resRow[4]
      else:
         ligand = ''
      if resRow[5] == 'disulfide':
         bridge = "SS-"
         if resRow[6]:
            bridge = bridge + resRow[6]
      elif resRow[5] == 'salt':
         bridge = "+-"
         if resRow[6]:
            bridge = bridge + resRow[6]
      else:
         bridge = ''
      resTableRows.append ( TableRow(TableData (resRow[3], resRow[2], resRow[1], ligand, bridge), resRow[3]) )

   return '\n<table class="residue">' + string.join (resTableRows, "\n")    +" </table>"



```

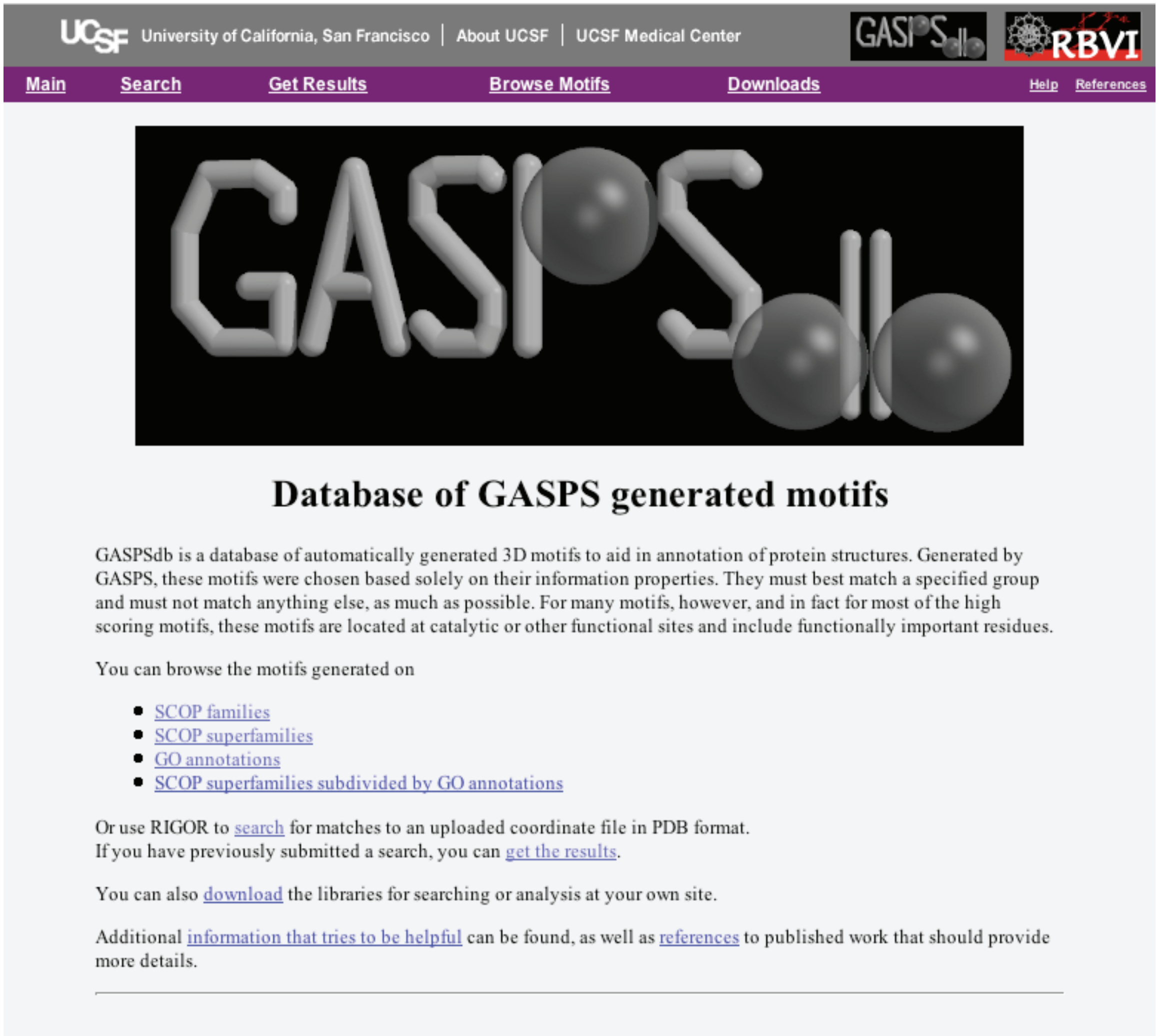
```
def DescribeMotif (groupName, motifID=None, struct=None, debug=False):
   c = GetCursorFromDatabase();
   if not motifID:
      c.execute ('select id from motifs where groupName = "%s" and struct = "%s"' % (groupName, struct))
      motifIDS = c.fetchall()
      assert len (motifIDS) == 1
      motifID = motifIDS[0][0]

   c.execute ("select groupName, struct, gScore from motifs where id = %s" % motifID)
   groupName, struct, gScore = c.fetchone()
   resTable = GetResTableByMotifID (c, int(motifID))
#  c.execute ("""select resType, name, chain from residues r
#              inner join motif_residue mr on mr.resID = r.id
#              where mr.motifID = %s""" % motifID)
#
#  residues = c.fetchall()
#
#  try:
#     resToSort = [ (int(resRow[1]), resRow) for resRow in residues]
#     resToSort.sort()
#     resPairs = [row[1] for row in resToSort]
#  except ValueError:
#     pass
#  resTable = '\n<table class="residue">' + string.join ([TableRow(TableData (resRow[0], resRow[1], resRow[2]), resRow[0]) for resRow in residues], "\n")    +" </table>"
   imageName =  imagePath + groupName+"_"+ struct +".r3d.png" #group_struct.r3d.png
   imageTag = '<img src="'+imageName+'" alt="Motif Image" width="240" height="192"/>'
   try:
      gScore =  float (gScore) #gScore
   except TypeError:
      gScore = 0.0  #Acdtually these should simply be skipped
                 # but leave them in so I have a chance of finding them!
