UNIVERSITY OF CALIFORNIA, SAN DIEGO

Characterizing Protein Structures by Surface Mapping

A dissertation submitted in partial satisfaction of the requirements for the degree
Doctor of Philosophy

in

Bioinformatics

by

Elaine Ellen Thompson

Committee in charge:

Professor Lynn F. Ten Eyck, Chair
Professor Vineet Bafna, Co-Chair
Professor J. Lawrence Carter
Professor J. Andrew McCammon
Professor Palmer Taylor
Professor Susan S. Taylor

2008

The Dissertation of Elaine Ellen Thompson is approved and it is acceptable in quality and form for publication on microfilm and electronically.

Co-Chair

Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Lynn Ten Eyck for his support as my advisor and chair of my committee. His support has been unwavering and his guidance invaluable. I have learned an immense amount by working with him.

I would also like to acknowledge Dr. Susan Taylor. I could not have completed this work without her help and guidance. Once I had the computational method developed, she helped me apply it to Protein Kinase A and the kinase family.

The description of Crevasse and FADE in Chapter 2 and the PKA results in Chapter 3 will be written up for publication. The dissertation author will be the primary investigator and author. Dr. Choel Kim provided the new high-resolution crystal structure of PKA used in Chapter 3 and Mike Pique helped with the optimization of FADE detailed in Chapter 2. Both will be co-authored. Dr. Susan Taylor and Dr. Lynn Ten Eyck will also be co-authors on the paper.

The results in Chapter 4 will be published as a new method for comparing protein structures. The dissertation author will be the primary investigator and author. Dr. Kannan Natarajan kindly provided me with F-helix sequences for aligning the kinases in Chapter 4, and both he and Dr. Alexandr Kornev helped me design and interpret the experiment. Both will be co-authored along with Dr. Susan Taylor and Dr. Lynn Ten Eyck.

VITA

1990	Bachelor of Science in Clarinet Performance and Biology Indiana University, Bloomington IN
1996	Master of Arts, Department of Neuroscience The Johns Hopkins University School of Medicine, Baltimore, MD
	Thesis: Reconstitution and Characterization of the Peripheral Benzodiazepine Receptor
1996-2000	Scientist, Trega Biosciences, San Diego
2000-2001	Database Consultant
2008	Doctor of Philosophy, University of California, San Diego

ABSTRACT OF THE DISSERTATION

Characterizing Protein Structures by Surface Mapping

by

Elaine Ellen Thompson

Doctor of Philosophy

University of California, San Diego, 2008

Professor Lynn F. Ten Eyck, Chair
Professor Vineet Bafna, Co-Chair

As the availability of protein crystallographic structures grows, new methods are needed to characterize, compare, and interpret crystal structure data. Surface geometry is important in molecular function, as interactions with other molecules in the cell happen at the protein surface. This dissertation describes a method of mapping and comparing surfaces of crystallographic structures, using the protein kinase family as a model. Pockets are rapidly computed using two pieces of software, FADE and Crevasse. FADE uses gradients of atomic density to locate grooves and pockets on the molecular surface. Crevasse, a new piece of software, segments the FADE output into distinct pockets that can be used in computations. A map of pockets on the catalytic core of Protein Kinase A shows the ATP and peptide docking

sites, locations for C-helix anchoring, the myristylation site, and sites for regulatory subunit interaction. There are other sites identified where Protein Kinase A is likely to interact with other proteins, but the binding partners have not been identified. Spatially clustering pockets across a family of aligned proteins can identify similarities and differences within the family. In the set of ten kinase cores studied, differences in the active site cleft between serine/threonine and tyrosine kinases are visible. Shared mechanisms of C-helix anchoring are also evident, and a novel site at the top of the N-lobe is present in all the kinases. There are other pockets on the kinases that are strongly conserved but have not yet been mapped to a protein-protein interaction.

Chapter 1. Introduction

With the advent of genomics and high-throughput crystallography, protein crystal structures are becoming widely available. The National Institute of General Medical Sciences Protein Structure Initiative (www.nigms.nih.gov/Initiatives/PSI) has been providing grant support for structural biology projects. The stated long-term goal of the project is to "make the three-dimensional atomic-level structures of most proteins easily obtainable from knowledge of their corresponding DNA sequences". The first step in the PSI initiative is to make a broad sample of X-ray crystallography and NMR structures available in order to understand how proteins are folded. The PSI is funding fourteen dedicated protein crystallography centers in an effort to advance the crystallography field and increase the number of crystal structures available. Completed crystal structures are deposited in the Worldwide Protein Data Bank (http://www.wwpdb.org/), an international structural data repository. Figure 1-1 shows the exponential growth in the number of structures archived in the RCSB Protein Data Bank over the past twelve years.

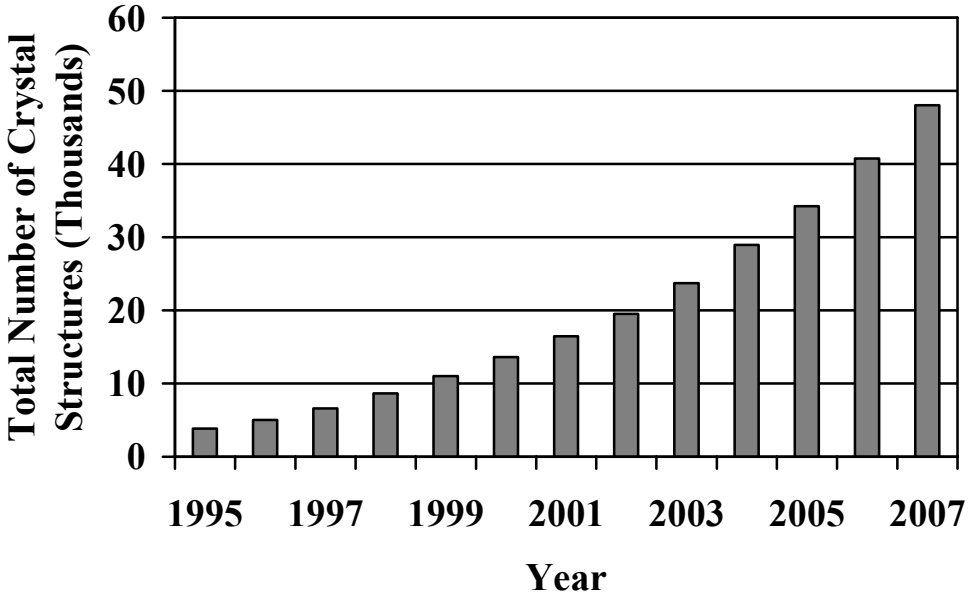

Figure 1-1. Growth of the PDB Across Twelve Years.
The graph shows the total number of searchable structures in the Protein Data Bank from 1995 to 2007.

Analysis of crystal structures can give insight into the structure and function of proteins that is not available from simple sequence analysis. The more structures that are available, the more important developing analysis tools becomes. Simple inspection and comparison of thousands of diverse structures is no longer possible. Here, a new methodology for comparing and contrasting pockets and crevices on the surfaces of protein crystal structures is introduced. Computationally identified pockets can be clustered, and similarities and differences on the surfaces of structurally related proteins mapped.

The first generation of protein comparisons was sequence-based. Even now, sequence homology is a useful first step in predicting similarity of structure and function. Known sequence domains can be used to rapidly classify a protein into

broad families. PFAM (Finn et al. 2006) is one of the more widely used sequence-based classification tools. Unfortunately sequence comparison is limited in the absence of crystal structures. Sequence is a weak predictor for secondary structure, and neither secondary structure nor sequence presently predicts folds reliably. Because of the importance of fold in protein function, structure based comparison tools for crystal structures were built as well. The SCOP classification database (Murzin et al., 1995) classifies proteins into families by fold and is a powerful tool for identifying proteins with conserved structural domains. CATH (Greene et al., 2007) is another tool that automatically assigns domains to new structures that are deposited into the PDB. Using these tools, scientists can often classify structures into subfamilies, identify evolutionarily related proteins, and make predictions about function based on known subdomains.

Analysis of primary structure and secondary structure is of limited utility in predicting ternary structure. Both stable and transient interactions with other proteins, nucleic acids, and small ligands and substrates are essential to the function of many proteins. Some complexes can be elucidated by co-crystallization, but others are resistant to current crystallographic techniques. Developing tools to predict sites of interaction and potential binding partners is important to move beyond primary and secondary structure classification. The function of a protein is strongly dependent on the surface properties of a protein, since the surface is what is presented to other proteins and ligands. Important surface features include charge, the availability of hydrogen bonding, and simple geometric shape. Shape complementarity plays an

important role in intramolecular interactions, and many binding sites are at pockets on the molecular surface. Pockets provide protection from solvent for enzymatic functions, and "lock and key" style shape specificity. Comparing pockets across related proteins can provide insight to shared functions, or sites of protein-protein interaction important to a particular protein family.

In order to map protein surfaces and compare pockets, a good algorithm is necessary. A number of pocket-finding algorithms have been developed in order to predict binding sites on proteins. The most recent and comprehensive is Abagyan's Pocketome (An et al. 2005), a large database of small, "druggable" pockets. Abagyan's method smooths over the Lennard-Jones potential, finding small areas in the protein that offer high numbers of potential Van der Waals interactions. It is extremely effective, and the Pocketome will prove to be a valuable source of information. Abagyan's methodology is tuned for locating small pockets, not the large, shallow grooves and pockets that are often involved in protein-protein interactions. Another publicly available algorithm is CASTp (Liang et al. 1998). CASTp finds pockets on the protein surface by using Veronoi tessellation. CASTp has a public server, and it finds larger pockets than Abagyan's method, but its utility is limited because it fails on approximately 1% of PDB structures (http://sts-fw.bioengr.uic.edu/castp/background.php).

Here, a program called FADE (Fast Atomic Density Evaluator, Mitchell et al. 2001) is used to map the protein surface and identify crevices and pockets. FADE is described in detail in Chapter 2. Briefly, it maps a protein onto a grid and computes

the rate of change of atomic density with respect to an expanding sphere at each grid point using convolutions to accelerate the computation. As shown in Chapter 2, the quantity calculated by FADE contains information about the protein surface and can map changes in surface curvature. FADE was chosen for its speed, the availability of full source code, its ability to run successfully on any structure that fits into memory, and potential extensibility. FADE was originally developed to map protein-protein interactions and it is able to find extended, shallow grooves on the protein surface as well as smaller ligand-binding pockets. Because of the grid-based approach, FADE does not fail on complex or irregularly shaped geometric structures. FADE is used in combination with my new piece of software, Crevasse, which segments the FADE output into distinct pockets that can be measured, compared, and stored in a database. Crevasse is also described in detail in Chapter 2. FADE and Crevasse are able to rapidly process any structure that can be mapped onto a grid that fits into the computer's memory.

Finding binding sites on a single protein is interesting and useful, but this thesis expands on a novel, more powerful approach. Pockets can be used to study similarities and differences in molecular geometry across a group of structures. If a set of related proteins is aligned and pockets are computed across the proteins, the pockets can be spatially clustered. The spatial clustering can find groups of evolutionarily conserved pockets on the set of proteins. A pocket with a known function on one protein that belongs to a cluster of conserved pockets on the rest of the proteins can be used to guide research into the tertiary structure of the family.

Similarly, a pocket present on only a subset of the proteins suggests that the pocket could be involved in a function that distinguishes those proteins from the rest of the family. Studying pure surface geometry is particularly powerful because it can find relationships between proteins that are missed by sequence and fold comparison.

The protein kinase family was chosen as a test system for the surface mapping methodology. Protein kinases are of interest because they are essential to cellular function and survival, and central to multiple signal transduction pathways. Mutations in kinases are part of oncogenesis. The importance of the kinase family in cancer was discovered in the 1970s when the oncogenic protein in Rous Sarcoma Virus, v-Src, was determined to be a constitutively active protein tyrosine kinase (Brickell, 1992, review). Kinases have since been extensively studied and two cancer chemotherapeutic agents, sunitinib and imitinib, have been developed to inhibit tyrosine kinases. (Review: Grimaldi et al., 2007). Comparative studies of the kinase family have been undertaken both within and across species using sequence similarity. The human kinome has at least 518 putative kinase genes, with 244 mapped to known disease loci (Manning et al., 2002). The mouse kinome is of similar size, with 540 putative genes, including orthologs for 510 of the 518 human protein kinases (Caenepee et al., 2004). The Venter Institute Global Ocean Survey contained thousands of kinase sequences from microbial genomes. Those sequences have been classified into 45,000 genes and 20 families (Natarajan et al., 2007). The mammalian protein kinases fall into only one of the identified families from the Venter data.

Structural comparisons of the kinase family have been undertaken, but they are largely limited to main chain structural alignments. A structurally conserved catalytic core was identified (Hanks and Hunter, 1995) when crystal structures first became available. The kinase catalytic core is comprised of an N-terminal lobe of mostly beta strands, and a C-terminal lobe of helices. There is a cleft between the lobes where the Mg-ATP and protein substrates bind. Near the hinge region where the lobes join, there is a helix, called the C-helix, that moves to position key catalytic residues when the kinase is activated. The kinase is flexible, with the cleft opening to bind and release substrate and closing during catalysis to protect the reaction from uncontrolled hydrolysis. The core and catalytic mechanism is shared in both tyrosine and serine/threonine mammalian kinases, despite their wide diversity of substrates. Because they are involved in signal transduction cascades, kinases are highly regulated, and most are switched on and off; constitutively active kinases are relatively rare and tend to show high substrate specificity. Receptor tyrosine kinases are activated by extracellular ligands, while non-receptor kinases are regulated in response to intracellular conditions or second messengers. Regulatory mechanisms include phosphorylation and dephosphorylation, ligand binding, and interaction with regulatory domains or other proteins. Upon activation, the catalytic core adopts a characteristic conformation, recently described by Kornev. (Kornev et al., 2006, Ten Eyck, Taylor, and Kornev, 2008). There are two columns of highly structurally conserved amino acids, described as spines, which run through the center of the kinase

core. They organize the positioning of the adenine ring of the ATP substrate and the active site residues.

