Homology Based Motif Generation

Cameron Baker
cdb1516@rit.edu

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Homology Based Motif Generation

Cameron Baker

Rochester Institute of Technology

Thesis Submitted in Partial Fulfillment of the Degree Master of Science in Bioinformatics

College of Science

Thomas H. Gosnell School of Life Sciences

Project Approved September 24, 2015
Committee Approval:

_____________________________________________________
Date
Paul Craig, Ph.D.
Professor of Biochemistry and Bioinformatics
Head of the School of Chemistry and Materials Science
Chief Advisor

_____________________________________________________
Date
Gary Skuse, Ph.D.
Professor of Biological Studies
Committee Member

_____________________________________________________
Date
Feng Cui, Ph.D.
Assistant Professor
Committee Member
Abstract

Generating motifs from known active sites and matching those motifs to an uncharacterized protein is a classic way of determining protein function. Until now, the generation of motifs has been based purely on enzymatic function. This approach does not account for situations where highly different active sites can arrive at the same function by processes like convergent evolution. As such, a secondary metric on which to base the generation of motifs is necessary. This metric exists in the form of UniProt designation for homologous proteins on a global scale or PFam for designation of homologous proteins at the active site level.

Here, we describe a tool to generate highly selective motifs using the aforementioned metrics. We were able to collapse a large number of proteins into their representative motifs with little loss in sensitivity, creating an “average” representation of each motif. These motifs will aid the characterizing proteins of known structure but unknown function.
# Table of Contents

1. Introduction and Background
   i. The Issue 1
   ii. What is a Motif? 1
   iii. What are Active Sites? 2
   iv. Measuring Protein Function 3
   v. Differences in Evolutionary History with Respect to Catalytic Function 4
   vi. The UniProt Database 6
   vii. The PFam Database 7
   viii. Previous Efforts 7
   ix. Anchor Protein 7

2. Materials and Methods
   i. Generation of Motifs 8
   ii. Querying of the UniProt or PFam Databases 9
   iii. Protein Clustering 10
   iv. Profile Generation and Averaging of Distances and Angles 11
   v. Transformation of the Template 12
   vi. Evaluation of Motifs 15
   vii. Significant Features of the Grading Plots 17
   viii. Relationship Between Grading and Selectivity 18

3. Results 20

4. Discussion 23
   i. The Issue of Variance 26
   ii. Stringency 28

5. Conclusion and Future Plans 29

6. Running the Software 30

7. Acknowledgements 32

8. References 33

9. Appendices
   i. Motifs Generated and Respective Plots 35
   ii. Code 54
Introduction and Background

The Issue

The group to most recently survey the Protein Data Bank (PDB), found that the PDB contains over 80,000 entries for macromolecules consisting of proteins, nucleic acids, and other complex structures (Nadzirin, 2012). Of the 80,000 structures, over 2500 of them are designated as proteins of unknown function (filtering out proteins of known function and proteins with no function). The 2500 consists of truly unique protein structures, as many entries within the Protein Data Bank are known duplicates (Nadzirin, 2012). Of those 2500 unique protein structures, 42.5% of proteins of unknown function have a structure that is truly unknown, compared to a structure of known function in need of proper annotation. The goal of many protein characterization efforts is to find the function of the proteins in this 42.5%. The goal of this project is to modify our approach to creating motif templates and to test this new approach so that we can improve our existing alignment methods.

What is a Motif?

A motif is a secondary super-structure that arises in proteins. Interactions between secondary-structures within proteins, along with associated residues, can give rise to catalytic activity. Motifs relating to structure and motifs relating to function can be used to classify proteins. It is possible for researchers to define their own motifs based on known structures within proteins. Common approaches to motif generation are founded in retrieving the active site of a protein with a given catalytic function and matching that against other proteins to help determine the function of the unknown protein through template-based structural alignment (Hanson, 2014). There are two problems with this approach: a single protein structure does not account for slight variations with respect to distance and orientation of the active site residues
and it does not account for differing structures stemming from convergent evolution between two proteins of the same function. Classification and determination of function through motif matching is one of the goals of a major project at RIT (Hanson, 2014).

What are Active Sites?

Active sites are the golden goose when it comes to determining the function of proteins. They are areas of catalytic activity and the binding place of inhibitors, ligands, substrates, activators, and other cofactors. Residues located within proteins can sometimes be substituted intentionally to better crystallize a protein to resolve its 3D structure. The function or native structure of a protein may prevent it from easily crystallizing, so a troublesome amino acid may be replaced in order to facilitate crystallization of the structure. If we defined a motif without taking these substitutions into account, these motifs will not be biologically accurate. Intentional mutations within active sites are common due to the structural similarity between some amino acids. The goal of such mutations is to replace the amino acid with another that is structurally similar to maintain active site stability but catalytically neutral to disrupt function. These changes are often not noted within PDB files or the description of active sites. PDB files are files that contain the location of all of the atoms within a protein structure as well as metadata related to the experiment. PDB files are the standard output format for the Protein Data Bank (Berman, 2000) as well as the standard within protein modelling software.

An example of this is the substitution of active sites within serine proteases (Gayathri, 2006). This process is the only way to proceed with these experiments, as, without the experimental substitution of particular residues within these active sites, the proteins will degrade before crystallization. Necessarily, this must be accounted for when generating motifs for a given protein. Within the analysis, a motif that contains a residue change based on conservative mutations will be accounted for. A table of mutations commonly induced for ease
of crystallization can be found in Table 1 (Hatzfeld, 2010). Once we determine where these active sites are, we need a way to classify them based on their function.

<table>
<thead>
<tr>
<th>Residue</th>
<th>Conserved Substitutions</th>
<th>Residue</th>
<th>Conserved Substitutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ala</td>
<td>Ser</td>
<td>Leu</td>
<td>Ile; Val</td>
</tr>
<tr>
<td>Arg</td>
<td>Lys</td>
<td>Lys</td>
<td>Arg; Gln</td>
</tr>
<tr>
<td>Asn</td>
<td>Gln; His</td>
<td>Met</td>
<td>Leu; Ile</td>
</tr>
<tr>
<td>Asp</td>
<td>Glu</td>
<td>Phe</td>
<td>Met; Leu; Tyr</td>
</tr>
<tr>
<td>Gly</td>
<td>Pro</td>
<td>Ser</td>
<td>Thr; Gly</td>
</tr>
<tr>
<td>Gln</td>
<td>Asn</td>
<td>Thr</td>
<td>Ser; Val</td>
</tr>
<tr>
<td>Glu</td>
<td>Asp</td>
<td>Tyr</td>
<td>Trp; Phe</td>
</tr>
<tr>
<td>Cys</td>
<td>Ser</td>
<td>Trp</td>
<td>Tyr</td>
</tr>
<tr>
<td>His</td>
<td>Asn; Gln</td>
<td>Val</td>
<td>Ile; Leu</td>
</tr>
<tr>
<td>Ile</td>
<td>Leu; Val</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Table of common amino acid substitutions used in site-directed mutagenesis. These substitutions inactive a protein so that it can more readily be crystallized.

**Measuring Protein Function**

A protein with a known and verified catalytic function is given an Enzyme Commission number (EC#). The four numbers in the given code precisely describe the protein's function (Webb, 1992). For example a protein with an EC of 1.1.1.22 has the distinctions shown in Figure 1 (Druzhinina, 1975). The Catalytic Site Atlas is the main repository for this information and provides searching for enzymatic function by PDB ID, UniProt ID, or EC number (Furnham, 2014). As a result, it is easy to retrieve information on active sites positioning within the query as well as relevant meta information. It is hosted through the European Bioinformatics Institute (or EBI for short). A general example of an EC# can be seen in Figure 1. While it is possible to describe all functional proteins with an EC#, it is possible still that differences in active sites give rise to identical functions.
Differences in Evolutionary History with Respect to Catalytic Function

While EC numbers can adequately describe the function of a protein, it only accounts for the function of the protein, not the residues that determine that function. Active site residue differences between members of the same EC class can result from convergent evolution of catalytic function. Different enzymes may have distinct evolutionary histories but still serve the same function in their respective organisms. Even though these active sites may have an identical function, the evolutionary path that resulted in this function may have resulted in very different structures. Due to this complication, it is not prudent or valid to create a generalized motif from a single EC class (Osipovitch, 2015). An example of 2 proteins with identical function but separate evolutionary background is shown in Figure 2. This is a major issue that this project seeks to remedy.