                 #check http://babbittlab.ucsf.edu/cgi-bin/GASPSdb?do=describeGroup&group=00008236

   textTable = """<table>
               %s
               %s
               </table>
               """%( TableRow (TableData ("<strong>Structure:</strong>",  struct)),
                     TableRow (TableData ("<strong>G-Score:</strong>",  "%5.3f" % gScore)) )

   html =  TableRow (TableData (textTable, resTable, imageTag))

   c.execute ('select description from groups where name = "%s";' % (groupName))
   description = c.fetchall()
   if len (description) == 1:
      groupName = groupName + " : " + description[0][0]
```

```
   if debug:
      print "Content-Type: text/html"
      print
      print html
      return

   print "Content-Type: text/javascript"
   print
   print dict(html = html, group=groupName)

def main():
   form = cgi.FieldStorage()
   group = form.getfirst ("group", "")
   motifID = form.getfirst ("motifID", "")
   struct = form.getfirst ("struct", "")
   debug = form.getfirst ("debug", "")
   if debug:
      import cgitb; cgitb.enable()

   DescribeMotif (group, motifID, struct, debug)


if __name__ == "__main__":
   main()

"""
http://gaspsdb.rbvi.ucsf.edu/cgi-bin/jsonMotif?group=c.37.1&struct=d1in4a2&motifID=
http://gaspsdb.rbvi.ucsf.edu/cgi-bin/jsonMotif?group=&struct=d1efva1
"""
```

Appendix 3: GASPSdb Web Interface

Shown here are screen shots and text from http://gaspsdb.rbvi.ucsf.edu.

GASPSdb Home Page



Database of GASPS generated motifs

GASPSdb is a database of automatically generated 3D motifs to aid in annotation of protein structures. Generated by GASPS, these motifs were chosen based solely on their information properties. They must best match a specified group and must not match anything else, as much as possible. For many motifs, however, and in fact for most of the high scoring motifs, these motifs are located at catalytic or other functional sites and include functionally important residues.