Among mammalian kinases, cAMP dependent protein kinase (PKA) is the best described serine/threonine protein kinase. PKA is one of the major effector molecules in cAMP mediated signal transduction and it has multiple phosphorylation substrates. PKA is essential to cellular function; null mutants of PKA in yeast and drosophila are lethal (Lane and Kalderon, 1993, Toda et al. 1987). Multiple crystal structures of PKA have been solved, including apo-PKA, the binary complex with protein kinase inhibitor peptide (PKI), and partial holoenzyme complexes with regulatory subunits bound. PKA is unusual among kinases because it crystallizes in an active conformation. PKA has a unique structure, with two sequences beyond the catalytic core called the N-terminal and C-terminal tails. The positioning of the tails relative to the kinase core is essential to activation of the protein. Truncation of the N-terminal tail makes the protein more labile, and site-directed mutagenesis of the C-terminal tail inactivates the kinase (Johnson et al. 2001). PKA was chosen as a test molecule for this study because it has been thoroughly studied and multiple high-resolution crystal structures of the active kinase are available. It is possible to check the validity of the computation against the many known interactions with the PKA catalytic core, N- and C-terminal tails, and the regulatory subunits. Chapter 3 shows the FADE and Crevasse computation on PKA and shows how the software can be used to identify potential areas of interest on the surface of a single protein crystal structure.

In Chapter 4, the new methodology of pocket clustering is applied to a set of ten active kinase catalytic core crystal structures. Common features on the surface of the kinase catalytic cores are identified and described, including the active site cleft, shared mechanisms for C-helix positioning, a conserved feature at the top of the N-lobe, and potential novel sites of protein-protein interaction. The computation also identifies differences in the shape of the substrate binding pocket that discriminates between serine/threonine and tyrosine kinases.

An J, Totrov M, Abagyan R. (2005) Pocketome via comprehensive identification and classification of ligand binding envelopes. Mol Cell Proteomics. 2005 Jun;4(6):752-61. Epub 2005 Mar 9.

Brickell PM. (1992) The p60c-src family of protein-tyrosine kinases: structure, regulation, and function. *Crit Rev Oncog. 3(4)*:401-46.

Caenepeel S, Charydczak G, Sudarsanam S, Hunter T, Manning G. (2004) The mouse kinome: Discovery and comparative genomics of all mouse protein kinases. *PNAS 101 (32)*:11707-11712.

Finn RD, Mistry J, Schuster-Böckler G, Griffiths-Jones S, Hollich V, Lassmann T, Moxon S, Marshall M, Khanna A, Durbin R, Eddy SR, Sonnhammer ELL, Bateman A. Nucleic Acids Research (2006) Pfam: clans, web tools and services. *Database Issue 34*:D247-D51

Greene LH, Lewis TE, Addou S, Cuff S, Dallman T, Dibley M, Redfern O, Pearl F, Nambudiry R, Reid A, Sillitoe I, Yeats C, Thornton JM, Orengo CA. (2007) The CATH domain structure database: new protocols and classification levels give a more comprehensive resource for exploring evolution. Nucleic Acids Research: 35(1), D291-D297(1).

Grimaldi AM, Guida T, D'Attino R, Perrotta E, Otero M, Masala A, Cartenì G. (2007) Sunitinib: bridging present and future cancer treatment. *Ann Oncol.* 2007 Jun;18 Suppl 6:vi31-4.

Hanks SK, Hunter T. (1995) Protein kinases 6. The eukaryotic protein kinase superfamily: kinase (catalytic) domain structure and classification. *FASEB J. 9*(8):576-96.

Johnson DA, Akamine P, Radzio-Andzelm E, Madhusudan, Taylor SS. (2001) Dynamics of cAMP-Dependent Protein Kinase. *Chem. Rev. 2001(101)*:2243-2270.

Kornev AP, Haste NM, Taylor SS, Ten Eyck LF. (2006) Surface comparison of active and inactive protein kinases identifies a conserved activation mechanism. *Proc Natl Acad Sci USA.;103(47)*:17783-8.

Liang J, Edelsbrunner H, Fu P, Sudhakar PV, Subramaniam S. (1998) Analytical shape computing of macromolecules I: molecular area and volume through alpha shape. *Proteins. 33*, 1-17.

Liang J, Edelsbrunner H, Fu P, Sudhakar PV, Subramaniam S. (1998) Analytical shape computing of macromolecules II: identification and computation of inaccessible cavities inside proteins. *Proteins. 33*, 18-29.

Lane M, Kalderon D. (1993) Genetic investigation of cAMP dependent protein kinase function in Drosophila development. *Genes Dev 7:*1229–1243

Manning G, Whyte DB, Martinez R, Hunter T, Sudarsanam S (2002). The Protein Kinase Complement of the Human Genome. *Science 298*:1912-1934.

Mitchell JC, Kerr R, Ten Eyck LF. 2001. Rapid atomic density methods for molecular shape characterization. *J. Mol Graph Model. 19*(3-4):325-30, 388-90.

Murzin AG, Brenner SE, Hubbard T, Chothia C. (1995) SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J. Mol. Biol.* 247, 536-540.

Natarajan, K, Taylor, S, Zhai, Y, Venter, C, Manning, G (2007) Structure and function of the microbial kinome. *PLoS Biol. 5(3)*: e17.

Ten Eyck LF, Taylor SS, Kornev AP. (2008) Conserved spatial patterns across the protein kinase family. *Biochim Biophys Acta. 1784(1)*:238-43.

Toda T, Cameron S, Sass P, Zoller M, Wigler M. (1987)Three different genes in S. cerevisiae encode the catalytic subunits of the cAMP-dependent protein kinase. *Cell 50*:277–287

Chapter 2. FADE and Crevasse

Molecular shape complementarity plays an important role in protein-protein and protein-ligand interactions. I have developed a method using FADE, the Fast Atomic Density Evaluator (Mitchell, Kerr, and Ten Eyck, 2001) and my new piece of software, Crevasse, to rapidly map grooves and crevices on the protein surface. The method is scriptable and scalable, and can guide research into inter- and intra-molecular binding.

Section 2.01 FADE - Fast Atomic Density Evaluator

The protein is first mapped using FADE. FADE uses the gradients of radial counting functions to characterize the shape of the molecular surface, without actually computing the surface. FADE is based on counting the number of atoms within spheres of varying radii. FADE maps atomic centers from a protein crystal structure onto a grid and computes a score at each grid point. The FADE score at each grid point is the slope of log(N(r)) vs. log(r) where r is the radius of a sphere centered at the grid point and N(r) is the number of atoms within the sphere. Figure 2-1 illustrates how FADE works. The counting function accumulates atomic neighbors slowly at points next to a protrusion, shown as blue circles in the figure. In contrast, if the counting function starts in a pocket there is a rapid jump in the number of atomic neighbors accumulated as the radius expands (shown in red). Near flat surfaces (shown in green) the function accumulates atomic neighbors at an intermediate rate. For a perfectly uniform distribution of atoms, as found in a protein interior, the FADE

score should be 3; for points in a crevice, it will be larger than 3; and for points on a protrusion it will be less than 3. FADE computes these with a more efficient algorithm than direct counting, achieving speeds proportional to nlog(n) where n is the number of grid points. It takes only the atomic coordinates as input, giving the same effect as a curvature calculation without the necessity of computing a surface.

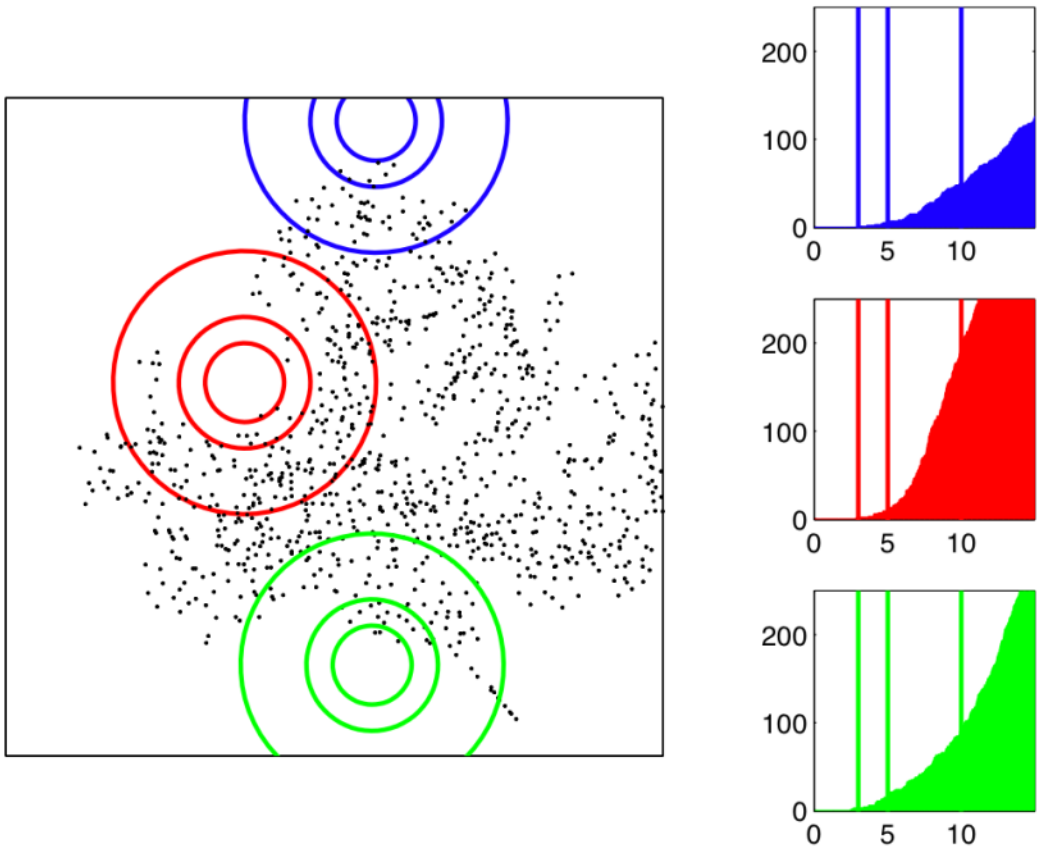

Figure 2-1. The relationship between the radial counting function and shape.
The radial counting function has a rate of increase which varies according to the local environment. Points near a protrusion accumulate atomic neighbors slowly (blue), while points inside a crevice will see a rapid increase in the number of atomic neighbors (red). Regions near a flat edge have behavior which lies between these two extremes (green). (Figure courtesy of Mitchell, Kerr & Ten Eyck)

By restricting the output to points where the FADE score is high and removing interior points, FADE can output a set of grid points that lines crevices and grooves on the molecular surface. In the original publication, FADE mapped crystallographic structures onto a grid with 1.0Å spacing. The input set included only those atoms that were resolved in the crystallographic structures. Hydrogen atoms, which are not usually visible in X-ray crystallographic structures, were not included in the computation. The software was able to map grooves on the protein surface, but it did not fill ligand pockets well, or flood deeper pockets differently from shallower ones. I was able to improve the FADE results dramatically by using finer grids and computationally adding hydrogen atoms. Because each atom is counted irrespective of type, adding hydrogen atoms improves the FADE computation by filling the interior of the protein more completely and representing the number of atoms at the molecular surface more accurately. The finer grid provides a better representation of atomic density and more detail in the output. With improvements in computer hardware, it has become practical to use 0.5 Å grids.

FADE needed some improvements before it could be used for the computations. Most importantly, I found and corrected an algebraic grid indexing error in the output routines. Mike Pique rewrote the memory allocation routines to run more efficiently and added error checking because the smaller grid spacing is more likely to cause out of memory conditions. Finally, I debugged and rewrote the output routines to run more efficiently and provide output that can be easily read by visualization programs and Crevasse. A change log is included in Appendix A.

Missing side chain atoms and residues are common in crystal structures of proteins because the experiment is sensitive to the average electron density. Mobile or disordered atoms have very low average electron density and are thus essentially invisible in these experiments. Such atoms must be treated cautiously in FADE computations because shape of the protein surface is partly dependent on side chain conformation. There are many ways to model missing atoms in a crystal structure, but no way to determine the crystallographic conformation of software-constructed side chains. Arbitrary placement of side chain atoms is not accurate enough for the computation. It is preferable to interpret FADE data very cautiously in regions of the protein where there are missing atoms.

To show the dependence of the FADE computation on grid size and the input atom set, FADE was tested using a high-resolution Protein Kinase A structure (PDB file 1RDQ). FADE was run both with and without hydrogen atoms added, and on half and one-ångström grids. Reduce 3.1 (Word et. al., 1999) on default settings was used to add hydrogen atoms to the crystal structure before the FADE computation. FADE was used on default settings, returning scores for points anywhere from zero to three grid points from an atomic center, summing over ten steps of sphere expansion, and with a maximum score cutoff of 6.0. Zero scores are outside the protein and thus were excluded from the analysis. The remaining FADE scores were between 1.0 and 5.0. Figure 2-2 shows a histogram of FADE scores after the four different computations and shows summary statistics on the returned scores. The distribution of the FADE scores becomes sharper with the addition of hydrogen atoms, indicating a more

uniformly filled grid in the interior of the protein. The mean FADE score also changes from 2.776 on a one ångström grid with no hydrogen atoms to 2.931 on a half ångström grid with hydrogen atoms added. A perfectly filled grid has a theoretical radial counting gradient of 3.0, so the higher mean score of the computation on a half-ångström grid with hydrogen atoms added to the protein indicates an improvement in the representation of the well-packed protein interior.

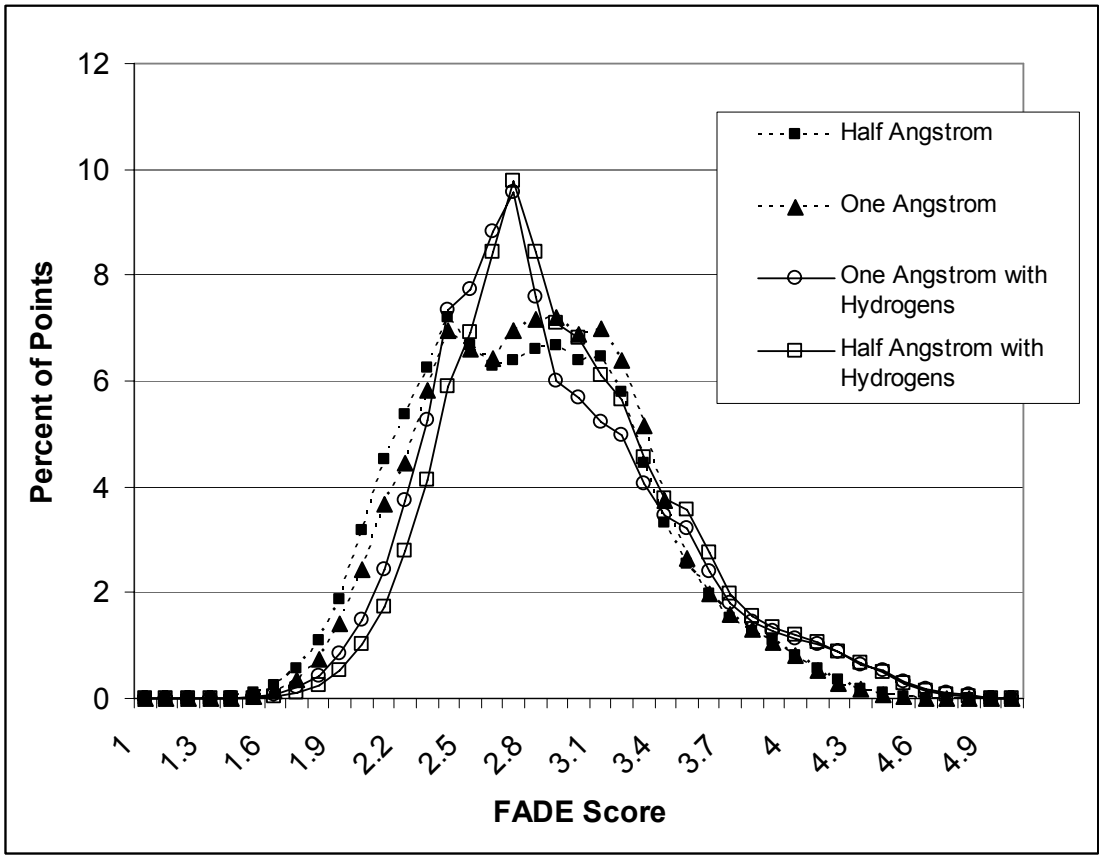

Figure 2-2. Distribution of FADE Scores by Grid Spacing
Distribution of FADE scores under different grid spacing and hydrogen atom conditions. Open markers show the distribution of scores with hydrogen atoms are added to the protein. Closed markers show the distribution of scores without added hydrogen atoms.