Figure 2 shows an example of proteins that are share low level of structural similarity but share the same function, hence the same EC# - *.*.*.*. The function, alcohol dehydrogenase, is conserved in both structures even though those structures show very little active site similarity. In Figure 3, it can be seen how this difference in structure is observed in the area about the
active sites. The structural alignment below displays how, if one was to query the function of one protein by comparing its active site with the active site of the other, it would not result in a positive match, despite the fact they do share a function. In order to provide stringency to the classes to account for differences in evolutionary backgrounds, we must apply evolutionary groupings across a set of proteins for a given EC, as denoted by relationships within the UniProt or PFam databases.

Figure 2: Proteins 1A4U (top left) and 1DEH (top right) both have the same function. The middle image is a superposition of 1A4U and 1DEH) created using PyMOL (PyMOL, 2015). The active site alignment between them (bottom row) shows that they are highly different.
Homology, in the context of biology, denotes a relationship of similarity between two objects that arises through evolutionary history. Our forearm bones are homologous with the forearm bones of chimpanzees. Genes from humans that encode the same gene product as a related gene in *E. coli* can be homologous. Proteins that are nearly identical can be homologous depending on the database definition of homology. While two proteins are either homologous or not, there is often be a measure of dissimilarity between homologous proteins. In reference to this project, the databases referenced contain designations for proteins based on 3D homology rather than sequence homology. When looking at the amino acid sequence of a protein, it is tough to determine true relationships until you get to a 3D view. As such, this project is measuring variations within known homologs. An example of these variations among homologs could be the differing distances between the residues of an active site.
The UniProt Database

UniProt, standing for Universal Protein Resource, is a database for protein sequence, classification, and functional information also hosted through the European Bioinformatics Institute (EBI) (UniProt Consortium, 2014). It forms the first discriminatory database that will be cross mapped to a given EC# to better classify protein motifs. The main reason UniProt will be used as a cross reference is due to the fact that any given protein may be crystallized multiple times and stored under different protein names within the PDB. UniProt contains information on all of these homologs by encompassing all of them under a single UniProt ID (UniProt Consortium, 2014). This ID is also pointed at directly from the Catalytic Site Atlas, making it the pivotal resource. Due to slight differences in homologs, it will be important to take all homologous proteins into account when measuring the variance within active sites to generate the most accurate representation of the active site for a group of proteins. This homology is measured at a global level in which the structure for the whole protein is taken into account. While the proteins shown in Figures 2 and 3 are members of the same EC class, they belong to separate UniProt families. Motifs generated from this distinction successfully match their parent family (see Results).

The PFam Database

PFam, short for Protein family, is a third database hosted by the EBI (Finn, 2016). Its function is largely curatorial, just like UniProt. This important difference between PFam and UniProt is the criteria on which proteins are grouped. Rather than grouping by homology at the global level, PFam groups are grouped by evolutionarily conserved domains at the residue level (Finn, 2016). This term has multiple implications and raises a number of interesting questions. Are these conserved domains related to active sites or are they just structural? Are the homologies described within the PFam database as structurally rigorous as the UniProt
database? The purpose of including Pfam as a reference base relies on the logging of evolutionarily conserved active sites at the residue level. These active sites over time may have degraded to the point where they are no longer functional but still belong within the same Pfam family. This allows us to generate motifs with the goal of studying the evolutionary changes that may have caused an active site to lose function. This will form a distinct dataset in contrast to the proteins generated by using UniProt as a reference which may lead to improved accuracy for active site alignments.

**Previous Efforts**

The general process of motif generation by active sites was described in work done by Osipovitch et al. and was found not to be stringent enough for complete motif generation (Osipovitch, 2015). Motifs generated in this fashion were found to be too narrowly defined as they were generated from a single protein. In some cases, motifs were unable to match to proteins from which they were generated.

**Anchor Protein**

Before going into the methods, it is important to describe some of the substantial features that will be commonly referred to later. An anchor residue is the centermost residue in an active site, as determined by distances to all other residues. The anchor residue will have the smallest average distance to all other residues, indicating that it is central to all of them. The anchor residue performs the basis of much of the analysis as a point of reference within a structure. All changes and measurements in the structure are done in relation to the anchor residue. Measurements to determine the distance between the anchor residue and others rely on built in PyMOL distance measurement function. The alpha carbons between two proteins serve as the point of reference for the measurement. Figure 4 shows the active site and
distances between the alpha carbon of the anchor residue and all other carbon alphas within the active site. Along with the anchor residue, the residue it will be compared against in context will be referred to as the other residue.

Figure 4: Distances between anchor residue and other residues within the active sites of protein 1GZN. The anchor residue is glutamate 279; the other residues are threonine 313, lysine 277, aspartate 275, and asparagine 280.

### Materials and Methods

#### Generation of Motifs

Development of the project was done primarily in Python, using the PyMOL API to exploit built in functions related to protein modelling and to trivialize the process. The multi-step process is summarized in Figure 5 and expanded upon in further sections.
Querying of the UniProt or PFam Databases

The process begins with the querying of either the UniProt or PFam databases for all of the IDs relating to a given EC class. This is done by retrieving all of the proteins for the EC class and matching their UniProt IDs to those hosted on the Catalytic Site Atlas. If the PFam database is being used, an additional step is included to retrieve the PFam IDs from the UniProt website. It should be noted that not all protein records are complete within the Catalytic Site Atlas. As of yet, approximately 20% of proteins tested have no active sites yet associated with them despite being logged as such within the Protein Data Bank. These proteins are filtered out at this step. For the sake of simplicity, EC/ID will refer to the protein set created from the overlap between the EC class and the PFam or UniProt designations.
Protein Clustering

This step focuses on grouping the proteins into their respective groups (related to the ID derived above). This step is highly related to the step above and done at the same time. As the UniProt or PFam IDs are being collected, the proteins are allocated to those IDs. Sample groupings are depicted in Table 2.

<table>
<thead>
<tr>
<th>UniProt classifier</th>
<th>Proteins</th>
</tr>
</thead>
<tbody>
<tr>
<td>P00590</td>
<td>1AGY, 1CEX, 1CUA, 1CUB, 1CUC, 1CUD, 1CUE, 1CUF, 1CUG, 1CUH, 1CUI, 1CUJ, 1CUS, 1CUU, 1CUV, 1CUW, 1CUX, 1CUY, 1CUZ, 1FFA, 1FFB, 1FFC, 1FFD, 1FFE, 1OXN, 1XZA, 1XZB, 1XZC, 1XZD, 1XZE, 1XZF, 1XZG, 1XZH, 1XZI, 1XZJ, 1XZK, 1XZL, 1XZM, 2CUT, 3ES3, 3ESA, 3ESB, 3ESC, 3ESD</td>
</tr>
<tr>
<td>P11373</td>
<td>3DCN, 3DD5, 3DEA</td>
</tr>
<tr>
<td>P52956</td>
<td>3GBS</td>
</tr>
<tr>
<td>None</td>
<td>5AJH, 4PSE, 4PSD, 4PSC, 4OYY, 4OYL, 4EB0, 4CG3, 4CG2, 4CG1, 3QPD, 3QPC, 3QPA</td>
</tr>
</tbody>
</table>

Table 2: Protein groupings for EC 3.1.1.74

The next step involves breaking down each protein by active site residues. To do this, the program references the Catalytic Site Atlas to determine the active sites for that protein. From there it instructs PyMOL to load that protein and to remove all residues that are not within that active site. Across all steps of direct manipulation of PDB files, it was important to filter out duplicate atoms for a given location resulting from x-ray crystallography. Within many protein structures, it was common for identical atoms to be repeated at identical locations that appear in the PDB files at ANISOU records. They did not cause inaccuracies within the algorithm, but had to be removed nonetheless as they caused errors within PyMOL commands. An example of duplicate structures can be found in Figure 6.
Profile Generation and Averaging of Distances and Angles

For each of the proteins within the EC/ID class, a profile is made showing the distance between each residue and the anchor and the angle formed by the carbon alpha of the anchor residue, the alpha carbon of the other residue, and the carbon beta of the other residue. The angle formed in relation to the anchor residue and other residues will be important for further transformations within the structure.

For each of these residues, all of these measurements are taken and averaged against each other with respect to the residue in which the measurement was made. The combination between all of the protein active site profiles allows us to easily account for substitutions within the structure. A count for the occurrence of a residue within each active site is also taken so that the program will know which residue is representative of the class and which is a substitution. Often, substitutions are outliers and only occur in a small percentage of structures. Still, they are also largely disparate in terms of distance and angle and, thus, can have a large effect on the average. While substitutions are not accounted for when averages are being calculated, they
are accounted for when the new motif is generated. Figure 7 shows the average statistics for EC 3.1.1.74 and UniProt ID P00590. This is an example of the residues and corresponding distances within the new motif.