You can browse the motifs generated on

- SCOP families
- SCOP superfamilies
- GO annotations
- SCOP superfamilies subdivided by GO annotations

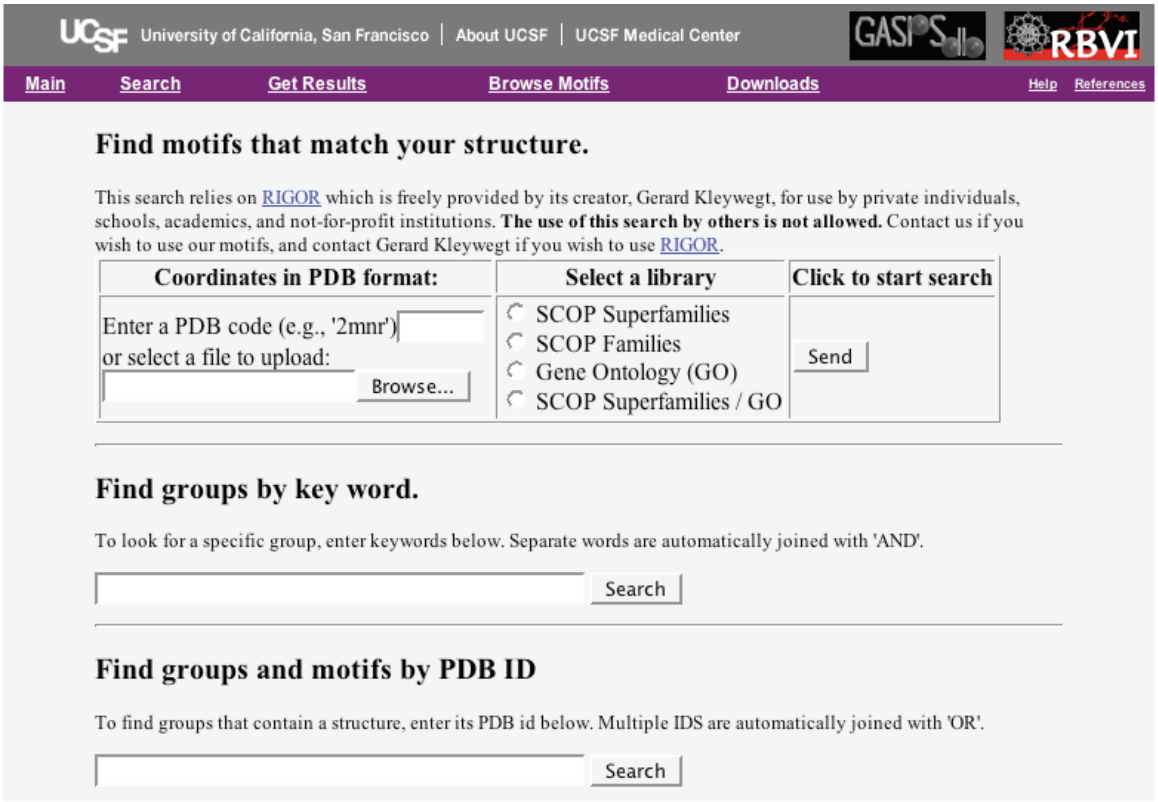
Or use RIGOR to search for matches to an uploaded coordinate file in PDB format.
If you have previously submitted a search, you can get the results.

You can also download the libraries for searching or analysis at your own site.

Additional information that tries to be helpful can be found, as well as references to published work that should provide more details.

Figure 1. Home page of GASPSdb.

GASPSdb Search Page



Find motifs that match your structure.

This search relies on RIGOR which is freely provided by its creator, Gerard Kleywegt, for use by private individuals, schools, academics, and not-for-profit institutions. **The use of this search by others is not allowed.** Contact us if you wish to use our motifs, and contact Gerard Kleywegt if you wish to use RIGOR.

Coordinates in PDB format:	Select a library	Click to start search
Enter a PDB code (e.g., '2mnr') or select a file to upload:	○ SCOP Superfamilies ○ SCOP Families ○ Gene Ontology (GO) ○ SCOP Superfamilies / GO	

Find groups by key word.