Table 2-1. Summary Statistics on FADE Scores

Grid Spacing (Å)	Hydrogens Added	Mean	Min	Max
1.0	No	2.776	1.143	4.713
0.5	No	2.819	1.247	4.804
1.0	Yes	2.877	1.301	4.977
0.5	Yes	2.931	1.205	4.996

Section 2.02 Using FADE

The FADE software has built-in filters to limit the number of points returned. FADE accepts five parameters, a minimum and maximum distance from atomic centers, minimum and maximum FADE scores, and the number of times to expand the radius and count atoms. After the counting function is complete, FADE limits the number of points returned by filtering them to meet the user-provided criteria. The distribution of FADE scores changes with both the counting radius and the distance from atomic centers so it is important to determine settings for the software that provide useful output. Setting the distance filter in FADE to return only points 1.5 Å (3 grid points) from grid points with an atomic center returns a tight shell around the molecule. Under grid mapping error, it is a reasonable approximation of the Van der Waals surface of the protein, 1.7 Å from carbon atoms.

Under the conditions of 10 counting steps and 3 grid points from atomic centers suggested in Mitchell, Kerr, and Ten Eyck, a minimum FADE score cutoff of 4.6 was empirically chosen to return points that line obvious pockets. Higher FADE score cutoffs produce overly sparse output, while lower thresholds can return points on flat surfaces. The output from FADE is a set of grid points and scores. Figure 2-3

shows a graphic representation of the output from FADE. The structure is the catalytic core of PKA (PDB ID 2CPK), residues 40-300, with PKI and all solvent atoms removed from the structure. The FADE output is represented with a small sphere at each grid point that had a FADE score above 4.6 and was 1.5 Å from a gridpoint where an atomic center was mapped. The redder points indicate higher FADE scores, and the blue ones indicate lower scores. The figure shows how well raw FADE output maps areas on the molecular surface where the exponential density changes rapidly. Deep sites, including the ATP binding site (partly obscured in the figure) and the P+1 substrate binding are well-lined with points that pass the distance and exponential density filters and the centers of the sites have the highest FADE scores.

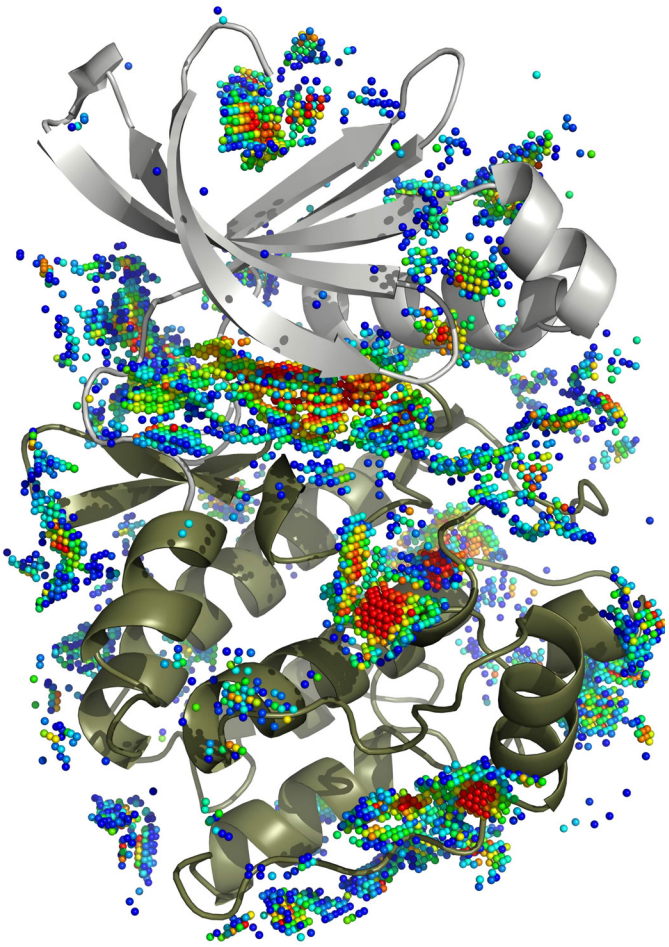

Figure 2-3. Sample FADE Output
Raw FADE output on Protein Kinase A residues 41-300. FADE output is colored by score from blue to red, with blue representing lower scores and red representing higher scores.

Section 2.03 Crevasse

The raw FADE output is difficult to characterize and score because of the uninteresting isolated points and small clusters of points visible in Figure 2-3. FADE output is better for computations after the signal-to-noise ratio is improved by discarding outlying points. The remaining large groups of points can be segmented

into discrete binding sites or crevices. Distinct crevices are useful because they can be computationally characterized, compared, and stored in a database. The size, shape, and center of the pockets can be computed and used to identify individual sites of potential protein-protein interaction or ligand binding. A piece of software, Crevasse, was written in C++ to segment FADE output into individual crevices. The software performs the following steps:

1. Reads FADE output into a grid. Crevasse assumes that FADE was run with reasonable scoring and distance filters and stores zeros for points with no returned FADE data and ones for points with data.

2. Runs a recursive depth-first search (Cormen et al., 2001) seeded at each nonzero grid point to identify sets of connected points. Points that are identified as connected but do not meet the connectivity threshold are added to the set, but not queued for exploration. The search algorithm is greedy; once a point belongs to a set, it can not be added to a different set.

3. Counts the number of points in each set identified in the search and filters the sets by discarding sets with fewer than a threshold number of points. If no sets meet the threshold size, the software terminates with a useful message.

4. Calculates the object-oriented bounding box for each retained set of points using the eigenvectors of the covariance matrix of the points.

5. Filters the groups of points a second time based on a user-specified minimum length for the longest axis of the bounding box. Again, if no sets meet the threshold the software terminates.

6. Outputs sets of points and summary information for each identified set.

Crevasse runs at the command line and was designed to be called using shell scripts. It can be compiled and run on any platform with the GCC compiler, making it compatible with FADE. Crevasse runs very quickly, with file I/O taking most of the run time. There is one step where the C++ standard template library sort is called so increasing the stack size on the machine running crevasse can be necessary for larger grids. Crevasse uses the public domain Template Numerical Toolkit library from the National Institute of Standards and Technology (http://math.nist.gov/tnt/index.html) for the eigenvector calculations. Full source code for Crevasse is included in Appendix B.

Crevasse is flexible and user-configurable. There are command line options to set the minimum number of points to define a crevice, set the minimum length required to define a crevice, change the number of points required to add a point to the search queue (the connectivity threshold), and choose the way adjacent points are queued for exploration. As mentioned above, the minimum length setting applies a lower threshold to the length of the longest axis of the bounding box. Changes to the minimum number of points in a crevasse and minimum length of the long axis change the sensitivity of the software and should be optimized depending on the problem considered. Searches for potential small ligand binding sites would require a shorter length setting than searches for large protein surface features. The connectivity threshold controls whether adjacent patches of points are grouped into separate

segments. Higher settings return more, smaller segments while lower settings can connect long, thin grooves on the protein surface that may have lower connectivity. Finally, the way points are queued can be changed. The default setting includes diagonals as adjacent points, so up to twenty-six points are queued for exploration during the depth-first search. There is an optional setting to queue and search only the six grid points at x±1, y±1 and z±1. When FADE output is restricted to a thin layer of points 1.5 Å from grid points with atomic centers, searching twenty-six points provides better results.

Summary output from Crevasse is shown in Table 2-2. The axis lengths are the length of the major axis through the points and the two other orthogonal axes that define the bounding box. Crevasse can also output the coordinates for the center of the bounding box, or for the eight corners. Crevasse was run on the FADE output on the PKA catalytic core previously shown in Figure 2-3. The software identified twenty-one distinct sets of points on the protein surface. The site numbering is arbitrary; in this computation the ATP and PKI binding site is identified as site 8 and is the largest site identified by the computation. Site 9, the second largest site, is involved in regulatory subunit binding.

Table 2-2. Crevasses found on PKA Core

Output from the Crevasse computation, showing the point count, box volume, and length of the axes through the bounding box for each site.

Arbitrary site ID	Number of points in site	Volume of Bounding Box in Å^3	Length of Axis 1 in Å	Length of Axis 2 in Å	Length of Axis 3 in Å
1	497	798.7	4.4	12.1	14.9
2	81	43.2	2.2	3.3	6.0
3	259	982.4	5.4	9.3	19.4
4	124	185.1	3.8	6.7	7.3

5	107	195.7	3.5	4.8	11.4
6	149	207.8	3.6	4.5	12.8
7	130	184.5	4.1	5.7	7.9
8	1351	3359.1	8.0	13.4	31.3
9	503	989.3	7.4	9.6	14.0
10	245	312.5	5.1	7.4	8.3
11	113	130.8	2.3	8.1	7.0
12	314	348.6	5.4	5.5	11.8
13	181	256.7	3.4	7.2	10.4
14	123	133.2	2.9	6.4	7.1
15	89	29.5	2.6	3.2	3.6
16	115	97.4	2.3	5.3	8.2
17	253	429.0	5.5	6.6	11.8
18	342	340.5	5.1	6.4	10.5
19	300	477.9	5.1	6.4	14.7
20	321	505.2	4.4	7.8	14.7
21	84	88.1	2.4	5.0	7.3

Figure 2-4 is a graphic representation of the output. To generate Figure 2-4, Crevasse wrote the sets of points as hydrogens in a pseudo-PDB file format to load into a molecular viewer. Each distinct set of points is given a residue number. The sets of points were arbitrarily colored by residue in PyMol to show the distinct, computationally identified pockets. The segmentation preserves and separates major sites like the ATP binding site and the P+1 peptide binding site. The single points around the N-lobe and adjacent to the C-helix have been filtered out and the signal-to-noise improved markedly.

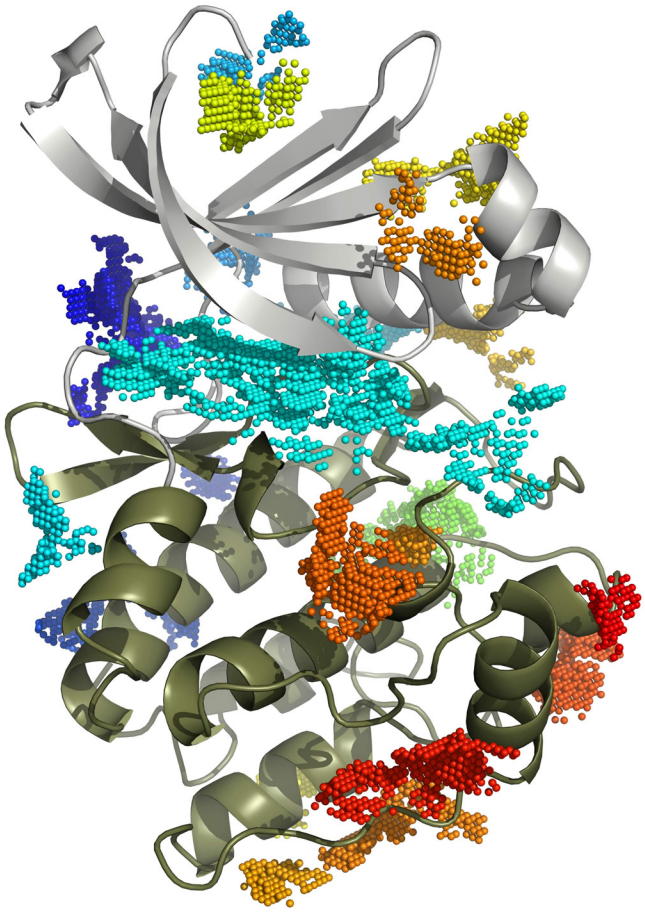

Figure 2-4. Sample Crevasse Output
Sample Crevasse output on residues 40-300 of Protein Kinase A. Patches of colored spheres are the grid points that define each separate crevice on the protein surface.

The combination of FADE and Crevasse takes under a minute to map the surface of PKA on a 2.4 GHz Pentium III. Most of the computation time is spent on the counting function, which scales at nlog(n) where in is the number of grid points. With a series of simple shell scripts, a family of proteins can be mapped quickly and efficiently for analysis of surface features.

The description of Crevasse and FADE in Chapter 2 and the PKA results in Chapter 3 will be written up for publication. The dissertation author will be the primary investigator and author. Dr. Choel Kim provided the new high-resolution crystal structure of PKA used in Chapter 3 and Mike Pique helped with the optimization of FADE detailed in Chapter 2. Both will be co-authored. Dr. Susan Taylor and Dr. Lynn Ten Eyck will also be co-authors on the paper.

Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Section 22.3: Depth-first search, pp.540–549.

Mitchell JC, Kerr R, Ten Eyck LF. 2001. Rapid atomic density methods for molecular shape characterization. *J. Mol Graph Model. 19*(3-4):325-30, 388-90.

Word JM, Lovell SC, Richardson JS, Richardson DC. 1999. Asparagine and glutamine: using hydrogen atom contacts in the choice of sidechain amide orientation. *J. Mol. Biol. 285*, 1735-1747.