![Figure 7: Average distances of various active site residues for EC 3.1.1.74 and UniProt class P00590. The top two lines are descriptions unique to the motif being generated. The first is telling the EC# and UniProt or PFam ID specific to the run. The second line shows the sample size that the new motif is drawing from. From there are two lines describing the common separated output for each of the residues across all of the proteins within the sample set.]

The active site residue is described as followed: residue number within the protein, the amino acid at that site, the distance between that residue and the anchor residue, the angle between that residue and the anchor, and the number of occurrences of that residue in that position within the class. Here, we see even though there are a number of potential residues for a given site (CYS, ALA, or SER at site 120, ALA or SER at site 42, and GLN or LEU at site 121), only one of the residues show real prevalence at a given sequence position, as shown in the counts.

**Transformation of the Template**

Now that the average sites are compiled for the given EC/ID, a model can be constructed. The active sites for a protein from the EC/ID class serve as the template for the finished motif. The protein chosen does not matter as long as it contains all of the consensus (not substitution) residues. From here, the residues that have a potential for substitutions to occur are cut down to the alpha and beta carbon. Visually, a residue with a potential for
substitution will be represented by a stick. This stick is composed of the alpha carbon, beta carbon, and the bond between them. The alpha-beta carbon substitution residue is noticeable in many residues, both within figures and appendices. With the residues represented as they need to be, it is time to physically move them into the position as defined by the averages for the class. This involves two steps: the shifting in respect to the average distance and the rotation with respect to the carbon alpha (anchor residue) - carbon alpha (other residue) - carbon beta (other residue) angle.

The shifting of residues involves moving each atom within the residue along a 3D vector in relation to the anchor. The basis of this 3D vector is the positioning of the alpha carbons of the anchor and other residue using the formulas shown in Figure 8.

\[
\begin{align*}
\text{Let } A & \text{ be the alpha carbon of the anchor residue} \\
\text{Let } O & \text{ be the alpha carbon of the other residue} \\
\text{Let } x, y, \text{ and } z & \text{ be different dimensions} \\
\vec{OA} = \vec{A} - \vec{O} = [A_x - O_x, A_y - O_y, A_z - O_z]
\end{align*}
\]

Figure 8: Definition of a 3D vector.

Once the 3D vector has been calculated, we create a translation vector to apply to each of the atoms within the other residue. The calculations for this are shown in Figure 9.

\[
\text{Let } B = \text{Average Distance between } A \text{ and } O - \text{Distance between } A \text{ and } O \text{ in the model protein} \\
\text{Translation Vector} = [B \times x, B \times y, B \times z]
\]

Figure 9: The translation vector to be applied to each atom in the other residue.

From here, that translation vector is applied to each atom within the other residue. The new position of the atoms is correct in respect to the distance. A majority of the time, this distance is a fraction of an angstrom.
With the distances changed, the last thing needed to complete the new motif is the rotation of the residue. Luckily, PyMOL has built in commands that allow us to get and set dihedral angles for a 4 atom system. Additionally, it allows you to compute it with only 3 atoms if the middle two atoms are the same. An illustration of a dihedral angle is in Figure 10. We can see the angle of rotation denoted as D.

![Figure 10: Dihedral Angle [14]](image)

If it was not possible to use the same two atoms as the edge of rotation, it would have been possible to create a pseudo atom along the same plane as the alpha carbons of the anchor and other residue and use that as the second atom from which to measure the angle of rotation. This was not needed due to leniency within PyMOL’s API. As with the differences in distances, the differences in angles were often very small.

![Figure 11: Motif for EC class 3.1.1.74 and UniProt class P00590 before (shown in blue) and after (shown in green) adjustment against the average residue distance and angle. Notice differences in location of identical](image)
residues across both structures. The stick-like connections are potential residue substitutions reduced down to the carbon alpha and carbon beta.

Despite the changes in distance and changes in rotation both being minute a large percentage of the time, it is important to create a motif that accurately represents the structure of the EC/ID class. An example of motif structure as it is transformed is shown in figure 11. In this figure, it is very easy to see the slight adjustments to the motif structure as it is finished.

Now that the model is transformed into the motif, it is saved for use by the user in the form of a PDB structure file.

**Evaluation of Motifs**

Motifs generated were evaluated on their ability to align to the protein set from which they were derived and the alignment to all other proteins in that EC class (regardless of whether or not they had known active sites). This was done through alignment methods built within PyMOL that superimpose 3D structures and calculate the average distance between atoms in that alignment.

The RMSD (root-mean-square deviation) is the metric by which the alignments are graded. The formula for RMSD is as follows (Coutsias, 2004). The RMSD within the PyMOL align function uses alpha carbons as the default pairing atoms for residues and is described in Figure 12.

\[
RMSD = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \delta_i^2}
\]

*Where \( \delta \) is the distance and \( N \) is the set of pairs of equivalent atoms*

Figure 12: Formula for RMSD calculation.
There is another built-in alignment algorithm in PyMOL called Super (PyMOL, 2015). This algorithm performs structural alignment with a series of refinement cycles that eliminate atoms of high variability. While this algorithm performs well on full protein structures, we cannot afford to eliminate atoms from alignment calculations when only working with a small number of residues.

The alignments between the newly generated motifs allowed us to quantify the selectivity of the motif. This was measured through the statistical analysis of those alignments and the visualization of those analyses on box and whisker plots. By using these plots, it is possible to visualize what makes a motif flagged as insufficiently selective.

**Significant Features of the Grading Plots**

The plot is used to represent the relationship between the alignment scores between the motif and the proteins of various sets. These protein sets consist of all of the possible PFam or UniProt classifications associated with a given EC#. For example, EC# 3.1.1.72 has two PFam families associated with it. The test for motifs associated with EC# 3.1.1.72 will align against 3 separate protein sets: one for proteins within PFam family PF00734, one for PF01083, and one to contain the remaining proteins not found in each.

The boxplot allows us to easily measure major features within the alignment scores across a large number of proteins with a single motif. Within a single plot, it is possible to measure the median alignment score, first and third quartile, and maximum and minimum bounds. Statistical outliers also appear as either hollow (probable outliers) or solid (true outliers) points.

There is a slight discrepancy between the visualization of the alignment scores and the way the algorithm determines the metric for significance. Where the plot displays the median, the algorithm uses the mean of the alignment scores. Despite this, for visual checking of flagged
motifs, the median can still be used as a guide. The way that the algorithm flags motifs is described in the next section.

**Relationship between Grading and Selectivity**

An alignment of a motif to the other proteins in its EC class was graded such that the average RMSD of alignments within that motif’s EC/ID class was outside of a standard deviation of the alignments between that motif and the proteins of other EC/ID classes. An illustration of this concept is shown in Figure 13 below. In the figure, we can see the self-match RMSD and variance compared to the alignment score between the motif and all other proteins in the dataset.

![Figure 13: The deviation in alignment scores for motif EC 3.1.1.74 and UniProt P00590.](image)

If the average alignment between the motif and the proteins within the same EC/ID class being checked was within one standard deviation of the average alignment between the motif and proteins found not to be in the same EC/ID class, it was flagged as a potential false
positive. As such, all of these motifs have to be hand checked. There are 3 possible situations that can result in a potential false positive: the motifs being checked may be identical, they may be sufficiently close in structure as to be a true false positive, or the structure can be so poor that it matches with the proteins of no known active site. The poorly generated structures seem to have little consensus and are easily flagged at this step. An example of a poor motif can be seen in the structure for 2.7.11.1_PF00069 (Appendix 13) in which there is a large difference between active site residues. These arise in situations where there is a large range in distances for a given residue between the proteins used to generate the motif. Of the poor motifs generated, most were found to be in homologous groups defined on much broader designations. This returns to the original problem where an EC# is not stringent enough to accurately define a motif. An example of a good motif, in terms of box and whisker plot, can be shown below, in figure 14. After testing, the process is complete and the motifs are ready for use in protein active site determination.

![Box plot of protein distances](image)

Figure 14: The average distances for proteins aligned to new motif 1.1.1.1_P11766. The red line is superimposed on the average to show the distinction from the other plots across the graph. An example of a poor motif can be found in Appendix 18.
Results

This program was tested across a variety of EC classes with the aim of covering a wide range of proteins in mind. Of the EC classes tested, 3.1.1.72 had 19 proteins associated with it, 3.1.1.74 had 74, 1.1.1.1 had 126, and 2.7.11.1 had over 1600. The sample set for 2.7.11.1 was limited to 250 proteins.