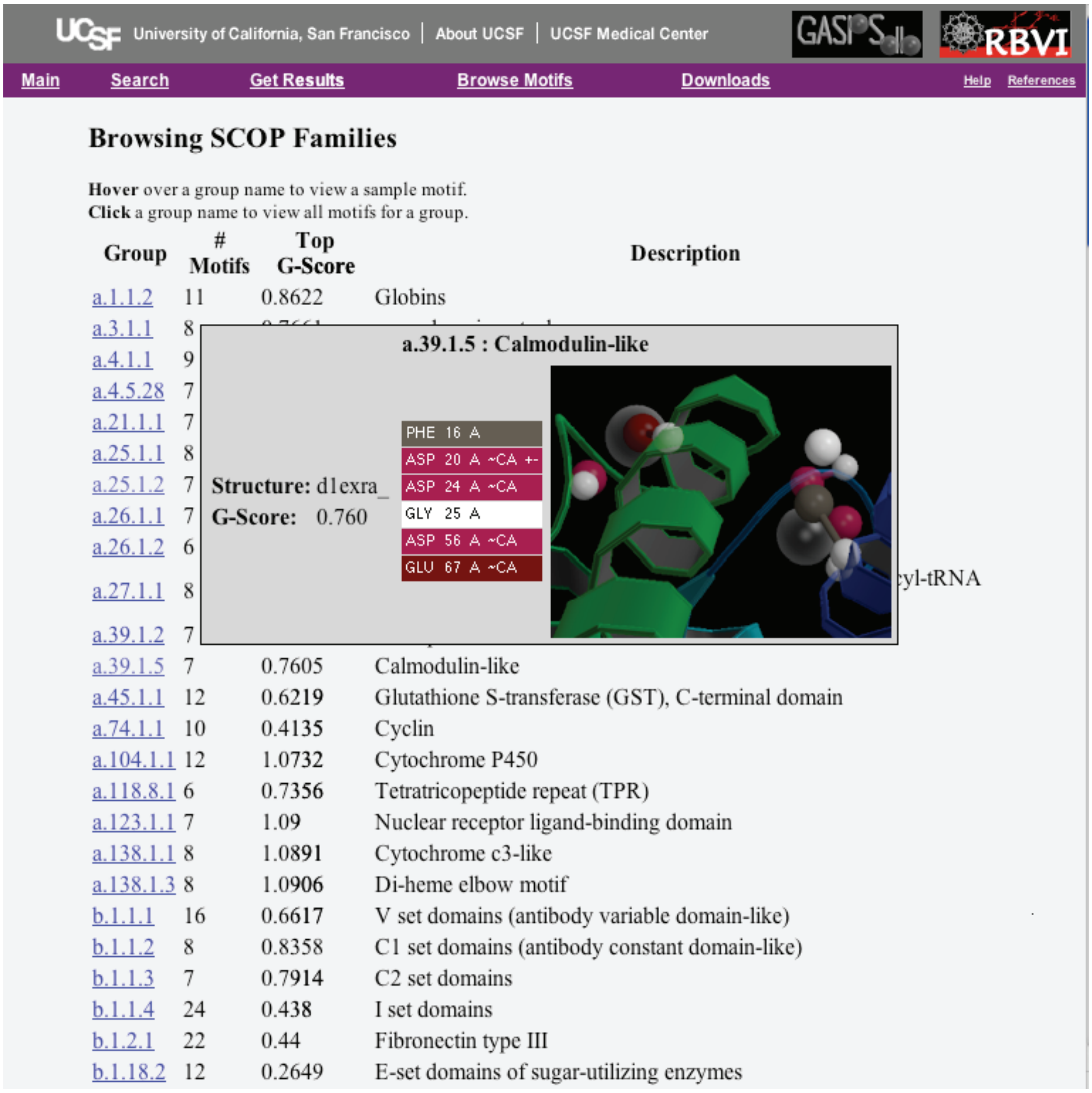
To look for a specific group, enter keywords below. Separate words are automatically joined with 'AND'.

Find groups and motifs by PDB ID

To find groups that contain a structure, enter its PDB id below. Multiple IDS are automatically joined with 'OR'.

Figure 2. GASPSdb Search Page.

GASPSdb Browsing Page



Browsing SCOP Families

Hover over a group name to view a sample motif.
Click a group name to view all motifs for a group.

Group	# Motifs	Top G-Score	Description
a.1.1.2	11	0.8622	Globins
a.3.1.1	8		
a.4.1.1	9		
a.4.5.28	7		
a.21.1.1	7		
a.25.1.1	8		
a.25.1.2	7		
a.26.1.1	7		
a.26.1.2	6		
a.27.1.1	8		yl-tRNA
a.39.1.2	7		
a.39.1.5	7	0.7605	Calmodulin-like
a.45.1.1	12	0.6219	Glutathione S-transferase (GST), C-terminal domain
a.74.1.1	10	0.4135	Cyclin
a.104.1.1	12	1.0732	Cytochrome P450