Chapter 3. Mapping the surface of Protein Kinase A

Section 3.01 Introduction

cAMP dependent protein kinase (PKA) was chosen as a model molecule to test FADE and Crevasse. Protein kinases regulate other proteins by covalently attaching a phosphate group from ATP (adenosine triphosphate) to the hydroxyl on a serine, threonine, or tyrosine residue of a substrate protein. PKA phosphorylates multiple targets in response to intracellular increases in cAMP (3'-5'-cyclic adenosine monophosphate), an important second messenger molecule. As mentioned in Chapter 1, PKA has been extensively studied, making it a good test molecule for FADE and Crevasse. There is a comprehensive review published by Johnson et al. in 2001 that describes PKA in great detail. Briefly, PKA is composed of two subunits, a catalytic subunit and a regulatory subunit. The catalytic subunit has a structurally conserved catalytic core that is present in all mammalian kinases (Hanks and Hunter, 1995). The cAMP binding domains are on the regulatory subunit. Upon cAMP binding, the catalytic subunit dissociates from the regulatory subunit and phosphorylates target proteins. The catalytic subunit can be divided into an N-terminal tail of 40 amino acids, a conserved catalytic core, and a C-terminal tail of 50 residues. The catalytic core consists of two lobes. The N-terminal lobe or N-lobe is made primarily of five antiparallel β-strands and a prominent helix, called αC or the C-helix. The C-terminal lobe, or C-lobe is larger and formed primarily from α-helices. There is a highly conserved loop between strands β1 and β2 called the glycine loop. It has a glycine rich sequence (GXGXφG) where φ is tyrosine or phenylalanine that forms part of the

ATP binding site (Review: Huse and Kuryian, 2002). The substrate ATP binds in a pocket formed by the two lobes and the glycine loop, and the substrate peptide binds in a groove formed between them. The substrate phosphorylation site, called the P site, is positioned very close to the γ-phosphate of ATP.

PKA is activated to its catalytically competent state by phosphorylation and conformational changes. Figure 3-1 shows a diagram of PKA activation. The inactive kinase is on the right. T197 is not phosphorylated, the C-helix is moved away from the kinase. E91 is moved out of the active site cleft. The left-hand diagram shows the active kinase. When the C-helix is in the active position, a conserved set of amino acids lines up between the F-helix and the active site cleft, forming a regulatory spine that stabilizes the kinase. (Kornev et al., 2006). T197 is phosphorylated and the C-helix is positioned close enough to the enzyme to move E91 into the active site cleft to form a salt bridge with K72 and the conserved D184 from the DFG motif is moved into the active site cleft. When the enzyme is active, it is still flexible. There is subtle interdomain movement between the N- and C-lobes. It is unclear how far the enzyme has to open to release ADP and bind ATP between each round of catalysis but some movement is likely necessary (Johnson et al. 2001).

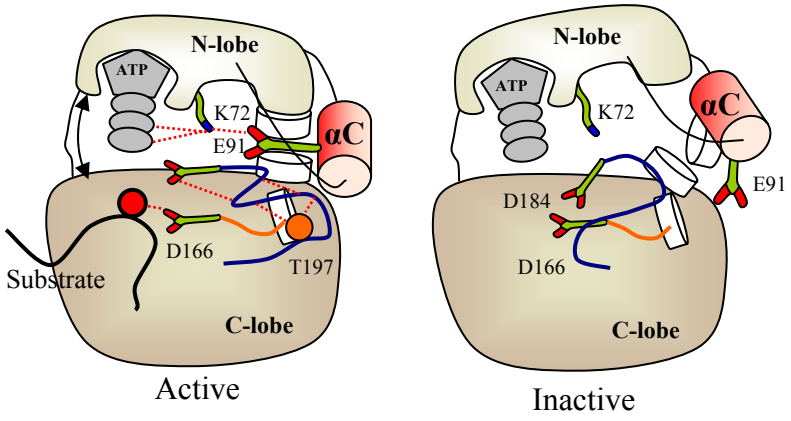

Figure 3-1. Protein Kinase A Subdomains and Conformations
The two main conformations of PKA are illustrated. Phosphorylation of T197 and positioning of the C-helix to move E91 into the active site cleft activates the kinase. In the inactive form, T197 is not phosphorylated and the C-helix is moved away from the protein, removing E91 from the active site cleft. (Figure courtesy of A. Kornev.)

Protein Kinase A has regulatory subunits that associate and dissociate from the kinase. There are four regulatory subunit genes, called RIα, RIβ, RIIα, and RIIβ and the combination of catalytic and regulatory subunit is called the PKA holoenzyme. The PKA holoenzyme is not catalytically active. The kinase is activated when cAMP binding to the regulatory subunit causes a conformational change in the regulatory subunit that dissociates the holoenzyme, freeing the catalytic subunit (Review: Cheng et al., 2008). When cAMP levels fall, the regulatory subunit returns to a conformation that binds PKA and the holoenzyme reforms. The RI and RII subunits are distinguished by the sequence that binds to the active site of the catalytic subunit. RI regulatory subunits have a pseudosubstrate sequence, while RII regulatory subunits have a substrate sequence which is phosphorylated upon binding. Different regulatory subunit isoforms are expressed in different tissues. Once the PKA holoenzyme has formed, it can be sequestered by interaction with A-kinase anchoring proteins

(AKAPs). Another protein that will be mentioned is Protein Kinase Inhibitor (PKI). PKI binds to the PKA catalytic subunit and has a nuclear export sequence, moving it out of the nucleus. A 20-residue inhibitor peptide from PKI has been a useful tool for crystallization of PKA and is seen in the structure by Knighton et al. (1991) used in this analysis.

Section 3.02 Methods

FADE and Crevasse were used as described in Chapter 2 to map the surface of Protein Kinase A. Two different PKA structures were used. One is crystallized with PKI but no ATP, (Knighton et al. 1991, PDB ID 2CPK), and the other is a new crystal of PKA with ATP and PKI (coordinates kindly provided by Dr. Choel Kim). The structures were chosen because they do not contain the regulatory subunit, they have all of the heavy atoms resolved, the conformation is closed, and they have no mutations or alterations to the protein. The N- and C-terminal tails were removed from the protein, and FADE and Crevasse were run over the catalytic core, residues 41-297. Unless mentioned otherwise, only peptide was used for the computation and water, ATP, PKI, and any non-biological molecules that co-crystallized with the kinase were removed. Hydrogen atoms were added to the structures with Reduce 3.1 (Word et. al., 1999) on default settings. FADE was set to return grid points with scores above 4.6, 1.5Å from grid points with atomic centers. Crevasse was set to require 80 points and 5Å length on the longest axis to output a pocket. The resulting data allow construction of a surface map that highlights important structural features of the kinase.

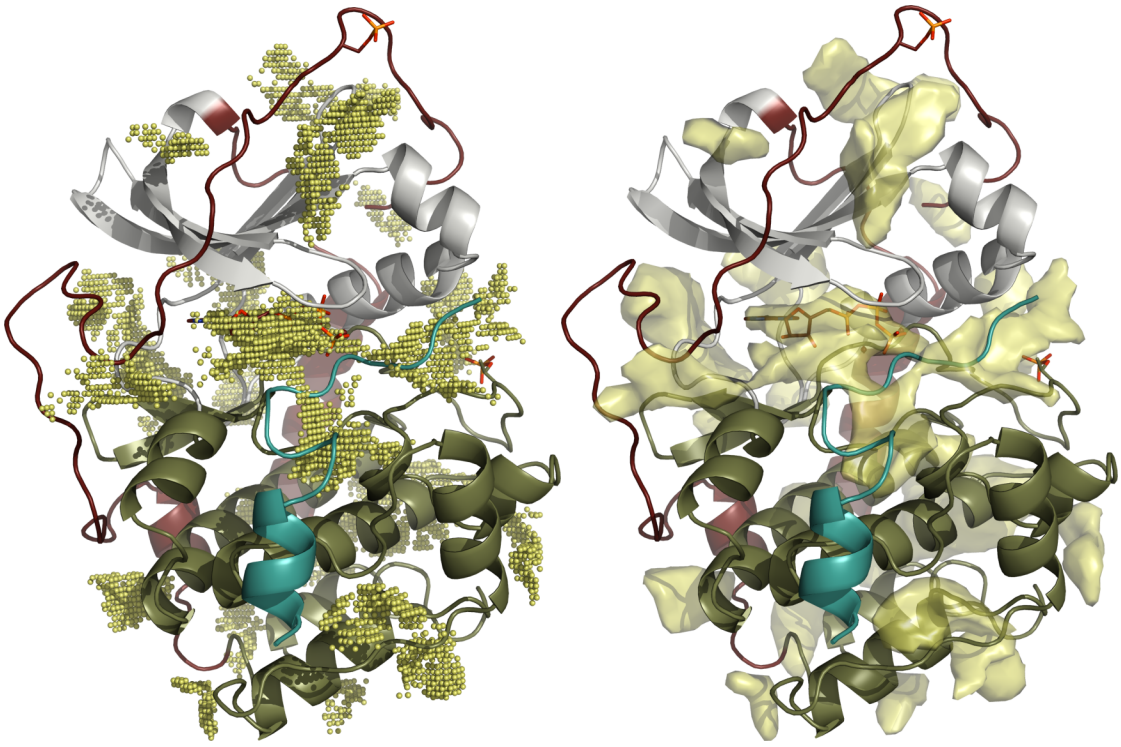

Figure 3-2. Sample Crevasse Computation on PKA Catalytic Subunut
The PKA N-lobe is grey, C-lobe is tan, N and C-tails are red, and PKI is pictured in green. The crevasse computation is shown in gold. Raw points from the computation are pictured on the left. On the right, the points have been represented as a surface in PyMol.

Section 3.03 ATP and Substrate Site

The classical view of protein kinases separates the substrate binding pocket into two parts, the ATP binding site, and the substrate binding site. FADE shows that there is a continuous surface that comprises the kinase active site rather than two separate sites for ATP and substrate. The N and C lobes of the kinase create a space for substrate binding and catalysis, the active site cleft. Figure 3-3 shows the results of the Crevasse computation on the active site cleft with PKI and different parts of the ATP ligand removed. As atoms are removed from the structure, the computed pocket expands to fill the resulting empty space and shows the different parts of the ATP

docking site. Part A shows the computation with PKI and all ligand atoms removed from the coordinates. Part B shows the results with the adenosine ring added to the coordinates. The area that is no longer filled is the adenosine docking site. Similarly, part C shows the results with both adenosine and ribose added to the coordinates and identifies the ribose docking site. Part D has the full ATP molecule added and shows the phosphoryl transfer site. Finally, in part E PKI is added. The active site cleft is completely filled, and a new indentation on the surface appears over PKI.

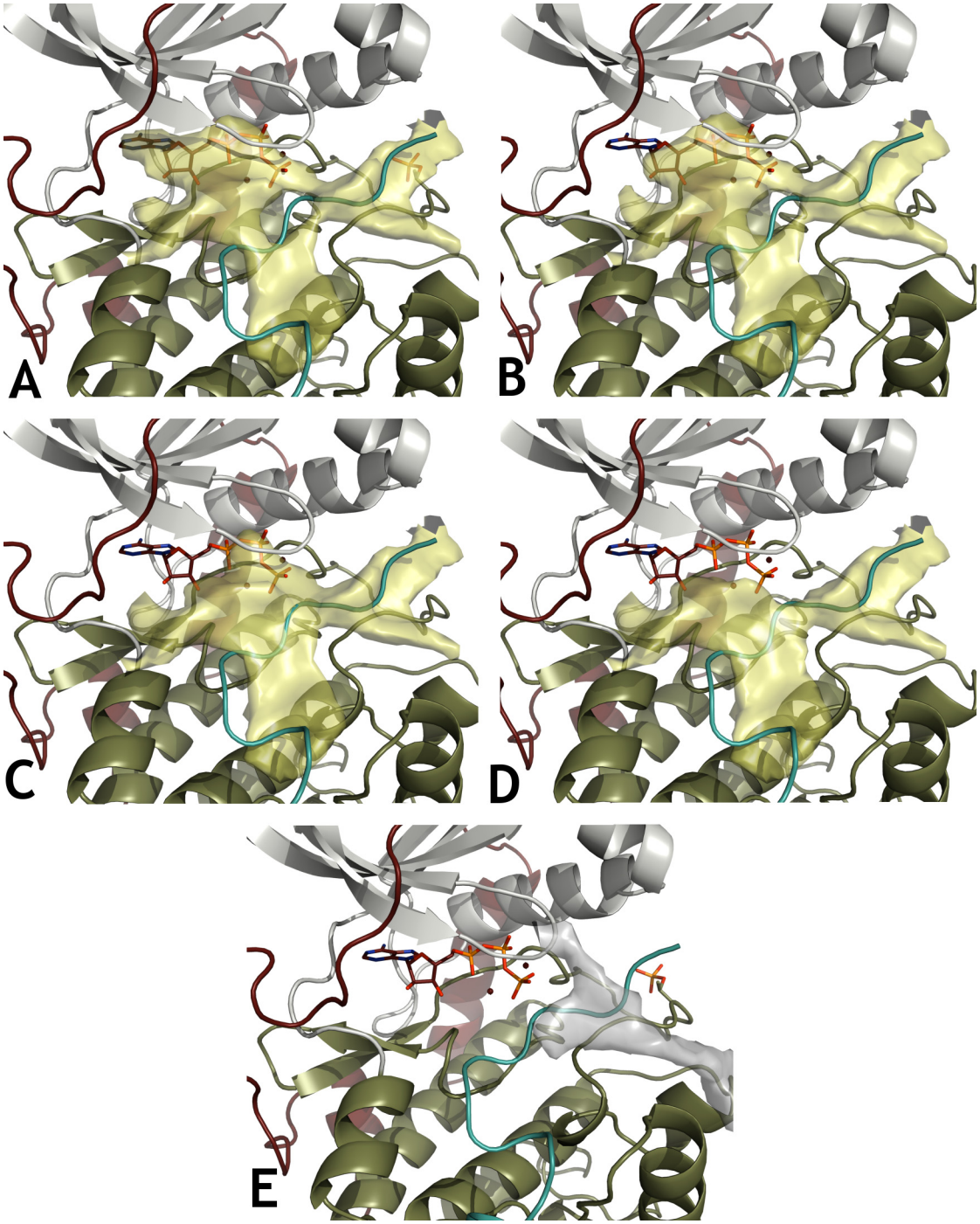

Figure 3-3. Dissection of the ATP Docking Site in PKA.
PKA ribbons are shown with the N-lobe in grey, the C-lobe in tan. The Crevasse computation is shown as surfaces. **A**. Ligand removed completely. **B**. Adenine ring included. **C**. Adenine and ribose included. **D**. Full ATP molecule included. **E**. Computation with PKI. The cleft is filled and a new pocket (grey) appears over PKI.