From these EC classes, 14 motifs were generated based on PFam families and 16 motifs were generated based on UniProt designations. Of the motifs generated across both designations, 19 were flagged as trouble motifs (either not selective enough or poor in structure, as shown in Table 4). Of these, 5 were found to be identical structures, 4 were found to be poorly generated motifs, 6 were found to be near misses, and 4 were found to be false positives. An example of an identical structure can be found on the in Figure 13. Identical structures occur when the proteins between separate PFam or UniProt families are structurally identical. In a perfect system, both of these families would be represented by a single family as there is not enough structural difference to determine which is which through 3D alignment. Near miss structures separated by PFam or UniProt designation that turned out to have near identical amino acid sequence at slightly different 3D conformation. An example of a near miss alignment can be found in Figure 16. There is little difference between identical structures and near-miss. With only a single amino acid being different between structures and near perfect structural alignment, near miss structures are all but identical. As such, near miss structures may be very closely related evolutionarily. Despite this, they may still be considered false positives if a protein for a given EC/ID family matches to its near miss alternative. The final result for all motifs generated is 28 accurate motifs derived from 469 proteins from across both UniProt and PFam designations. These motifs represent 4 UniProt families and 3 PFam families for EC 1.1.1.1, 4 UniProt families and 2 PFam families for 2.7.11.1, 1 UniProt and PFam family for 3.1.1.74, and 2 UniProt and 2 PFam families for 3.1.1.72. Table 3 shows this distribution.
<table>
<thead>
<tr>
<th>EC</th>
<th>Number of Proteins</th>
<th>Number of Motifs (PFam)</th>
<th>Number of Motifs (UniProt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1.74</td>
<td>19</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.1.1.72</td>
<td>74</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1.1</td>
<td>126</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2.7.11.1</td>
<td>Over 1600 (limited to 250)</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>469</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 3: Number of generated motifs across the sample datasets.

Figure 15: Identical structures 1.1.1.1_PF00107 (left, in blue) and 1.1.1.1_PF08240 (right, in white). Notice the identical positioning of partial residues within the motifs. Each of the lines within the active site motif represent the alpha carbon and the beta carbon for a given residue within the active site. Because these residues have a chance to contain substitutions, they are rendered down to the alpha carbon and beta carbon representations. The process for deciding partial residues is described above.
### Table 4

<table>
<thead>
<tr>
<th>Flagged Motif</th>
<th>Motif Flagged Against</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1.1_PF08240</td>
<td>1.1.1.1_PF00107</td>
<td>Identical Motifs</td>
</tr>
<tr>
<td>2.7.11.1_PF12796</td>
<td>2.7.11.1_PF4</td>
<td>Identical Motifs</td>
</tr>
<tr>
<td>2.7.11.1_PF12796</td>
<td>2.7.11.1_PF00023</td>
<td>Identical Motifs</td>
</tr>
<tr>
<td>2.7.11.1_PF00433</td>
<td>Numerous</td>
<td>Poor Motif</td>
</tr>
<tr>
<td>2.7.11.1_PF00169</td>
<td>Numerous</td>
<td>Poor Motif</td>
</tr>
<tr>
<td>2.7.11.1_PF12796</td>
<td>2.7.11.1_PF00531</td>
<td>Identical Motifs</td>
</tr>
<tr>
<td>2.7.11.1_PF12796</td>
<td>2.7.11.1_PF16095</td>
<td>Identical Motifs</td>
</tr>
<tr>
<td>2.7.11.1_PF00069</td>
<td>Numerous</td>
<td>Poor Motif</td>
</tr>
<tr>
<td>2.7.11.1_P53355</td>
<td>2.7.11.1_O15530</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_P31751</td>
<td>Numerous</td>
<td>Poor Motif</td>
</tr>
<tr>
<td>2.7.11.1_P53355</td>
<td>2.7.11.1_Q9</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_P53355</td>
<td>2.7.11.1_O14965</td>
<td>False Positive</td>
</tr>
<tr>
<td>2.7.11.1_O15530</td>
<td>2.7.11.1_P11309</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_O15530</td>
<td>2.7.11.1_O14757</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_P28523</td>
<td>2.7.11.1_Q9</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_P28523</td>
<td>2.7.11.1_O14965</td>
<td>False Positive</td>
</tr>
<tr>
<td>2.7.11.1_Q9</td>
<td>2.7.11.1_O14757</td>
<td>Near Miss</td>
</tr>
<tr>
<td>2.7.11.1_Q9</td>
<td>2.7.11.1_O14965</td>
<td>False Positive</td>
</tr>
<tr>
<td>2.7.11.1_O14965</td>
<td>2.7.11.1_P11309</td>
<td>False Positive</td>
</tr>
</tbody>
</table>

Table 4: Of the motifs generated, here are the designations of the motif and the motif it matched against. There are 4 possible results: identical motifs, a poor structure, a near miss in terms of structure, and a false positive that arises from similar alignment scores between a motif and a different protein set.
Discussion

Of the structures generated, a number of motifs turned out to be identical structures. The cause of this is clear. Within the PFam database, it is not uncommon for protein families to have the same function. For example, as shown in Figure 15, PF08240 is a catalytic domain containing zinc ions and PF00107 is simply a zinc binding dehydrogenase. While identical motifs are more commonly a result of PFam searches, they do occur in motifs derived from UniProt as well. Of the motifs that were flagged as potential false positives, all but one of them was identical to another motif that was flagged.

Disregarding identical structures, 14 of the 30 structures generated were flagged as problematic motifs. Motifs flagged this way fall into 3 categories: poor structures, near misses, and false positives. Poor structures are a result of poorly defined homologous groups. When the motif generation algorithm works through the averages between the distances of the amino acids in vastly different sites, the distances become vastly different. This results in a motif that does not accurately represent the active site it was built to represent. An example of this can be seen in Appendix 18 on page 51. These motifs are easily flagged as they have a large range of alignment scores even within their own motif set. Poorly generated structures represent 4 of the 14 problem motifs.
Near misses arise when two motifs can be described as nearly identical in terms of 3D structure and amino acid composition. An example of a near miss is shown in Figure 16. In this figure, we can see a near perfect alignment between motifs 2.7.11.1_Q9 and 2.7.11.1_O14757. Alignments between these motifs will be of more interest in future evolutionary studies, as they are so closely related despite being designated as separate PFam or UniProt families. Near misses make up 6 of the 14 motifs within the problem set. In terms of site characterization, it may be possible to lump near misses in with motifs of identical structure, due to the close similarity between the motifs.

The last source of flagging was false positives. These are caused by close alignment by ultimately different structures. An example of a false positive can be seen in Figure 17. Within this alignment, there is good superposition between 2 of 4 residues across both structures.
Despite the proper alignment of a handful of residues, these active sites are fundamentally different due to their composition in terms of the 3D positioning of the residues. An example of a false positive discovered by this flagging is further shown in Figure 18. In this graphic, there is no clear distinction between proteins belonging to P28523 and O14757 in terms of alignment scores between 2.7.11.1_P28523 and the proteins of each. Due to the small difference between alignment scores between both motifs, it would be tough to determine whether an active site would belong to one or the other algorithmically. False positives represent 4 of the 14 flagged structures.

Figure 17: Near miss alignment between motifs 2.7.11.1_P28523 (teal) and 2.7.11.1_O14757 (yellow). 2.7.11.1_P28523 consists of Aspartate 156, Lysine 158, Asparagine 161, Alanine 193, and Serine 194. 2.7.11.1_O14757 consists of Aspartate 130, Lysine 132, Glutamate 134, Asparagine 135, and Threonine 170.
Figure 18: Box and whisker plot showing the relative distribution of alignment scores for motif 2.7.11.1_O14757 against other motifs derived from EC 2.7.11.1. Notice the overlap in distributions between the average of 2.7.11.1_O14757 (far left) and 2.7.11.1_P11309 (mid left). The red line is superimposed on the average of 2.7.11.1_O14757.

There was the idea to use a set of randomly generated proteins to further test the false positive rate of the motifs. When matching against the set of proteins with no known active site within an EC class, if the structure was not poor, there were zero significant alignments between the motifs and this set. As a result of no alignments between motifs and the set of uncharacterized proteins with a known EC class, it is possible to say that the false positive rate between a motif and a completely unrelated protein (not within the same EC class) is effectively zero. Because motifs failed to match with uncharacterized proteins within it’s own EC class but not familial designation, it was determined that the random protein set was not needed.