a.118.8.1	6	0.7356	Tetratricopeptide repeat (TPR)
a.123.1.1	7	1.09	Nuclear receptor ligand-binding domain
a.138.1.1	8	1.0891	Cytochrome c3-like
a.138.1.3	8	1.0906	Di-heme elbow motif
b.1.1.1	16	0.6617	V set domains (antibody variable domain-like)
b.1.1.2	8	0.8358	C1 set domains (antibody constant domain-like)
b.1.1.3	7	0.7914	C2 set domains
b.1.1.4	24	0.438	I set domains
b.1.2.1	22	0.44	Fibronectin type III
b.1.18.2	12	0.2649	E-set domains of sugar-utilizing enzymes

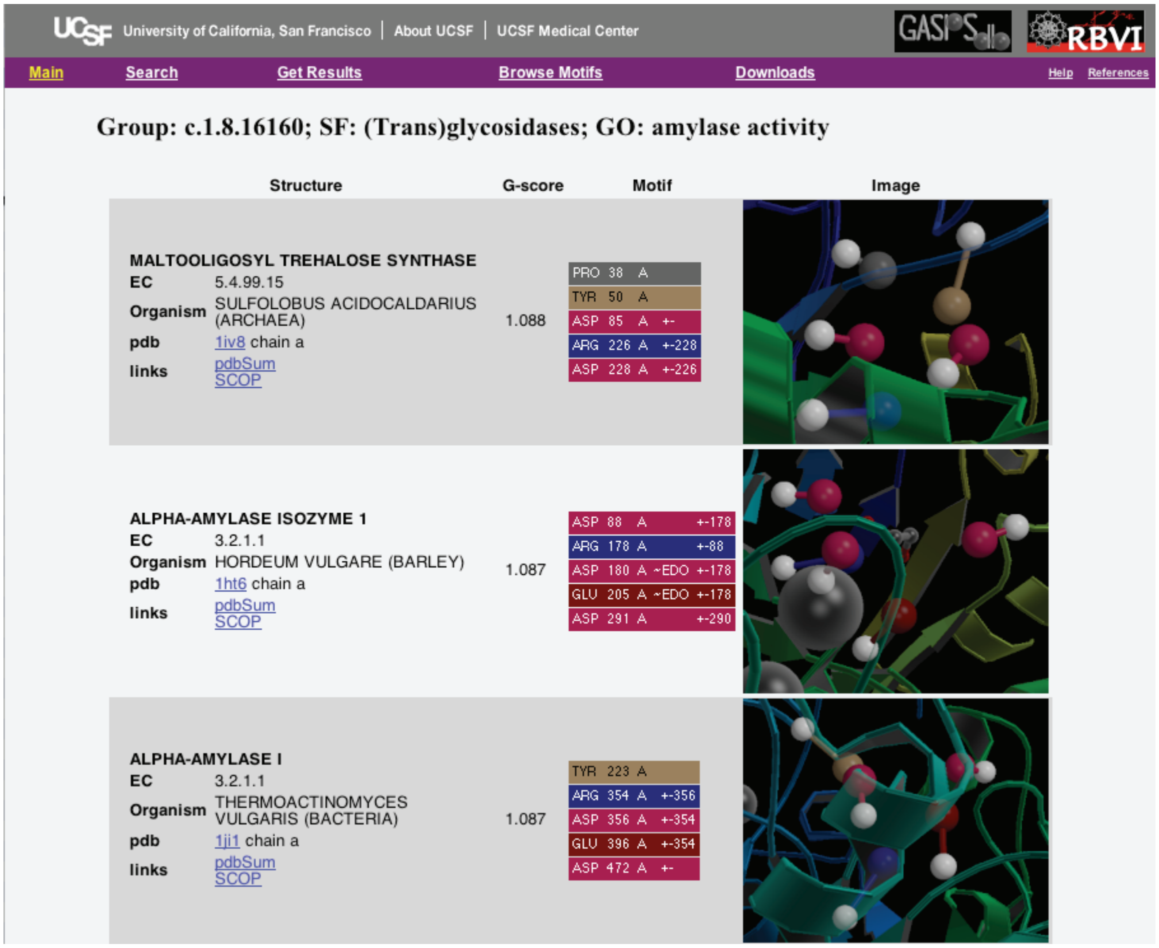
a.39.1.5 : Calmodulin-like

Structure: d1exra_
G-Score: 0.760

PHE 16 A
ASP 20 A ~CA +-
ASP 24 A ~CA
GLY 25 A
ASP 56 A ~CA
GLU 67 A ~CA

Figure 3. GASPSdb Browse Page.
Currently browsing scop superfamilies. The popup window is showing the top-scoring motif from the Calmodulin-like family in response to the pointer hovering over its group ID.