Further examination of the interaction of PKI with PKA shows some of the specificity determinants. Figure 3-4 shows the same Crevasse computation in Figure 3-3, but with the amino acids in the C-terminal region of PKI shown. The area identified as a pocket has been rendered as a gold surface in Pymol. In the computation on this particular structure, the points overlying the surface where the C-terminal region of PKI binds are contiguous with the active site cleft, but on some of the other PKA crystal structures, they segment into a separate binding site. The P site pseudosubtrate alanine, P+1 isoleucine, and three arginine residues at P-2, P-3, and P-6 fit into the computationally identified pocket. The two arginine residues are the main specificity determinant for PKA and the algorithm highlights the close fit onto the PKA surface. The pocket for the P-2 arginine is lined by E170 from the catalytic loop, Y204 from the P+1-loop, and E230 from the C terminus of the F-helix. E203 from the P+1-loop is at the bottom of the pocket for the P-6 arginine. The peptide binding site can be split into three areas, an N-terminal docking site for the P+1 residue, the phosphoryl transfer site, and a C-terminal docking site occupied by the arginine residues.

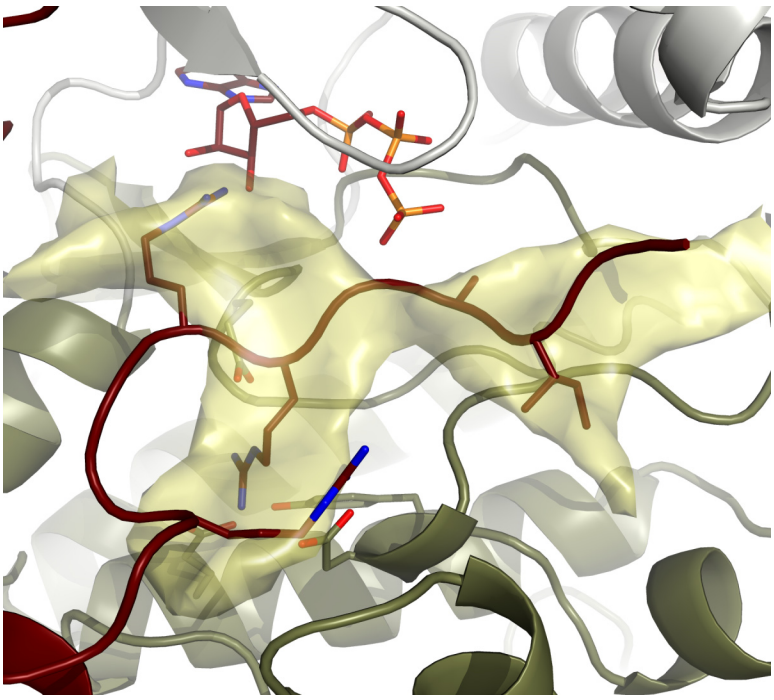

Figure 3-4. Interaction between PKA and PKI
The Crevasse computation is shown as a gold surface, computed in Pymol. PKI was removed for the computation. The P alanine, P+1 isoleucine, and the P-2, P-3 and P-6 arginines fit tightly against the PKA surface into computationally identified pockets. Residues E170, E203, Y204, and E230 from PKA are also shown.

The algorithm was also run on the PKA catalytic subunit in complex with adenosine (Naryana et. al, 1BKX). The crystal structure of 1BKX is in a closed, active conformation that is very similar to that of the crystal in Figure 3-4. The main difference between the two structures is the absence of pseudosubstrate. Nonetheless, the peptide binding site has very similar geometry. The binding sites for the P+1 isoleucine, and the P-2, P-3 and P-6 arginines are formed in the absence of PKI. This shows that the shape of the peptide binding site in PKA is not induced but rather that shape complementarity may play a role in PKA substrate recognition. Figure 3-6 shows an overview of the organization of the PKA active site cleft.

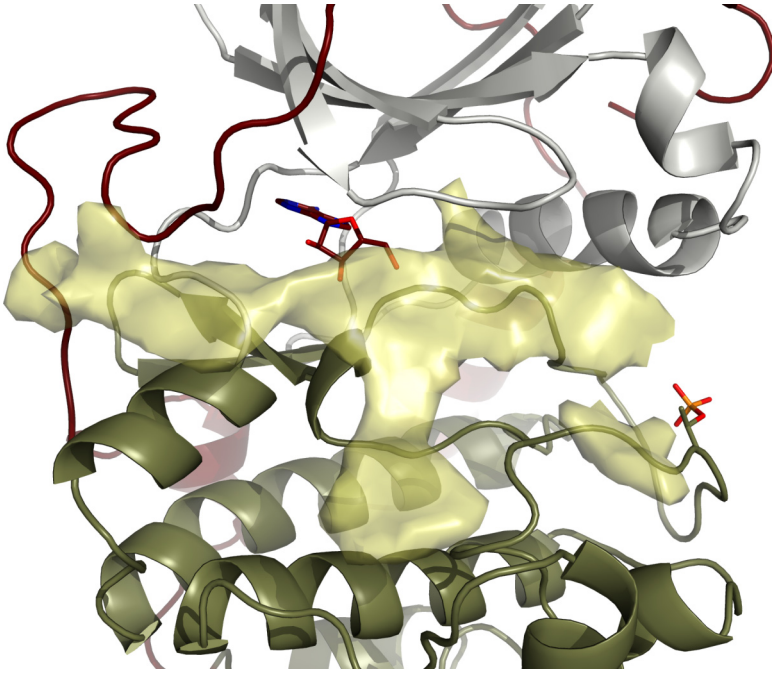

Figure 3-5. Substrate Binding Site in the Absence of PKI.
The gold surface is the computationally identified peptide docking site, showing that the large groove for the peptide backbone and the P-2 and P-3 arginines forms in the absence of substrate.

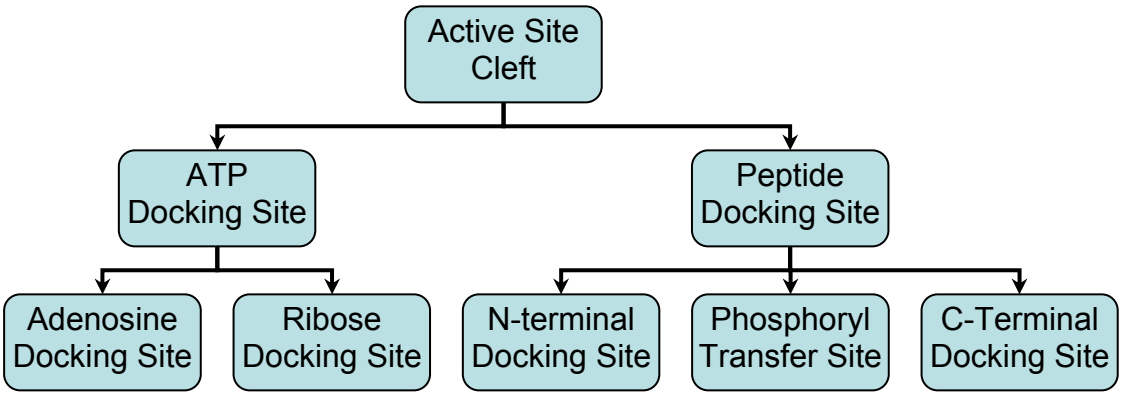

Figure 3-6. Roadmap to the Active Site
The hierarchy of docking sites in the PKA Active Site Cleft.

Section 3.04 C-Helix Positioning

PKA has a dynamic structure and has been shown to reorganize upon activation. As mentioned in the introduction, positioning of the C-helix is important in activating the kinase. Figure 3-7 shows the organization the kinase C-helix. The C-helix can be organized into four regions where inter and intra-molecular interactions take place, the N-lobe surface, the C-lobe surface, the N-terminal end, and the C-terminal end.

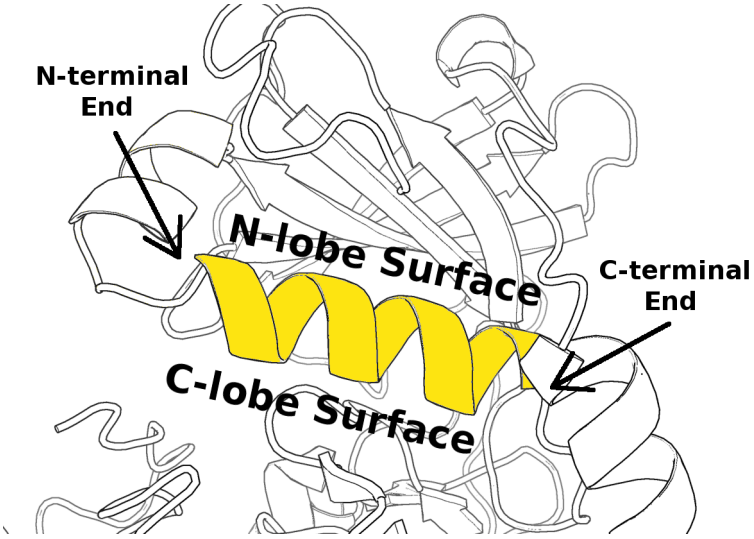

Figure 3-7. C-Helix Organization.
Organization of the kinase C-helix, shown on PKA ribbons. The C-helix is shown in gold. Across the kinase family, C-helix positioning is determined by inter- and intramolecular interactions with the four labeled regions, the N-lobe surface, C-lobe surface, N-terminal, and C-terminal.

On the PKA catalytic core the algorithm identifies two distinct pockets adjacent to the C-helix, one at the middle of the N-lobe surface and a second near the C-terminal end of the C-lobe surface. Figure 3-8 shows the computationally identified pockets on the kinase core and the how they are occupied in the complete structure.

The pocket on the N-lobe surface of the C-helix is filled by the FxxF motif formed by residues 347-350 from the C-terminal tail. Residues F26 and W30 from the N-terminal tail fit into the pocket on the C-lobe surface, and stabilize the C-helix from the C-lobe surface.

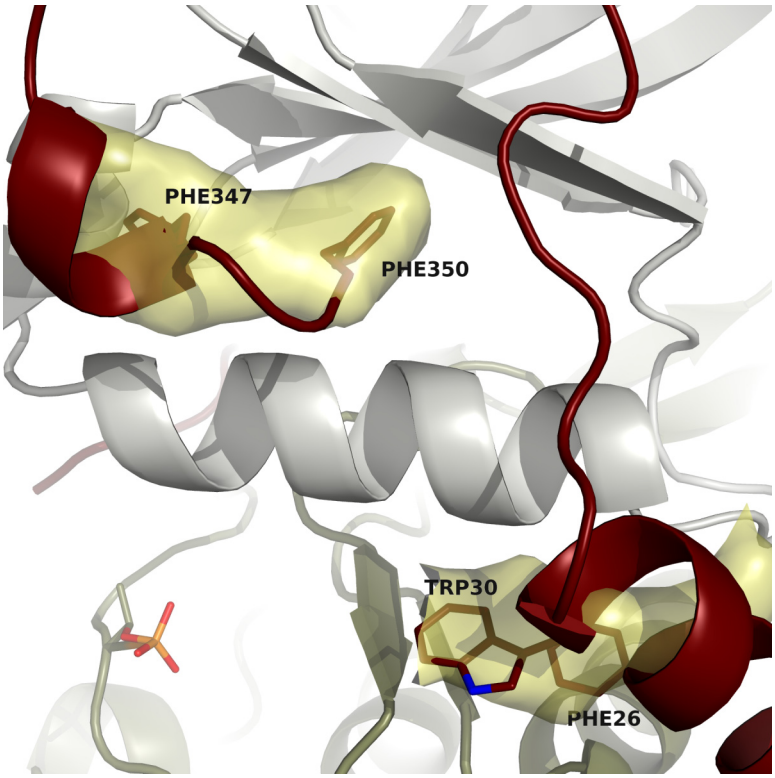

Figure 3-8. Protein Kinase A C-helix with Computation.
Pockets indentified by the computation are shown as gold surfaces, rendered in PyMol. The N-lobe is colored grey and the C and N-terminal tails are dark red. F347 and F350 fill the C-lobe surface site and F26 and W30 fill the the N-lobe surface site.

Section 3.05 β-Sheet Cap

The PKA catalytic core was defined as beginning at residue D41 thorough a combination of sequence and structure homology studies with other kinases. When the Crevasse computation is run starting at residue D41, there is a pocket that appears at the top of the N-lobe. It is lined by F41, M71, F110, and Y117. L40 folds over into

the center of the hydrophobic pocket. I335 from the C-terminal tail also folds into the same region, although it is not inside the computationally identified pocket. The pocket and residues lining it are pictured in Figure 3-9

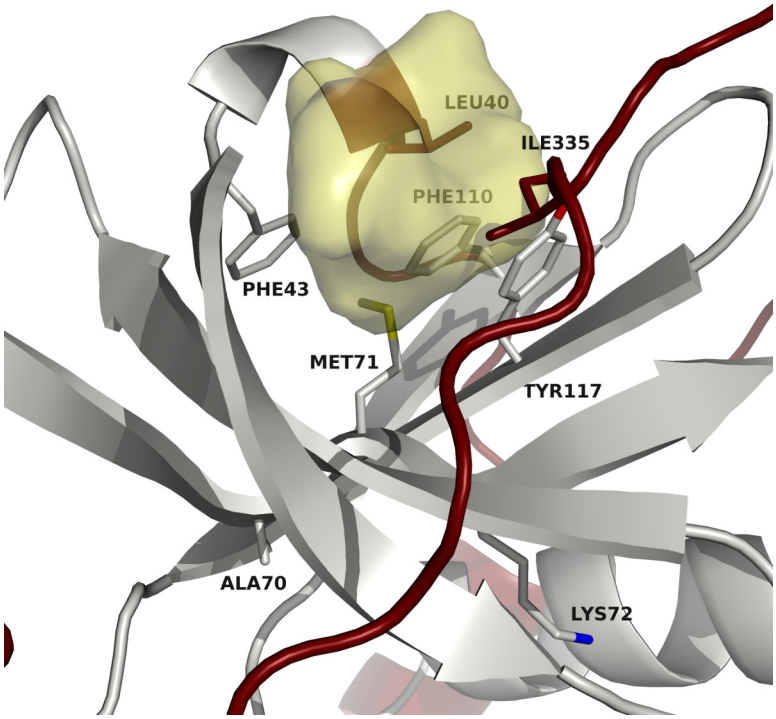

Figure 3-9. β-Sheet Cap in PKA
The N-terminal and C-terminal tails are shown in deep red.

The function of the β-Sheet cap is unknown. Methionine 71 at the bottom of the cap is adjacent to two important residues, K72 and A70. K72 is an essential catalytic residue, so its positioning is critical for proper function of the kinase. A70 is also conserved, and it makes a hydrophobic contact with the adenine ring. Proper positioning of A70 and K72 points M71 into a hydrophobic region on top of the β-3 strand. This may serve to further stabilize the active form of the kinase with hydrophobic interactions.