The Issue of Variance

It was noticed that as the number of proteins used to generate a motif rises, the variance when matching this motif back to those proteins does as well. This is expected as when you adjust the template, you are making changes that may result in a template that will not match
any of the proteins used to build it. While these changes are small when using small EC/ID sets, they are more pronounced when working with large sets. The variance that arises within a motif can even be noticed in Figure 14 in which 1.1.1.1_P11766 has visible differences in the alignment scores between the motif and the proteins used to generate it (RMSD of around 0.5 on average). An example of a motif with small variance can be seen in 2.7.11.1_O14757, in which there is little visible variance the alignments between the motif and proteins within the same family.

Not necessarily variance, but the confidence in a motif, comes into question when the PFam family used to generate it is relatively broad. This can be seen in Figure 19.

![Figure 19: Large variance and poor matching within the motif 3.1.1.72_PF01083 as shown by the broad box and whisker plot and poor alignment score.](image_url)

In the specific example above, the PFam family described belongs to the singular function “cutinase”. As such, the PFam family is little more than stating the EC class. Drawing from the original premise of the problem, it is expected that motifs defined by insufficiently stringent EC/ID classes will be poor characterizations, as shown above. In cases like these, it is possible to discard the motif as a poor result and reexamine the PFam or UniProt class. If
enough research is done, it may be possible to petition for a new PFam or UniProt class to properly describe subfamilies within that protein set. From here, the algorithm can be reapplied to create stringent motifs. Of the motifs generated, 3 were discarded due to these issues:

3.1.1.72_PF01083, 2.7.11.1_PF00069, 2.7.11.1_PF00433. Of these 2.7.11.1_PF00069 and 2.7.11.1_PF00433 showed little to no consensus. Another one, 3.1.1.72_O59893 of the UniProt family, is on the border of having excessively poor self-alignment and high variance.

**Stringency**

Currently, our protein characterization software uses purely residue matching as an initial measure for hits (Hanson, 2014). While this process does work, it misses proteins that do not match directly to a template in cases of substitutions of residues within the active site. For example, imagining the two motifs in Figure 16 as an alignment between a motif and an uncharacterized protein, the algorithm may miss this alignment due to the differences in residues at the active sites despite the good 3D alignment. As such, this process uses the 3D alignment of the motif to the protein of interest as a metric of matching motif to protein in the hopes of determining function. In general, the motifs generated will not match with a reasonable RMSD to the proteins with the same function (as shown by the “None” set within the plots included in the appendices). While there are some outliers, the RMSD between a motif generated of a known function and an uncharacterized protein should be the new initial measure for that reason. Even in old analyses, we can see hits arrive at very high RMSD’s just to be filtered out immediately despite having an identical amino acid sequence.

Using RMSD as the new significance measure in motif-protein discovery is enhanced by the method of generalized motifs but it is also hindered by it. While there are a small number of false positives in terms of motifs matching to other proteins of the same function as determined by significantly different alignment averages, these alignments are still within 1 RMSD a significant portion of the time. This statement means that alignments of a generalized motif will
match pretty well to proteins of the same function but different PFam or UniProt designation. While this is an issue, it is secondary to the main goal of matching motifs to uncharacterized proteins, and can be dealt with manually down the line. Determining a positive match through 3D alignment is much more important than determining possible PFam or UniProt designations.

To summarize this section, 3D alignment should be the new metric from which motif-protein relationships are determined compared to residue sequence comparison. While the motifs generated through the methods described can sufficiently match only to the appropriate proteins, good matches may not resolve without manual checking at a UniProt/PFam level.

**Conclusion and Future plans**

At this point in development, the software can be considered fully functioning and ready for use. It has potential to advance projects ranging over a wide variety of user research interests. If one is more interested in further active site characterization, this group has identified that a potential application of this work could be to explore the idea that a motif can be described as just the alpha carbon and beta carbon atoms within its residues. Users may also be interested in evolutionary relationships between active sites as documented in the PFam database and shown within these motifs.

While documentation can be found below on how to run the program, some important notes must be taken into account. It is very necessary for the users to denote the minimum number of proteins that should be used to form the basis of a motif. Where it is possible to generate a motif for a UniProt class from a single protein, it is impossible to say if that protein will match further additions to that class as well. As such, the minimum number of proteins for a motif must scale with the number of proteins within that EC class. It is still possible to generate active motifs representing over 20 proteins.
Running the Software

Installation of the software requires PyMOL version 1.7.4 or newer and the code hosted on GitHub (https://github.com/CameronBaker/Thesis). Once downloaded, one needs to go into the plugin manager with PyMOL and select the file (ProcessProtein.py). The plugin manager is shown in Figure 20, with the “choose file” button highlighted.

![Figure 20: PyMOL plugin manager with “Choose file” button highlighted. From here, the user selects ProcessProtein.py and installs it by pressing “Install”.

From then on, the functions implemented will be available for the user. The R script, used to flag motifs as possibly bad or identical, is not necessary for running the motif generator, only for the verification of those motifs. There are 3 useful function for PyMOL available with ProcessProteins: ActiveSites, GenerateMotif, and EvalMotif.

ActiveSites takes a given protein and searches the Catalytic Site Atlas for known active sites. If active sites are found, PyMOL strips that protein structure down to its known active sites. This is shown in Figure 21.
Figure 21: The active site residues of 1GZN shown in magenta (left) on the backdrop of the rest of the protein. The image on the right shows the isolated active site as generated by ActiveSites.

GenerateMotif is the heavy lifter, actually generating the new motif based on user specifications. The parameters are as follows: EC number, MinimumProteins, MaximumProteins, Database, DataDump, and a working directory. The EC number is self-explanatory, the EC number associated with the function you are interested in. MinimumProteins sets the minimum number of proteins required to make a motif. The idea behind this parameter is that motifs with insufficient representation within an EC class will not accurately represent future proteins it may be matched against. It won’t be a real average if there are only one or two proteins to serve as a basis for the motif. If one of those residues is a substitution, in all other cases, the motif based on a single protein may not match others. MaximumProteins serves as a limiter for the sake of the user. While running the algorithm on EC 2.7.11.1 will generate a number of motifs across 1600 proteins, it will not make sense time wise if you are looking for simple results for testing. It is possible to remove any limit by passing in “none” at this parameter. The “Database” parameter refers to either UniProt or PFam as the metric to measure along with EC. The default database is UniProt unless the user passes in “pfam”.

DataDump is a true or false indicating whether or not the user would like to save all structures related to the motif generation. While not particularly necessary for normal use, it is
required for statistical evaluation. If the protein has known active sites, the method only saves those active sites. Otherwise, it saves the whole structure. WorkingDirectory is self-explanatory. The default working directory (within Windows) is not normally user writeable unless the user is running with administrator permissions. As such, it will not be able to save your motifs if you are running from this directory. It is recommended that you create a folder to work out of with this software to organize the location of your motifs.

The limiting factor in this software is the web crawling for structures and active sites. Users can expect a run time of about 3 seconds per protein (5 minute for an EC class with 100 proteins in it). All of the mathematical functions performed by PyMOL take a fraction of a second each.

EvalMotif is by and large an extension of GenerateMotif. It takes the EC, Database, and WorkingDirectory. The database is passed in as “UP” for UniProt or “PF” for PFam. It computes the alignments between all motifs and all active sites or full proteins if there are no active sites to be found. From here, the user can run a standalone script, stats.R (located within my github), that will flag problem motifs and visualize relationships as a box and whisker plot.