GASPSdb Group Description Page

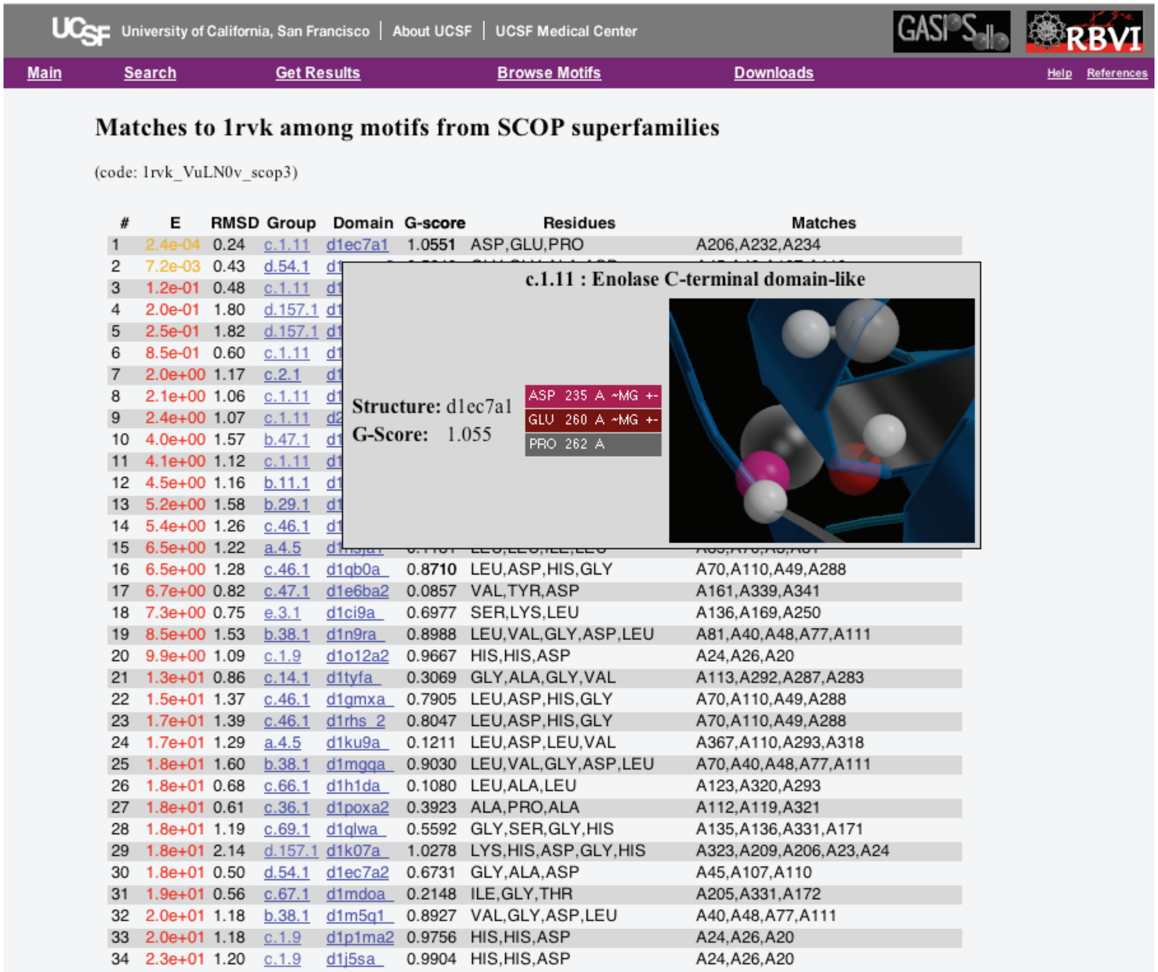


Group: c.1.8.16160; SF: (Trans)glycosidases; GO: amylase activity

Structure	G-score	Motif	Image
MALTOOLIGOSYL TREHALOSE SYNTHASE **EC** 5.4.99.15 **Organism** SULFOLOBUS ACIDOCALDARIUS (ARCHAEA) **pdb** 1iv8 chain a **links** pdbSum SCOP	1.088	PRO 38 A TYR 50 A ASP 85 A +- ARG 226 A +-228 ASP 228 A +-226	
ALPHA-AMYLASE ISOZYME 1 **EC** 3.2.1.1 **Organism** HORDEUM VULGARE (BARLEY) **pdb** 1ht6 chain a **links** pdbSum SCOP	1.087	ASP 88 A +-178 ARG 178 A +-88 ASP 180 A ~EDO +-178 GLU 205 A ~EDO +-178 ASP 291 A +-290	
ALPHA-AMYLASE I **EC** 3.2.1.1 **Organism** THERMOACTINOMYCES VULGARIS (BACTERIA) **pdb** 1ji1 chain a **links** pdbSum SCOP	1.087	TYR 223 A ARG 354 A +-356 ASP 356 A +-354 GLU 396 A +-354 ASP 472 A +-	

Figure 4. GASPSdb Group description page, partial.