Section 3.06 C-Lobe

The C-Lobe on PKA is involved in protein-protein interactions with the regulatory subunits. There is a pocket adjacent to the APE α-F loop, PKA residues 206-308. E208 in the conserved APE motif hydrogen bonds to R280 and helps to stabilize the activation loop (Natarajan et al. 2007). The pocket is a site of protein-protein interaction with the RII-β regulatory subunit. The end of the RII-β c-terminal helix falls over the pocket, blocking it. There is a hydrogen bond between RII-β K263 and the backbone amide of K213 that forms within the computationally identified pocket.

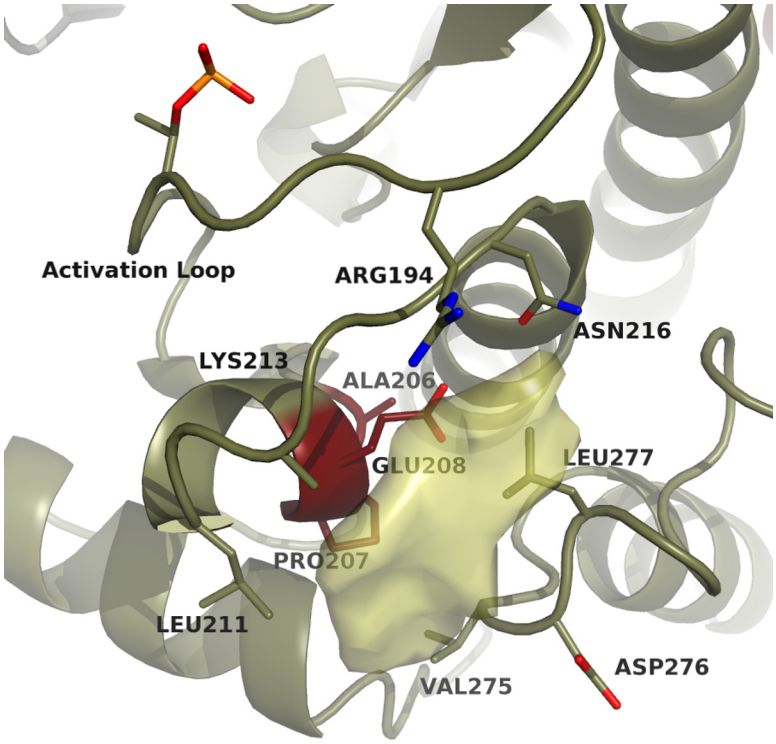

Figure 3-10. Conserved APE Motif and Pocket.
The APE motif resides are shown in dark red. Other nearby residues are also shown. K213 was not fully resolved in this crystal; the shape of the pocket changes slightly in structures with K213 but is still present.

Section 3.07 Myristic Acid Covalent Modification

Purified mammalian PKA has been shown to have a myristic acid covalent modification (Zheng et al. 1993) on the n-terminal tail. Zheng identified an acyl binding pocket adjacent to the n-terminal tail, and when nonmyristylated recombinant PKA crystals are grown in the presence of octanoyl-*N*-methylglucamide (MEGA-8) detergent, a detergent molecule occupies the site (Naryana et al. 1993). When recombinant PKA is crystallized in the absence of detergent, the acyl pocket is still present, and is occupied by solvent.

Crevasse successfully identifies the empty acyl myristylation site on the recombinant enzyme as a pocket. When the algorithm is run, co-crystallized molecules such as detergents, water molecules, and glycerol are removed from the protein. Examination of the PKA structure with MEGA-8 shows a computationally identified pocket in the same spot that is neatly filled by the detergent molecule. The pocket is completely absent in the myristylated mammalian protein, where the computation was run with the myristic acid included. The myristic acid packs well enough to smooth the surface of the protein and fill the acyl pocket. This demonstrates the ability of Crevasse to identify potential small molecule binding sites, and shows the stability of the acyl site in nonmyristylated PKA. The empty pocket under the N-terminal tail present in all three structures is also of interest. It is not known whether the N-terminal tail always stays closely packed with the kinase core, or whether it is mobile. If the N-terminal tail was to move away from the kinase, the

pocket visible in all three structures would be exposed for a protein-protein interaction.

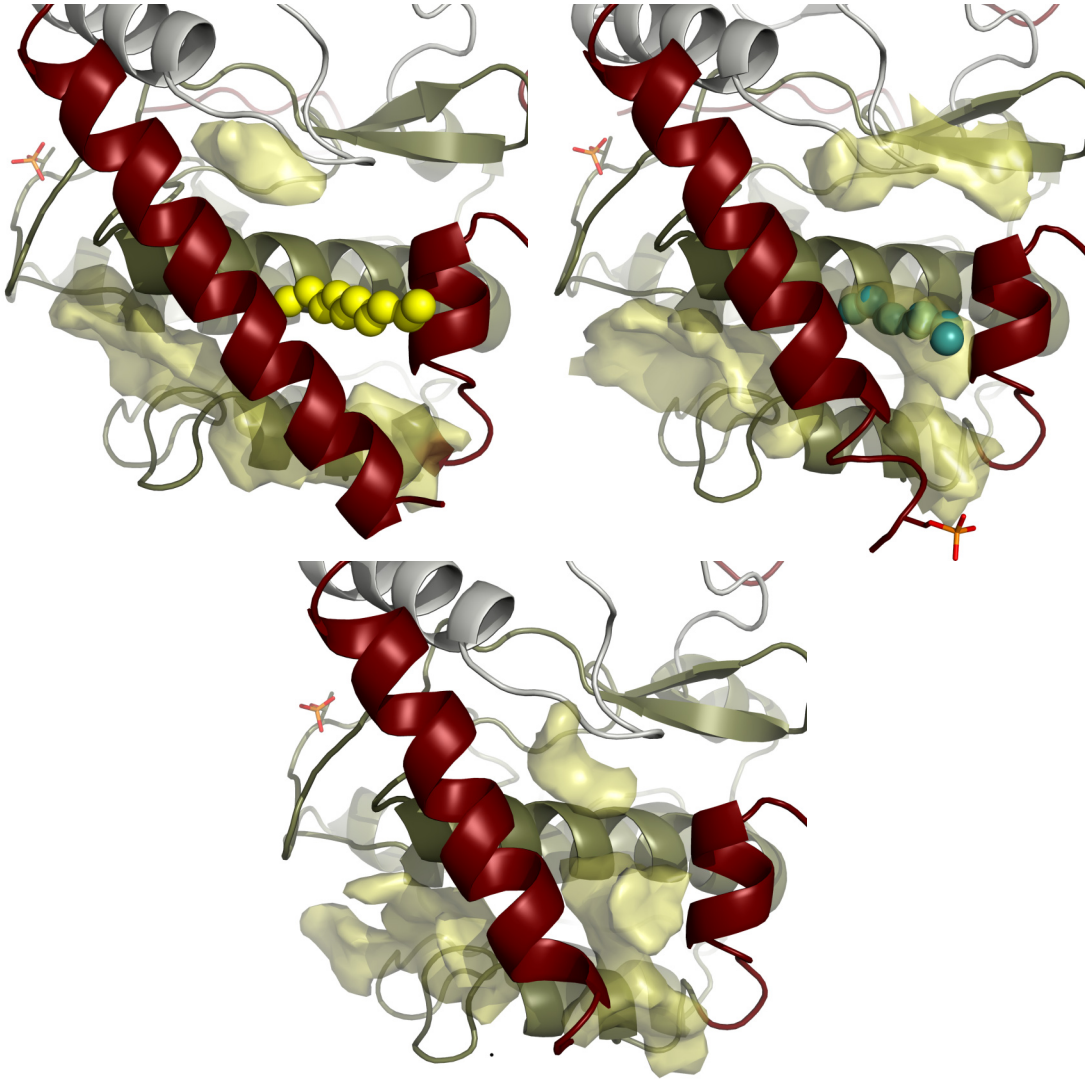

Figure 3-11. N-terminal Tail and Acyl Pocket.
The top left shows myristylated PKA, with the atoms from the myristic acid that were resolved in the crystal structure shown as yellow spheres. The top right is PKA crystallized with MEGA-8. The detergent molecule is shown as green spheres and can be seen occupying a computationally identified pocket. The bottom figure is PKA without detergent or myristic acid, and there is an empty pocket at the same site.

Section 3.08 Pockets of Unknown Function

The algorithm identifies a potentially interesting site adjacent to a highly conserved proline residue. Rather than a single proline, PKA has a distinctive pro-pro sequence, residues P236 and P237. The pocket is at the "bottom" of the canonical view of the kinase core, adjacent to the G-helix and the G-H linker. The function of the proline motif is presently unknown.

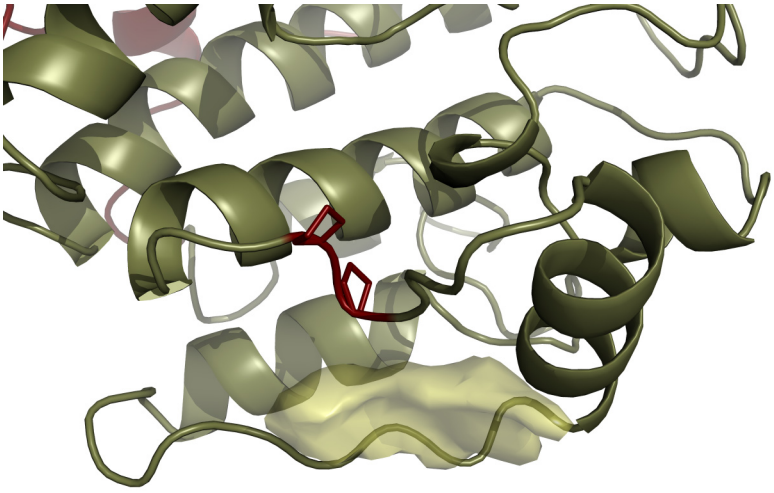

Figure 3-12. Proline Motif and Associated Pocket
P236 and P237 are shown in red. The pocket is adjacent to the G-helix on the right of the figure.

In Figure 3-11 in the myristylation section above, there is a pocket next to the N-terminal tail in all three of the kinases shown. In the myristylated structure with the best resolved N-terminal tail, the pocket can be seen to extend underneath the end of the N-terminal tail. If the N-terminal tail were to move away from the catalytic core, the site would be exposed for interaction. There is another large pocket of unknown function at the hinge region over the linker and αC-β4 loop, shown in Figure 3-13. The pocket is not occupied in any of the PKA structures solved to date.

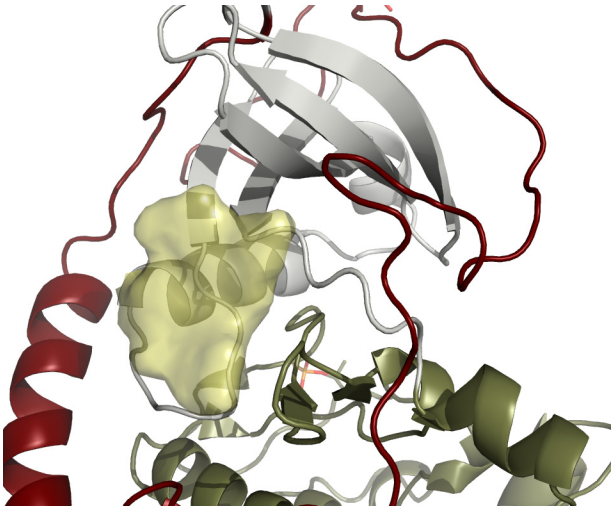

Figure 3-13. PKA Ribbbons with Hinge Region Pocket.
PKA ribbons from 2CPK showing a large pocket at the hinge region, over the linker and αC-β4 loop.

Finally, the algorithm is sensitive enough to detect a set of conserved, buried water molecules. The pocket of water is adjacent to the F-helix inside the molecular surface of the kinase. The buried pocket is present in 2CPK as well as the newer, high-resolution crystal structure.

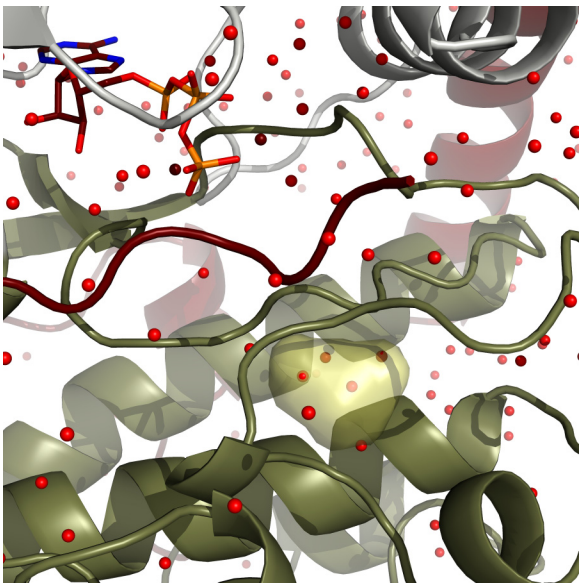

Figure 3-14. Buried Water

High-resolution PKA structure showing water molecules in and around the structure. Four water molecules are included in the gold-colored pocket.

Section 3.09 PKA Holoenzyme

The PKA holoenzyme can also be analyzed with the pocket-finding algorithm. The R1-α, RII- α, and RII-β holoenzymes were analyzed by computing pockets over the catalytic and regulatory subunits separately and combined. Pockets present in the separate subunits that disappear when the algorithm is run on the holoenzyme are filled, and illustrate areas of shape complementarity. If a pocket is also present on the apo-enzyme it can be inferred that the fit is not induced. Pockets may be on either the catalytic or regulatory subunit. The computation shows considerable similarities between the RI-α and RII- β holoenzymes. The RI-α and RII-β holoenzyme complexes are pictured in

Figure 3-15. There are two computationally identified pockets involved in the intramolecular interaction between the catalytic subunit and the fragment of RII-β in the crystal. The active site clefts, pictured in Figure 3-16, and Figure 3-17, show surface geometry very similar to the PKI-bound enzyme in Figure 3-4. Again, the P-2 and P-3 arginines occupy the computationally identified peptide docking site within the active site cleft. The P+1 site is occupied by a hydrophobic residue, and P+2 is occupied by either serine or cysteine.