Acknowledgements

Mikhail Osipovitch for the original idea of automating motif generation. Before his work, all motif generation was done by hand picking significant residues

J. Nick Fisk for his help with editing, both in the fleshing out of ideas and the better understanding of biological concepts.
References


Appendices

Motifs Generated and Respective Plots

A1: Motif 1.1.1.1_P00325, structure and plot
A2: 1.1.1.1_P00327, structure and plot
A4: 1.1.1.1_P11766, structure and plot
A5: 1.1.1.1 P39462, structure and plot
A6: 1.1.1.1 PF00106, structure and plot

PF: 1.1.1.1_PF00106
A7: 1.1.1.1_PF00107, structure and plot

PP: 1.1.1.1_PF00107

![Box plot showing RMSD for different protein designations.](image-url)
A8: 1.1.1.1_PF08240, structure and plot
A9: 2.7.11.1 O14757, structure and plot
A10: 2.7.11.1_P11309, structure and plot

UP: 2.7.11.1_P11309

[Graph showing data distribution with box plots for different protein designations]
A11: 2.7.11.1_P28523, structure and plot
A12: 2.7.11.1 Q9, structure and plot
A16: 3.1.1.72_PF00734, structure and plot

PF: 3.1.1.72_PF00734
A18: 3.1.1.72_Q99034, structure and plot

UP: 3.1.1.72_Q99034

RMSD

Protein Designation
A19: 3.1.1.74_P00590, structure and plot

UP: 3.1.1.74_P00590

RMSD

Protein Designation

3.1.1.74_P00590  None
Code

ProcessProteins.py

```
This file contains the methods responsible for obtaining
the information required from a given protein.
It includes methods to discern the functional amino acids
from the protein as found in the CSA as well as details specific
to those amino acids such as what residue it is and it's relation
to others within the functional group

Cameron Baker
```

```python
from pymol import cmd
from pymol import stored
import urllib2
import re
import os
from requests import exceptions

def ActiveSites(prot):
    """
    This function implements a command into pymol that allows the user to
    isolate active sites for a given protein from the pdb file as per locations
denoted within the catalytic site atlas. It involves the parsing of the
locations from the entry for the protein within the atlas.

    If active sites are found, a new object is built within pymol containing
    only those residues in their proper orientation

    No action if no active sites are found
    """
    cmd.fetch(prot)
    ActiveRes = []
    response = urllib2.urlopen('https://www.ebi.ac.uk/thornton-srv/databases/CSA/SearchResults.php?PDBID=%s' % prot)
    inchain = True
    for line in response:
        exp = '<td>' + '.' * i + '</td>'
        CurRes = re.findall(exp, line)
        for res in CurRes:
            res = res.split('>')[-1].split('<')[0]
            if inchain:
                ActiveRes.append(res)
            else:
                inchain = True

    for line in response:
        exp = '<td>' + '.*i+'</td>'
        CurRes = re.findall(exp, line)
        for res in CurRes:
            res = res.split('>')[1].split('<')[0]
            if inchain:
                ActiveRes.append(res)
            else:
                inchain = True
```

if res.isdigit() and inchain:
    ActiveRes.append(res)
    inchain = False
else:
    inchain = True

# Only keeps unique residues
ActiveRes = list(set(ActiveRes))

# Align against the full protein if there are no active residues
# As long as one protein has an active site it should align well
if len(ActiveRes) != 0:
    cmd.remove('%s and not chain a%(prot)' % (prot))
    SelectStatement = '+'.join(ActiveRes)
    cmd.create('%s_active%(prot)', 'resi %s in %s' % (SelectStatement, prot))
    cmd.delete(prot)
    # print "Active sites loaded"
    return 1
else:
    cmd.delete(prot)
    print "no active sites found for "+prot
    return 0
response.close()

cmd.extend("ActiveSites", ActiveSites)
print "ActiveSites pdbid"

def SiteDistances(prot):
    ""
    This function implements a command within pymol that will find the
    centermost residue within a protein. It works by reading out all
    of the distances between a given residue's alpha carbon and each other
    residue's alpha carbons. The residue with the smallest average distance
    across all residues will be the centermost.
    
    Once the centermost residue is determined, the angle between that
    residue's alpha carbon and each other residue's alpha and beta
    carbons will be determined and returned to the user.
    ""

    # Retrieve the list of residues
    stored.list = []
    cmd.iterate("(%s and name ca)" % (prot), "stored.list.append((resi, resn))")
    # print stored.list

    # Initialization of variables for finding the centermost residue
    res_num = len(stored.list)
    closest_res_res = 0
    smallest_dist = 100
    for res in stored.list:
        temp_dist = 0
        for res2 in stored.list:
            try:
                if res[0] != res2[0]:
                    temp_dist = temp_dist + 1
                    cmd.get_distance("resi %s in %s and name ca" % (res[0], prot))
                    break
            except:
                break
        if temp_dist < smallest_dist:
            closest_res_res = res[0]
            smallest_dist = temp_dist
    print "Centermost residue found at %s")" % (closest_res_res, prot)

    # Calculate angles
    angle_list = []
    for res in stored.list:
        if res[0] == closest_res_res:
            for res2 in stored.list:
                try:
                    if res[0] != res2[0]:
                        angle_list.append(cmd.get_dihedral("resi %s in %s and name ca" % (res[0], prot),
                                "resi %s in %s and name ca" % (res2[0], prot)))
                        break
                except:
                    break
    angle_list.sort()
    print "Centermost residue angles: ", angle_list
    return angle_list
and name ca"%(res2[0],prot))
    except:
        stored_tmp = stored.list
        if res[0] != res2[0]:
            stored.list = []
        cmd.iterate(\"(resi %s in %s and name ca)\"%(res2[0],prot),\"stored.list.append((ID))\")
        if len(stored.list) > 1:
            cmd.remove(\"id %s\"%(stored.list[-1]))
        stored.list = []
        cmd.iterate(\"(resi %s in %s and name ca)\"%(res[0],prot),\"stored.list.append((ID))\")
        if len(stored.list) > 1:
            cmd.remove(\"id %s\"%(stored.list[-1]))
    temp_dist = temp_dist + \
        cmd.get_distance("resi %s in %s and name ca\"%(res2[0],prot),\"resi %s in %s and name ca\"%(res[0],prot))
    stored.list = stored_tmp
    temp_dist = temp_dist / (res_num - 1)
    if temp_dist < smallest_dist:
        smallest_dist = temp_dist
        closest_res = res
        #print closest_res
        #print smallest_dist

    #like protein + profile, get it?
    PROfile = {}  

    #The following calculates the distance between the anchor residue and others
    #It then sorts the ordering and then calculates angles
    for res in stored.list:
        dist = cmd.get_distance("resi %s in %s and name ca\"%(closest_res[0],prot),
                                "resi %s in %s and name ca\"%(res[0],prot))
        PROfile[res[0]] = dist
        PROfile = sorted(PROfile.items(), key=lambda x: x[1])
    for res in stored.list:
        for site in range(0,len(PROfile)):
            try:
                if res[1] == "GLY":  
                    angle = cmd.get_angle("resi %s in %s and name ca\"%(closest_res[0],prot),
                                            "resi %s in %s and name ca\"%(res[0],prot),
                                            "resi %s in %s and name cb\"%(res[0],prot))
                else:
                    angle = cmd.get_angle("resi %s in %s and name ca\"%(closest_res[0],prot),
                                            "resi %s in %s and name ca\"%(res[0],prot),
                                            "resi %s in %s and name cb\"%(res[0],prot))
            except:
                print "Duplicate atoms found for a given carbon beta. Deleting duplicate"
                stored.list = []
                cmd.iterate(\"(resi %s in %s and name ca\"%(res[0],prot),\"stored.list.append((ID))\")
                if len(stored.list) > 1:
                    cmd.remove(\"id %s\"%(stored.list[-1]))
                    cmd.iterate(\"(resi %s in %s and name cb)\"%(res[0],prot),\"stored.list.append((ID))\")
                if len(stored.list) > 1:
cmd.remove("id %s"%(stored.list[-1]))
if res[1] == "GLY":
    angle = cmd.get_angle("resi %s in %s and name ca"%(closest_res[0],prot),
                        "resi %s in %s and name ca"%(res[0],prot),
                        "resi %s in %s and name n"%(res[0],prot))
else:
    angle = cmd.get_angle("resi %s in %s and name ca"%(closest_res[0],prot),
                        "resi %s in %s and name ca"%(res[0],prot),
                        "resi %s in %s and name cb"%(res[0],prot))

if res[0] == PROfile[site][0]:
    PROfile[site] = res,PROfile[site][1],angle

#Uncomment the following for numbers related to the residues in the structure
# print "anchor_residue,anchor_num,target_residue,target_num,distance,angle"
# for res in PROfile:
#     print "%s,%s,%s,%s,%s,%s"%(closest_res[1],closest_res[0],res[0][1],res[0][0],res[1],res[2])

#Uncomment the following 4 lines for visual representation of distances between centermost residue
# for res in stored.list:
#     if res != closest_res:
#         cmd.distance("resi %s in %s and name ca"%(closest_res[0],prot),
#                       "resi %s in %s and name ca"%(res[0],prot))