Showing the first three motifs from a group in the GO and SCOP combination groupings. The full page shows additional entries for each remaining motif in the group.

GASPSdb Search Results Page



Matches to 1rvk among motifs from SCOP superfamilies

(code: 1rvk_VuLN0v_scop3)

#	E	RMSD	Group	Domain	G-score	Residues	Matches
1	2.4e-04	0.24	c.1.11	d1ec7a1	1.0551	ASP,GLU,PRO	A206,A232,A234
2	7.2e-03	0.43	d.54.1	d1			
3	1.2e-01	0.48	c.1.11	d1			
4	2.0e-01	1.80	d.157.1	d1			
5	2.5e-01	1.82	d.157.1	d1			
6	8.5e-01	0.60	c.1.11	d1			
7	2.0e+00	1.17	c.2.1	d1			
8	2.1e+00	1.06	c.1.11	d1			
9	2.4e+00	1.07	c.1.11	d2			
10	4.0e+00	1.57	b.47.1	d1			
11	4.1e+00	1.12	c.1.11	d1			
12	4.5e+00	1.16	b.11.1	d1			
13	5.2e+00	1.58	b.29.1	d1			
14	5.4e+00	1.26	c.46.1	d1			
15	6.5e+00	1.22	a.4.5				
16	6.5e+00	1.28	c.46.1	d1qb0a_	0.8710	LEU,ASP,HIS,GLY	A70,A110,A49,A288
17	6.7e+00	0.82	c.47.1	d1e6ba2	0.0857	VAL,TYR,ASP	A161,A339,A341
18	7.3e+00	0.75	e.3.1	d1ci9a_	0.6977	SER,LYS,LEU	A136,A169,A250
19	8.5e+00	1.53	b.38.1	d1n9ra_	0.8988	LEU,VAL,GLY,ASP,LEU	A81,A40,A48,A77,A111
20	9.9e+00	1.09	c.1.9	d1o12a2	0.9667	HIS,HIS,ASP	A24,A26,A20
21	1.3e+01	0.86	c.14.1	d1tyfa_	0.3069	GLY,ALA,GLY,VAL	A113,A292,A287,A283
22	1.5e+01	1.37	c.46.1	d1gmxa_	0.7905	LEU,ASP,HIS,GLY	A70,A110,A49,A288
23	1.7e+01	1.39	c.46.1	d1rhs_2	0.8047	LEU,ASP,HIS,GLY	A70,A110,A49,A288
24	1.7e+01	1.29	a.4.5	d1ku9a_	0.1211	LEU,ASP,LEU,VAL	A367,A110,A293,A318
25	1.8e+01	1.60	b.38.1	d1mgqa_	0.9030	LEU,VAL,GLY,ASP,LEU	A70,A40,A48,A77,A111
26	1.8e+01	0.68	c.66.1	d1h1da_	0.1080	LEU,ALA,LEU	A123,A320,A293
27	1.8e+01	0.61	c.36.1	d1poxa2	0.3923	ALA,PRO,ALA	A112,A119,A321
28	1.8e+01	1.19	c.69.1	d1qlwa_	0.5592	GLY,SER,GLY,HIS	A135,A136,A331,A171
29	1.8e+01	2.14	d.157.1	d1k07a_	1.0278	LYS,HIS,ASP,GLY,HIS	A323,A209,A206,A23,A24
30	1.8e+01	0.50	d.54.1	d1ec7a2	0.6731	GLY,ALA,ASP	A45,A107,A110
31	1.9e+01	0.56	c.67.1	d1mdoa_	0.2148	ILE,GLY,THR	A205,A331,A172
32	2.0e+01	1.18	b.38.1	d1m5q1_	0.8927	VAL,GLY,ASP,LEU	A40,A48,A77,A111
33	2.0e+01	1.18	c.1.9	d1p1ma2	0.9756	HIS,HIS,ASP	A24,A26,A20
34	2.3e+01	1.20	c.1.9	d1j5sa_	0.9904	HIS,HIS,ASP	A24,A26,A20

c.1.11 : Enolase C-terminal domain-like

Structure: d1ec7a1
G-Score: 1.055

ASP 235 A ~MG +-
GLU 260 A ~MG +-
PRO 262 A

Figure 5. Search results table for search of 1rvk against SCOP superfamily motifs. The popup window is displaying the first matched motif because the pointer is hovering over its group id.

GASPSdb Help Page

About GASPSdb

Where do these motifs come from?