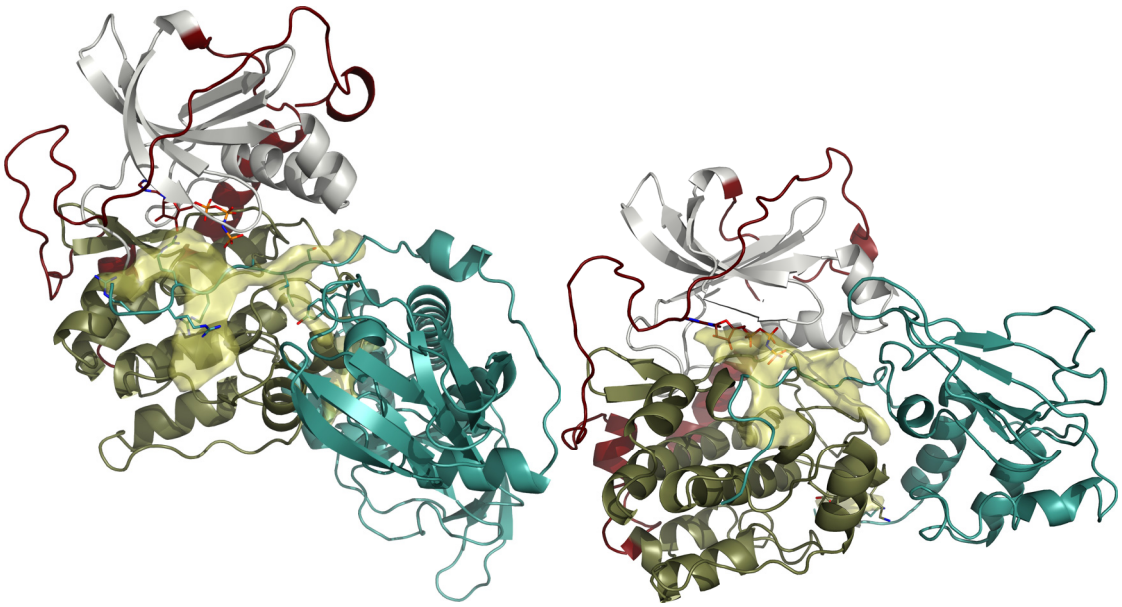

Figure 3-15. Overiew of PKA RI-α and RII–β Holoenzymes
The PKA catalytic subunits are pictured in grey and tan and the regulatory subunits in teal. The complex on the left is RI-α and on the right is RII–β. The gold surface shows the two computationally identified pockets where there is an interaction between the subunits.

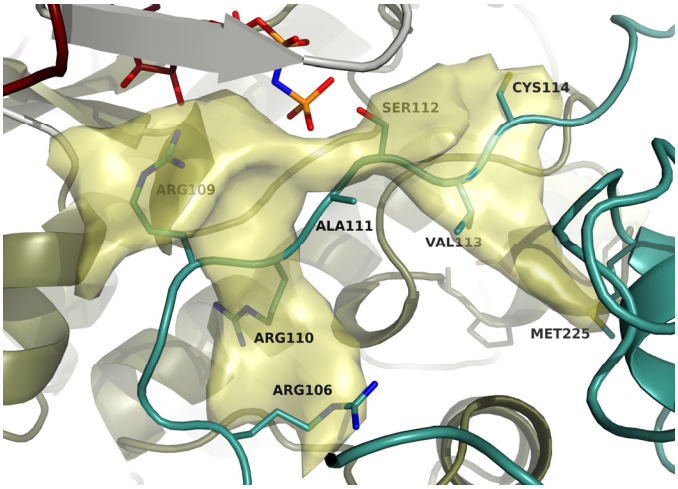

Figure 3-16. Active Site Cleft in RII-β Holoenzyme.
The computation on the catalytic subunit is shown as a gold surface. The peptide binding site is filled by the substrate peptide of RII-β.

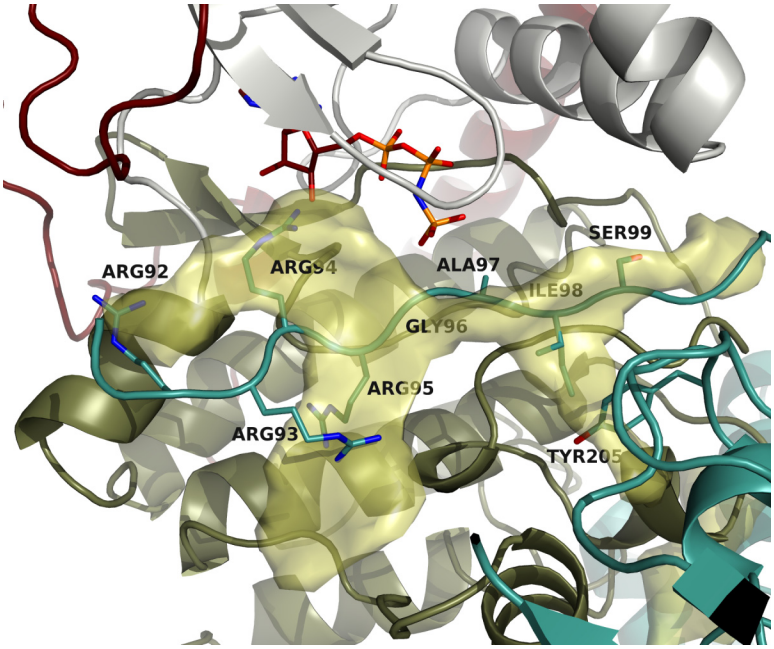

Figure 3-17. Active Site Cleft in RI-α Holoenzyme
The computation on the catalytic subunit is shown as a gold surface. The peptide binding site is filled by the pseudosubstrate of RI-α.

The other potential interaction site identified on the catalytic subunit is at the APE motif. The pocket matches the one pictured in Figure 3-10 that is located on the catalytic subunit adjacent to the APE motif; however, the pocket is not occupied by any amino acids but rather water.

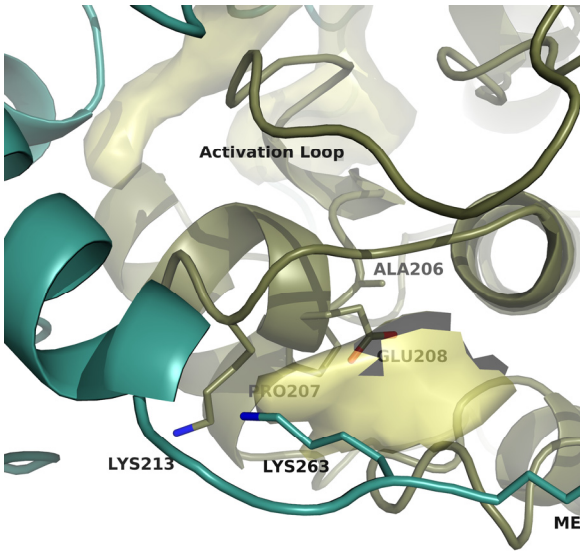

Figure 3-18. Close-up of APE Motif and Activation Loop
Close-up of RII-β subunit interaction at APE motif. The pocket adjacent to the APE motif is not filled, but interaction with it would be blocked.

On the regulatory subunit, there is a pocket identified by the computation that is present on both RI-α and RII–β. The pocket is adjacent to the BC helix and near phospotyrosine 197 on the C-subunit. In both isoforms, L211, S212, and W196 from the catalytic subunit have atoms in the computationally identified pocket. The RI-α also has K213 in the pocket, while in RII–β L198 is in the pocket.

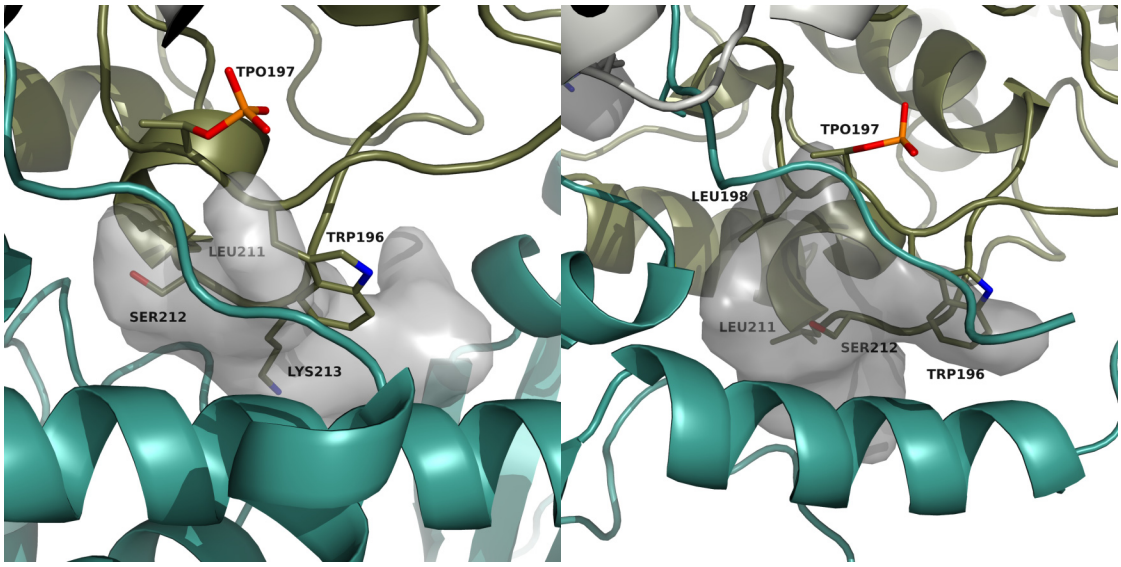

Figure 3-19. Interaction of PKA Catalytic Subunit with Regulatory Subunit
Pockets shown in grey were calculated on the regulatory subunit. RI-α is depicted on the left and RII–β on the right. The helix running left-to-right is the BC helix. Amino acids in the pockets are slightly different, with residues 211-213 and W196 interacting with R1-α. Instead of K213, the RII- β pocket shows an interaction with L198 from the activation loop.

The RII-α holoenzyme is computationally distinct from the other two holoenzymes. The N-lobe of the catalytic subunit is poorly organized in the crystal structure, with many of the side chain heavy atoms unresolved. Figure 3-20 shows an overview of the holoenzyme, with the cartoon of residues with missing atoms colored yellow. The N-lobe is not ordered as well as in the other holoenzymes, and there is a large pocket that is partly an artifact of the missing residues. The peptide docking site was resolved well in the structure, but there is not a distinct cleft as is present in the other two holoenzymes. This is the only PKA structure identified thus far where there is a peptide crystallized in the peptide binding site, but no clear pocket when the peptide is removed. The regulatory subunit has a similar BC helix pocket, shown in Figure 3-21. S212 and K213 from the catalytic subunit fall in the pocket, but W196

does not. The combination of the disorganized N-lobe and lack of a peptide binding site indicate that the organization of the catalytic subunit is fundamentally different in the RII-α holoenzyme crystal structure. It is unclear whether part of the differences are due to lack of the B-domain of the holoenzyme.

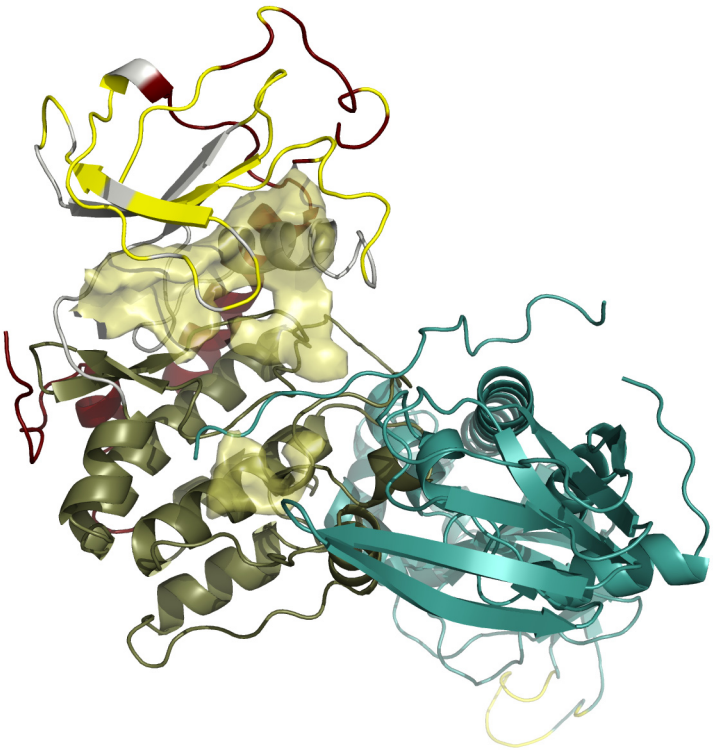

Figure 3-20. Overview of RII-α Holoenzyme
Overview of the RII-α holoenzyme. The N-lobe cartoon is colored yellow where there are residues with missing atoms. Much of the large pocket in the N-lobe is an artifact of the missing atoms.

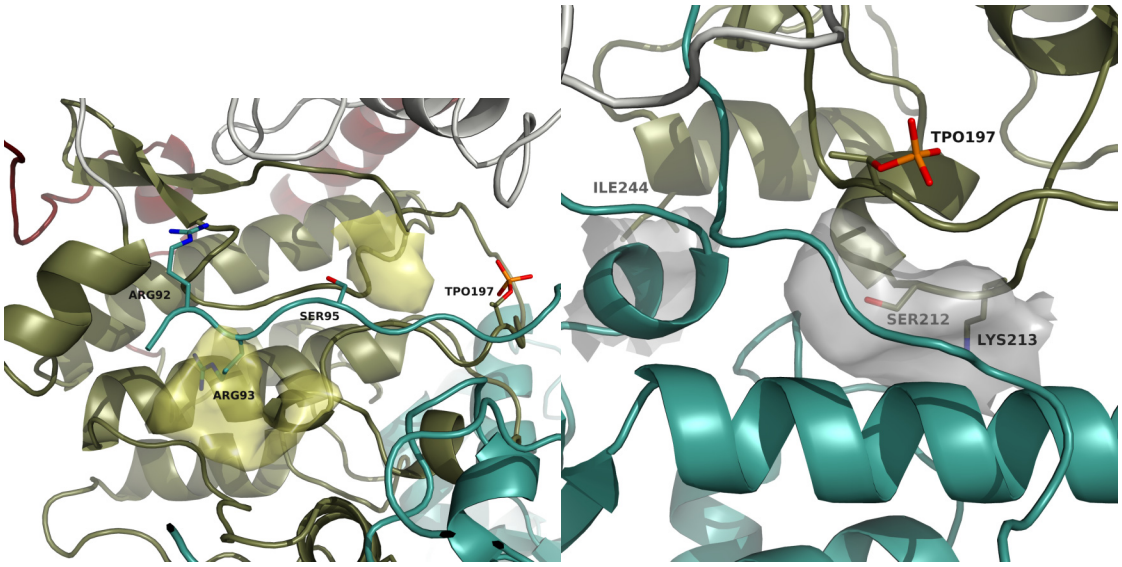

Figure 3-21. Peptide docking site and BC Helix in the RII-α Holoenzyme
The peptide docking site is on the left, with pockets in gold. Note the lack of a long crevice for the peptide backbone in this structure. On the right is the BC helix pocket on the regulatory subunit. S212 and K213 fall in the pocket, but W196 does not.