#Comment out the delete command to keep residues after quantification. Good for visualization
# cmd.delete("all")
return PROfile

cmd.extend("SiteDistances",SiteDistances)

def UPfromProt(pdb):
    ""
    This function queries the uniprot database for a specific protein
    and returns the uniprot families that it belongs to
    ""
    try:
        response = urllib2.urlopen("https://www.ebi.ac.uk/thornton-srv/databases/CSA/SearchResults.php?PDBID=%s"%(pdb),timeout=20)
    except:
        try:
            response = urllib2.urlopen("https://www.ebi.ac.uk/thornton-srv/databases/CSA/SearchResults.php?PDBID=%s"%(pdb),timeout=20)
        except:
            print "Timeout"
            return 0
    UPList = []

    # Parse the Uniprot family out of the webpage
    try:
        for line in response:
            exp = "UNIID=[A-Z][0-9]*"
            curLine = re.findall(exp,line)
            if len(curLine) > 0:
                UPList.append(curLine[0].split("=")[1])
if len(UPList) is 0:
    return 0
except:
    return 0
return UPList

cmd.extend("Prot2UP", UPfromProt)

def PFfromProt(pdb):
    ""
    The search for pfam ID's requires querying uniprot with a uniprot ID and taking the associated pfam ID's
    for the given uniprot
    ""
    UPList = UPfromProt(pdb)
    if UPList == 0:
        return 0
    pfamID = []
    for id in UPList:
        try:
            response = urllib2.urlopen("http://www.uniprot.org/uniprot/%s"%(id),timeout=20)
        except:  # The second try is redundant
            response = urllib2.urlopen("http://www.uniprot.org/uniprot/%s"%(id),timeout=20)
            except:
                return 0
        for line in response:
            exp = "PF\d+"
            curLine = re.findall(exp, line)
            if len(curLine) > 0:
                for pfam in curLine:
                    if pfam not in pfamID:
                        pfamID.append(pfam)
    return pfamID

cmd.extend("Prot2pfam", PFfromProt)

def ECtoProt(EC):
    ""
    This function retrieves the list of proteins for a given EC from the protein data bank
    ""
    url = 'http://www.rcsb.org/pdb/rest/search'
    ECList = []

    # Query the PDB to obtain a list of proteins for a given EC
    queryText = ""
    <orgPdbCompositeQuery version="1.0">  
      <queryRefinement>
        <queryRefinementLevel>0</queryRefinementLevel>
        <orgPdbQuery>
          <version>head</version>
          <queryType>org.pdb.query.simple.EnzymeClassificationQuery</queryType>
          <description>Enzyme Classification Search : EC=%s</description>
    ""
<Enzyme_Classification>%s</Enzyme_Classification>
</orgPdbQuery>
</queryRefinement>
</orgPdbCompositeQuery>

```python
req = urllib2.Request(url, data=queryText)
queryResult = urllib2.urlopen(req)

#Parse the query into a usable list
for line in queryResult:
    protein = line.strip()
    ECList.append(protein.split("":"\[0\])

return ECList

def adjustRes(anchoanchor, res, distance, angle, protname):
    ""
    This method is responsible for translating the distance and angle
    between the anchor and another amino acid into the correct distance
    ""
    stored_list = []
    if res[0] == anchor[0][0]:
        return

    #A good chunk of the following is responsible for filtering out duplicate atoms from pdb structures
    cmd.iterate("(resi %s in %s and name ca)"%(res[0],protname),"stored_list.append((ID))")
    if len(stored_list) > 1:
        cmd.remove("id %s"%(stored_list[0]))
    stored_list = []
    cmd.iterate("(resi %s in %s and name ca)"%(anchor[0][0],protname),"stored_list.append((ID))")
    if len(stored_list) > 1:
        cmd.remove("id %s"%(stored_list[0]))

    #Retrieves the distance between the anchor and other residues
    dist = cmd.get_distance("resi %s in %s and name ca"%(anchor[0][0],protname),
                            "resi %s in %s and name ca"%(res[0],protname))
    #More residue cleaning
    stored_list = []
    cmd.iterate("(resi %s in %s and name cb)"%(res[0],protname),"stored_list.append((ID))")
    if len(stored_list) > 1:
        cmd.remove("id %s"%(stored_list[0]))

    #Retrieves the 3D coordinates of the alpha carbons in both residues
    anchor_ca_coord = cmd.get_coords("resi %s in %s and name ca"%(anchor[0][0],protname))
    other_ca_coord = cmd.get_coords("resi %s in %s and name ca"%(res[0],protname))
    #Finds the translation vector
    vx = anchor_ca_coord[0][0] - other_ca_coord[0][0]
    vy = anchor_ca_coord[0][1] - other_ca_coord[0][1]
    vz = anchor_ca_coord[0][2] - other_ca_coord[0][2]
    res_vector = [vx,vy,vz]
```
# Apply that vector to each atom in the other residue
```
cmd.translate(translation_vector,"resi %s in %s\%(res[0],protname))
```

# Sets the dihedral angle to be the consensus
```
if res[1] == "GLY":
    cmd.set_dihedral("resi %s in %s and name ca\%(anchor[0][0],protname),
                   "resi %s in %s and name ca\%(res[0],protname),
                   "resi %s in %s and name ca\%(res[0],protname),
                   "resi %s in %s and name n\%(res[0],protname),angle
else:
    cmd.set_dihedral("resi %s in %s and name ca\%(anchor[0][0],protname),
                   "resi %s in %s and name ca\%(res[0],protname),
                   "resi %s in %s and name ca\%(res[0],protname),
                   "resi %s in %s and name cb\%(res[0],protname),angle
```

def generateMotif(EC,entry_minimum,protein_cap,db,dump,outdir):
    ```
cmd.cd(outdir)
proteinList = ECtoProt(EC)
UPDictionary = {}
using_windows = False
if os.name == "nt":
    using_windows = True

if dump == "true":
    outfile = open('output.txt','w')

# Information for the user based on the initial retrieval of proteins
print "%s proteins found for %s\%(len(proteinList),EC)
print "Beginning protein grouping"
count = 0

# Used to contain information on incomplete records (no link between catalytic site atlas and UniProt)
NullCount = 0
NullList = []

# Subset the protein list for now
if protein_cap != "none":
    if len(proteinList) > int(protein_cap):
        proteinList = proteinList[0:int(protein_cap)]

# The following loop populates the proteins into their respective groups
# Groups are determined by pfam or uniprot ID, as shown on a couple lines down
for protein in proteinList:
    if db == "pfam":
        UPID = PFfromProt(protein)
    else:
        UPID = UPfromProt(protein)

    if UPID is 0:
        count = count + 1
        if count % 5 == 0:
            print "%s\%(count,len(proteinList))"
NullCount = NullCount + 1
NullList.append(protein)
continue
if len(UPID) > 1:
    for ID in UPID:
        if ID not in UPDictionary.keys():
            UPDictionary[ID] = protein
        else:
            UPDictionary[ID] = UPDictionary[ID]+","+protein
else:
    if UPID[0] not in UPDictionary.keys():
        UPDictionary[UPID[0]] = protein
    else:
        UPDictionary[UPID[0]] = UPDictionary[UPID[0]]+","+protein

count = count + 1
if count % 5 == 0:
    print "%s/%s"%(count,len(proteinList))

print "%s proteins with incomplete records"%NullCount

for entry in UPDictionary.keys():
    print "%s: %s"%(entry,UPDictionary[entry])
    if dump == "true":
        outfile.write("%s: %s"%(entry,UPDictionary[entry]))
        outfile.write("\n")

ProteinSet = UPDictionary[entry].split(",")
ProfileSet = list(set(ProteinSet))
ProfileSet = []
DistDict = {}
AngleDict = {}
count = 0
if not int(entry_minimum) < len(ProteinSet):
    print "Not enough proteins for entry %s"%entry
    continue

for protein in ProteinSet:
    #This generates the active sites and quantifies the distances and angles
    #Only keeps the active site if it successfully generated
    #Successful generation is due to the active site presence within the CSA
    count = count + 1
    if ActiveSites(protein) == 1:
        protein = protein + ",active"
        if dump == "true":
            cmd.save("%s.pdb"%protein,"%s"%protein)
            PROfile = SiteDistances(protein)
            ProfileSet.append(PROfile)
        if count % 5 == 0:
            print "%s/%s"%(count,len(ProteinSet))