These motifs were generated using an entirely automated method given the acronym GASPS. For specific details please refer to the GASPS reference on the References Page. In short, GASPS chooses a motif from a single structure that best separates related structures from all other structures. The GASPS score or G score measures the degree of this separation. For the motifs here, we used several different systems to define related structures. In all cases, the set of structures was reduced to exclude mutants as well as sequence-redundant structures at the level of 40% or 25% identity.

SCOP version 1.65

Members of the the same superfamily or family are grouped together.

Gene Ontology

Structures are grouped according to their molecular function terms, automatically assigning parent terms where appropriate so that groups can be defined at any level in the GO hierarchy. Terms that give redundant groups to terms lower in the hierarchy are ignored as well as groups that appear to be too general (those with more than 50 structures.)

SCOP superfamilies subdivided by GO annotations

In an attempt to get isofunctional, homologous groups, SCOP superfamilies were subdivided by all assigned and implied GO molecular function terms. Terms that give redundant groups were ignored.

GASPS uses SPASM (see References) to identify matches, so that each residue is modeled as two points in space: one representing the alpha carbon, and one representing the side chain centroid.

How does the search work?

We are indebted to others who have published and made available their motif searching tools for our use. Specifically Gerard Kleywegt and his RIGOR tool (see References). Our search receives your structure file in PDB format and finds all motifs in the specified library that have a close match in the PDB file.

How do I interpret the search results?

The search results on the search page are ranked according to an expectation value (E). The expectation value is computed according to the model generated by Stark et al. (see References), and is based on the RMSD as well as the type and number of residues in the motif. The G score may also help decide wether a match represents a significant similarity.

What is the G score?

Each motif is given a G score by GASPS. This is the score that GASPS tries to maximize as it constructs motifs. In short, a G score indicates how well conserved the motif is across the group, and how unique it is among unrelated proteins. This score has a theoretical range of 0-1.1, though any score near 1.0 is highly significant, and scores below about 0.4 are highly suspect. In cases with marginal E values, the G score may provide additional support.

In slightly more detail the G score is the sum of two components, the largest is the normalized area under an ROC style plot to a false positive rate of approximately 0.001, and the other component is the relative distance between true positive and false positive RMSD distributions. This latter component accounts for only 0 to 0.1 of the total G score, so that most G scores above 1.0 imply perfect separation in an ROC style plot (ROC area = 1.0).

What do the motif images show?

The motif images attempt to show the relative orientation of motifs and the local secondary structure and ligands in the protein. Residues in the motif are drawn as lopsided barbells. The smaller white sphere represents the alpha carbon and the larger colored sphere represents the side chain centroid. Side chains are colored according to residue type based on Bob Fletterick's 'shapely models' color scheme. Local secondary structure and ligands are drawn transparently to not mask the motifs.

These images are generated automatically with no effort to find a decent viewable orientation. With over 12,000 images, doing so by hand was not feasible.

GASPSdb References Page

References

Several tools provided by us and others have been instrumental in getting this resource online. Where possible, the tool names link to the relevant web sites.

GASPS

Genetic Algorithm Search for Patterns in Structures Responsible for generation of all motifs:

Polacco, Benjamin J. and Patricia C. Babbitt (2006) Automated discovery of 3D motifs for protein function annotation." Bioinformatics 22(6), 723-30.

SPASM

Provided by Gerard Kleywegt. This motif search tool was instrumental in calculating the scoring function for the above Genetic Algorithm.

Kleywegt, G. J. (1999). "Recognition of spatial motifs in protein structures." J Molecular Biology 285(4), 1887-97.

RIGOR

Sister program to SPASM. Simply does the reverse search of the above. Our search feature relies on RIGOR. Refer to citation above.

Raster3D

For final rendering of motif images (and GASPSdb logo!)

Merritt, Ethan A. and Bacon, David J. (1997). "Raster3D: Photorealistic Molecular Graphics" Methods in Enzymology 277, 505-524.

Molscript

Generated ribbon descriptions for rendering by Raster3D

Kraulis, Per J. (1991). "MOLSCRIPT: A Program to Produce Both Detailed and Schematic Plots of Protein Structures." Journal of Applied Crystallography 24, 946-950.

Statistical Model

The model described in the reference below was used to calculate the expectation values of matches to the motifs.

Stark, A., S. Sunyaev, et al. (2003). "A model for statistical significance of local similarities in structure." J Molecular Biology 326(5): 1307-16.

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.

Benjamin June 29, 2007

Author Signature Date