    #Uncomment the following to print out the distances and angles for a given protein
    #for res in PROfile:
    #    print res
    print "%s/%s"%(len(ProteinSet),len(ProteinSet))

if len(ProfileSet) > 0:
# Below is finding the average distance and angle between each residue within the active site
consensusDic = []
anchor = []
anchorset = []
ResCount = {}
for PROfile in ProfileSet:
    for res in PROfile:
        key = res[0]
        if res[1] == 0:
            anchorset.append(res)
        if key not in DistDict.keys():
            DistDict[key] = res[1]
        else:
            DistDict[key] = (DistDict[key] + res[1]) / 2
        if key not in AngleDict.keys():
            AngleDict[key] = res[2]
        else:
            AngleDict[key] = (AngleDict[key] + res[2]) / 2
        if key not in ResCount.keys():
            ResCount[key] = 1
        else:
            ResCount[key] = ResCount[key] + 1
print """ 
print "------------------------------"
print "MOTIF STATS for %s_%s:"%(EC,entry)
print "Different stats: %s across %s proteins"%(str(len(DistDict.keys())),len(ProteinSet))
print "Anchor residue will have a Distance of 0 and an angle of 90" 
print "Residue Position, Residue, Distance from anchor, Angle, Count"
for key in DistDict:
    if len(anchorset) > 0:
        anchormax = 0
        for res in anchorset:
            if ResCount[key] > anchormax:
                anchormax = ResCount[key]
                anchor = res

consensusDic.append([(key[0],key[1]),DistDict[key],AngleDict[key],ResCount[key]])
for line in consensusDic:
    print "%s,%s,%s,%s,%s"%(line[0][0],line[0][1],line[1],line[2],line[3])
model = UPDictionary[entry].split(";")[0]
count = 0
while True:
    if ActiveSites(model) == 1:
        stored.list = []
cmd.iterate("(%s and name ca)"%(model),"stored.list.append((resi,resn))")
    if anchor[0] not in stored.list:
        model = UPDictionary[entry].split(";")[count]
count = count + 1
cmd.remove('all')
break
count = count + 1
if count >= len(UPDictionary[entry].split(',')):
    print 'No active sites found'
    break
model = UPDictionary[entry].split(',')[count]

if count == len(UPDictionary[entry].split(',')):
    cmd.remove('all')
    continue

modellInfo = SiteDistances('%s_active'%model)
modelSites = []
ActiveSites(model)

for site in modellInfo:
    tmp = [site[0][0],site[0][1]]
    modelSites.append(tmp)

ActiveSite = []
SubstituteSet = []
ResMax = {}
ResMap = {}
for key in consensusDic:
    if key[0][0] not in ResMax:
        ResMax[key[0][0]] = key[3]
        ResMap[key[0][0]] = key[0][1]
    else:
        if ResMax[key[0][0]] < key[3]:
            ResMax[key[0][0]] = key[3]
            ResMap[key[0][0]] = key[0][1]

newCon = []
for key in consensusDic:
    if key[0][1] == ResMap[key[0][0]]:
        newCon.append(key)
    else:
        if key[0][0] not in SubstituteSet:
            SubstituteSet.append(key[0][0])

consensusDic = newCon
for site in SubstituteSet:
    cmd.remove('resi %s and not name ca+cb' %site)
if dump == "true":
    cmd.save('%s_%s_pre.pdb' %(EC,entry),'%s_active'%model)

for key in consensusDic:
    if key[0] in modelSites:
        adjustRes(adjustRes, key[0], key[1], key[2], model)

    cmd.save('%s_%s.pdb' %(EC,entry),'%s_active'%model)
    cmd.remove('all')

for prot in UPDictionary[entry].split(','):
if using_windows:
    os.system("del %s.pdb"%prot)
else:
    os.system("rm %s.pdb"%prot)

print ""
print "%s_%s written"%(EC,entry)
print ""
else:
    print "skipping %s. No usable proteins found"%(entry)
    UPDictionary.pop(entry)

if dump == "true":
    nullstr = ""
    for prot in NullList:
        cmd.fetch(prot)
        cmd.remove(prot)
        nullstr = prot+","+nullstr
    nullstr = nullstr[:-1]
    outfile.write("None: %s"%nullstr)
    outfile.close()

    cmd.remove("all")

print "done!"

    cmd.extend("GenerateMotif",generateMotif)
    print "GenerateMotif EC, MinProt, MaxProt, pfam/uniprot, DataDump, OutputDirectory"

    def EvalMotif(motif, db, dir):
        
            This method takes data dumped motifs, active sites, and protein structures
            and measures alignments
            
        
            if str(db) not in "PFUP":
                print "Enter the database argument as UP for UniProt or PF for PFam"
                return
            cmd.cd(dir)
            file = open("output.txt",'r')
            outfile = open("%s_%s_test.txt"%(motif,db),'w')
            MotifDict = {}

            for line in file:
                print line
                mname = line.split(' ')[0]
                mname = mname[:-1]
                if mname != "None":
                    mname = motif + "_*" + mname
                    protlist = line.split(' ')[1]
                    if "n" in protlist:
                        protlist = protlist[:protlist.index("n")]
                        protlist = protlist[:-2]
                    MotifDict[mname] = protlist

            print MotifDict
            for key in MotifDict:
if key == "None":    
    continue
try:    
    cmd.load("%s.pdb"%key)
except:    
    print "No motifs found for %s"%(key)    
    continue
for OKey in MotifDict:    
    protList = MotifDict[OKey].split("
for prot in protList:      
    if OKey == "None":    
        cmd.load("%s.pdb"%prot)    
        aln = cmd.align("%s"%key,"%s"%prot)    
        sup = cmd.super("%s"%key,"%s"%prot)    
        print "%s,%s,%s,%f,%f"%(key,OKey,prot,aln[0],sup[0])    
        outfile.write("%s,%s,%s,%f,%f\n"%(key,OKey,prot,aln[0],sup[0]))    
        cmd.remove("%s"%prot)      
    else:    
        try:    
            cmd.load("%s_active.pdb"%prot)    
        except:    
            print "It's not your fault"
        aln = cmd.align("%s"%key,"%s_active"%prot)    
        sup = cmd.super("%s"%key,"%s_active"%prot)    
        print "%s,%s,%s,%f,%f\n"%(key,OKey,prot,aln[0],sup[0])    
        outfile.write("%s,%s,%s,%f,%f\n"%(key,OKey,prot,aln[0],sup[0]))    
        outfile.close()    
        cmd.remove("%s_active"%prot)    
        except:    
            print "No motifs found for %s"%(key)
            try:    
                cmd.remove("%s_active.pdb"%prot)    
            except:    
                print "It's not your fault"
        cmd.remove("all")    
    print "done!"
file.close()    
outfile.close()    
    cmd.extend("EvalMotif",EvalMotif)    
print "EvalMotif EC, db, DumpDirectory"

stats.R

#Cameron Baker
#This R script handles the generation of statistical results related
#to the alignment scores for motifs passed in.
#Requires a full run data dump of the motif generator
#Simple pipe is as follows
#GenerateMotif -> EvalMotif -> boxplot_motifs

#Output is a number of boxplots relating to alignment scores
#and the flagging of motif pairs that are not significantly different into an outfile

boxplot_motifs <- function(filename){    
    df <- read.csv(filename,header=FALSE)    
    if(length(grep("pf",filename)) > 0){    
        desig <- "PF"
} else{
  desig <- "UP"
}

motifs <- unique(df[,1])
for(motif in motifs){
  single_motif = df[which(df[,1] == motif),c(2,4,5)]
  num_motifs = length(unique(single_motif[,1]))
  title <- gsub("[.]", ",",motif)
  
  other_motifs = unique(single_motif[,1])[which(unique(single_motif[,1]) != motif)]
  for(o_m in other_motifs){
    rmsd_set = single_motif[which(single_motif[,1] == motif),"V4"]
    rmsd_set_other = single_motif[which(single_motif[,1] == o_m),"V4"]
    set_mean = mean(rmsd_set)
    q <- quantile(rmsd_set_other,c(0.25,0.75))
    if(set_mean > q[1] &amp; set_mean < q[2]){  
      write(paste(desig,motif,o_m),"tocheck.txt",append=TRUE,sep="\n")
    }
  }
  
  jpeg(paste(title,".jpg",sep=""),width=200*num_motifs)
  boxplot(V4~V2,data=single_motif,ylim=c(0,2),main=paste(desig,"::",motif,sep=""),
  xlab="Protein Designation",ylab="RMSD")
  dev.off()

  }
}

print("Done!")

}

file.remove("tocheck.txt")
flist <- list.files()[grep(".txt",list.files())]
for(file in flist){
  boxplot_motifs(file)
}