Section 3.10 Conclusions

The combination of FADE and Crevasse has identified many known structural features of PKA, and some novel regions of potential interest. The algorithm identifies the binding site on the C-helix for the FxxF motif on the C-terminal tail and F26 and W30 from the C-terminal tail. The binding of the tails to the C-helix is essential for PKA activation. The substrate binding site is also clearly identified, and the algorithm shows the geometric continuity of the active site cleft. The peptide binding site is formed in the closed apo structure, and the PKI bound structure shows how the pseudosubstrate backbone and P-2, P-3, and P-6 arginines fit into the peptide docking site. There is also a pocket next to the APE motif, a site of protein-protein interactions, and a pocket where the myristic acid modification on the N-terminal tail

fits on the nonmyrstylated structures. A buried pocket is computed where there are ordered water molecules in the C-lobe.

There are other potentially interesting pockets on the PKA surface that are not occupied in the crystal structure. Residue 40 fits into a pocket at the top of the N-lobe, on top of the kinase spine. There is a pocket next to the conserved PP motif at residue 236 and one under the N-terminal tail that would be exposed if the N-terminal tail moved away from the surface. There is a large pocket between the lobes over the αC-β4 loop and linker region. The function of these sites is presently unknown.

When the computation is run on the holoenzyme, two sets of pockets are computed, one on the catalytic subunit and one on the regulatory subunit. In the RIα and RIIβ holoenzyems, the pseudosubstrate peptide falls into a very clear pocket at the peptide docking in the active site cleft. On the regulatory subunit, there is a large pocket at the BC helix occupied by PKA near W196 and S212. The computation shows differences in the RIβ holoenzyme. The N-lobe is disorganized and the pocket at the active site cleft is absent, with the pseudosubstrate occupying a much flatter surface. The BC helix pocket is present on the regulatory subunit, but it is occupied by slightly different amino acids from PKA.

The combination of FADE and crevasse provides a tool to confirm interactions on a well-studied, conserved structure. It also is able to identify potential new regions of interest and potential protein-protein interactions to study experimentally.

The description of Crevasse and FADE in Chapter 2 and the PKA results in Chapter 3 will be written up for publication. The dissertation author will be the

primary investigator and author. Dr. Choel Kim provided the new high-resolution crystal structure of PKA used in Chapter 3 and Mike Pique helped with the optimization of FADE detailed in Chapter 2. Both will be co-authored. Dr. Susan Taylor and Dr. Lynn Ten Eyck will also be co-authors on the paper.

Cheng X, Ji Z, Tsalkova T,Mei F. (2008) Epac and PKA: a tale of two intracellular cAMP receptors. *Acta Biochim Biophys Sin 40(7)*:651-662.

Hanks SK, Hunter T. (1995) Protein kinases 6. The eukaryotic protein kinase superfamily: kinase (catalytic) domain structure and classification. *FASEB J. 9*(8):576-96.

Huse M, Kuriyan J. (2002) The Conformational Plasticity of Protein Kinases. *Cell 109*: 275-282.

Johnson DA, Akamine P, Radzio-Andzelm E, Madhusudan, Taylor SS. (2001) Dynamics of cAMP-Dependent Protein Kinase. *Chem. Rev. 2001(101)*:2243-2270.

Karlsson R, Zheng J, Xuong HN, Taylor SS, Sowadski JM. (1993) Structure of the mammalian catalytic subunit of cAMP-dependent protein kinase and an inhibitor peptide displays an open conformation. *Acta crystallogr., Sect. D 49*: 381-388. (1ctp)

Kornev AP, Haste NM, Taylor SS, Ten Eyck LF. (2006) Surface comparison of active and inactive protein kinases identifies a conserved activation mechanism. *Proc Natl Acad Sci USA.;103(47)*:17783-8.

Knighton DR, Zheng JH, Ten Eyck LF, Ashford VA, Xuong NH, Taylor SS, Sowadski JM (1991) Crystal structure of the catalytic subunit of cyclic adenosine monophosphate-dependent protein kinase. *Science 253*:407-414 (2cpk)

Madhusudan, Trafny EA, Xuong NH, Adams JA, Ten Eyck LF, Taylor SS, Sowadski JM. (1994) cAMP-dependent protein kinase: crystallographic insights into substrate recognition and phosphotransfer. *Protein Sci. 3*: 176-187. (1jbp)

Narayana N, Cox S, Xuong NH, Ten Eyck LF, Taylor SS. (1997) A binary complex of the catalytic subunit of cAMP-dependent protein kinase and adenosine further defines conformational flexibility. *Structure 5*: 921-935. (1bx6)

Natarajan K, Haste N, Taylor SS, Neuwald AF. (2007) The hallmark of AGC kinase functional divergence is its C-terminal tail, a cis-acting regulatory module. *Proc Natl Acad Sci U S A. 104(4)*: 1272–1277.

Word JM, Lovell SC, Richardson JS, Richardson DC. 1999. Asparagine and glutamine: using hydrogen atom contacts in the choice of sidechain amide orientation. *J. Mol. Biol. 285*, 1735-1747.

Zheng J, Knighton DR, Xuong N, Taylor SS, Sowadski JM, Ten Eyck LF. (1993) Crystal structures of the myristylated catalytic subunit of cAMP-dependent protein kinase reveal open and closed conformations. *Protein Sci. 2*: 1559-1573.

Chapter 4. Mapping the Kinase Family

A new method using spatial clustering of pockets computed with FADE and Crevasse was used to compare and contrast surface geometry in the protein kinase family. Mammalian protein kinases have a conserved catalytic core highly homologous to Protein Kinase A. There is an N-lobe, a C-lobe, a C-helix, and a conserved active site cleft where ATP and substrate bind and phosphoryl transfer occurs. The catalytic core may be the majority of the molecule, as with PKA, or it may be only a small part of a much larger molecule, as in the receptor tyrosine kinases. Some kinases like c-Src have multiple regulatory domains, others like PKA have separate regulatory subunits. Comparing and contrasting the conserved catalytic core of the kinase family can provide insight into substrate recognition, activation mechanisms, and possible conserved sites of protein-protein interaction.

Ten kinase core x-ray crystal structures were selected for the comaparison, six serine/threonine and four tyrosine kinases. Only active structures in a closed conformation with the R-spine in a catalytically competent position were used (Kornev et al. 2006). The structures were truncated to include only the catalytic core. The alignment and subdomain definition provided Hanks and Hunter (1995) was used to determine where to truncate the proteins. The beginning of subdomain I was defined as a residue homologous to PKA D41, and the end of subdomain XI was defined as homologous to PKA F297. Waters, ligands, and metals were removed, leaving only the polypeptide. Alternate atom locations were also removed. No attempt was made to rebuild missing side chains or side chain atoms because of the

difficulty of determining a location for the rebuilt atoms. All ten kinase cores were pairwise structurally aligned on the backbone atoms of the F helix against 2CPK residues 218-230 using Pymol (DeLano Scientific). The PDB IDs, names, and residues of the ten kinases are listed in Table 4-1 and the aligned kinase catalytic cores are shown in Figure 4-1.

Table 4-1. Kinases Used in the Comparison

Type	Abbrev.	Name	PDB ID	Residues	F Helix
Ser/	CDK	Cyclin Dependent Protein Kinase	1FIN (A)	2-286	183-195
Thr	DAPK	Death Associated Protein Kinase	1JKK	11-275	197-209
	PHK	Phosphorylase Kinase	2PHK	17-287	209-221
	PKA	cAMP Dependent Protein Kinase	2CPK	41-297	218-230
	Sky1P	Sky1p SR Protein Kinase	1Q97 (A)	156-706	584-596
	ERK2	MAP Kinase	2ERK	21-311	206-218
Tyr	IRK	Insulin Receptor Kinase	1IR3	994-1263	1189-1201
	CSK	Carboxyl Terminal Src Kinase	1K9A (A)	193-443	366-378
	LCK	Lymphocye Specific Protein Kinase	3LCK	243-494	420-432
	c-Src	c-Src Proto-oncogene	1Y57	265-516	442-454

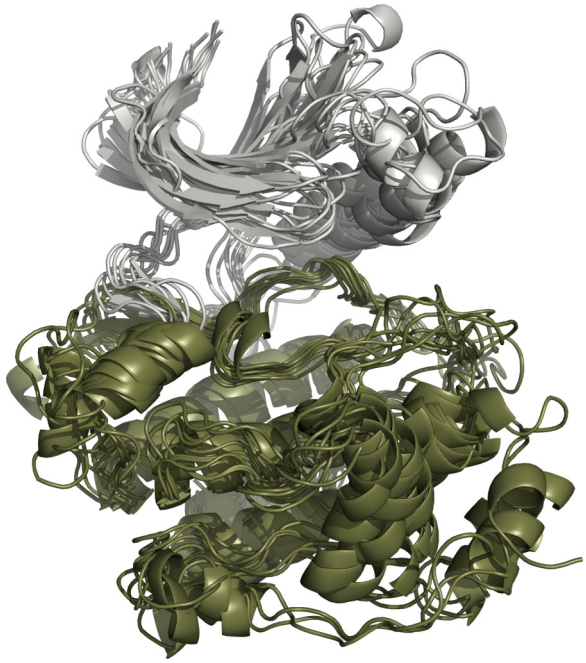

Figure 4-1. Aligned Kinase Catalytic Cores
Catalytic cores of the ten protein kinases, aligned on the F-helix. The N-lobe is shown colored grey and the C-lobe is tan.

After the crystal structures were aligned and processed, FADE and Crevasse were run on each structure as described in Chapter 2. Crevasse was set to discard groups of points with fewer than 80 points or the longest axis shorter than 5Å. As part of the Crevasse computation, the center of each crevice is computed as the mean of the points. To identify groups of potentially conserved binding sites, the crevice centers were spatially clustered using the fastclus procedure in SAS software. New clusters were initiated when the Euclidian distance between clusters exceeded 10 Å. The pockets clustered into 30 distinct spatial clusters. Many of the clusters highlight regions of known conserved function.

The rest of this chapter will present and then discuss the results of this computational experiment. In Section 4.01 through Section 4.05, I will present all of the computational results. Each section will contain results from a particular area of the kinase molecules. In the conclusions in Section 4.06 at the end of the chapter, I will discuss the interpretation of all of the data.

Section 4.01 Active Site Cleft

The kinase active site cleft was identified by the software as the largest crevice in all ten proteins and the centers of the active sites clustered tightly. Table 4-2 shows the clustering results for the active site cleft. The active site is 30 Å long and irregularly shaped, yet the site centers in all ten kinase cores fall no more than 2.57 Å from the cluster centroid.

Table 4-2. Active Site Cleft Cluster Centroid Distances
The second column is the distance in angstroms of the crevice center from the k-means cluster centroid.

Kinase	Å from centroid
CDK	1.16
CSK	1.53
DAPK	1.78
ERK2	1.33
IRK	1.77
LCK	1.74
PHK	0.74
PKA	2.12
Sky1P	1.01
c-Src	2.57

The results of the computation are depicted in

Figure 4-2 and

Figure 4-3.

Figure 4-2 shows all ten active site clefts overlaid on PKA ribbons. In all ten kinases, the nucleotide and peptide binding sites are identified as a single large, geometrically contiguous active site cleft with similar geometry. Both the N-lobe and the C-lobe contribute to both the ATP docking site and the peptide docking site. There are slight differences in the specific shapes of the active site clefts, illustrated in

Figure 4-3, with closely homologous proteins having the most similarly-shaped clefts. For example, the tyrosine kinases c-Src and CSK have active site clefts that overlap well, and the serine/threonine kinases Sky1P, PKA, and CDK are similar.

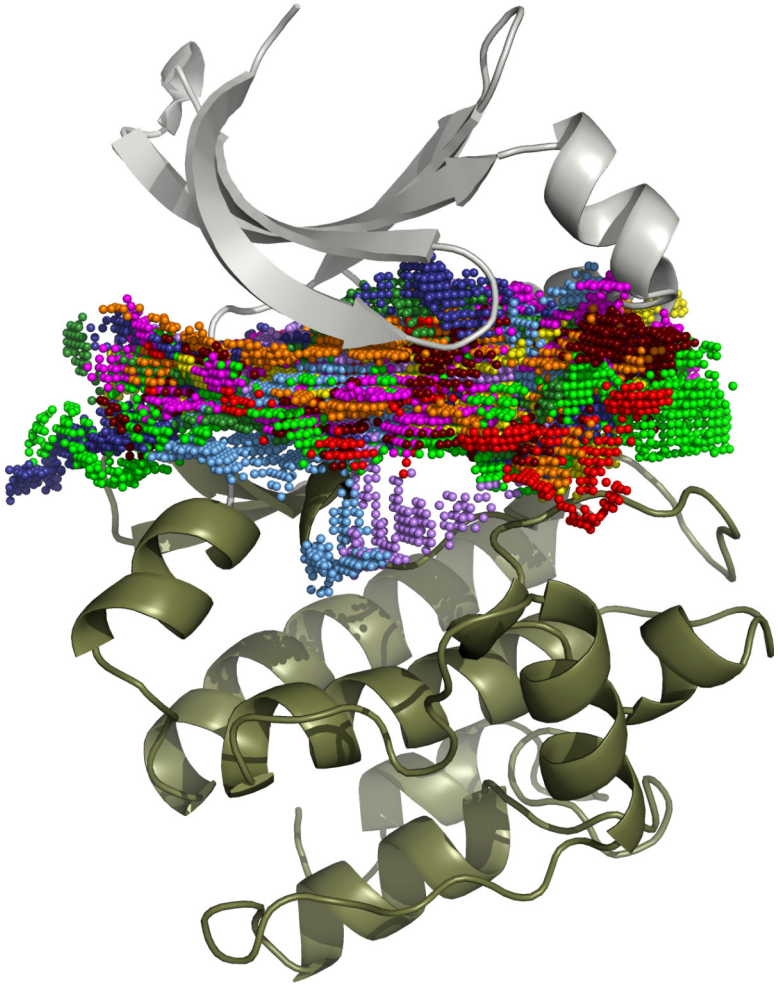

Figure 4-2. Active Site Cleft Computation
All ten active site clefts are depicted overlaid onto ribbons from the PKA catalytic core. The colored spheres represent grid points identified as lining the active site cleft. Points from each of the ten kinases are shown in a different color. (Not all points are visible, as PyMol renders based on the order the files are loaded.)

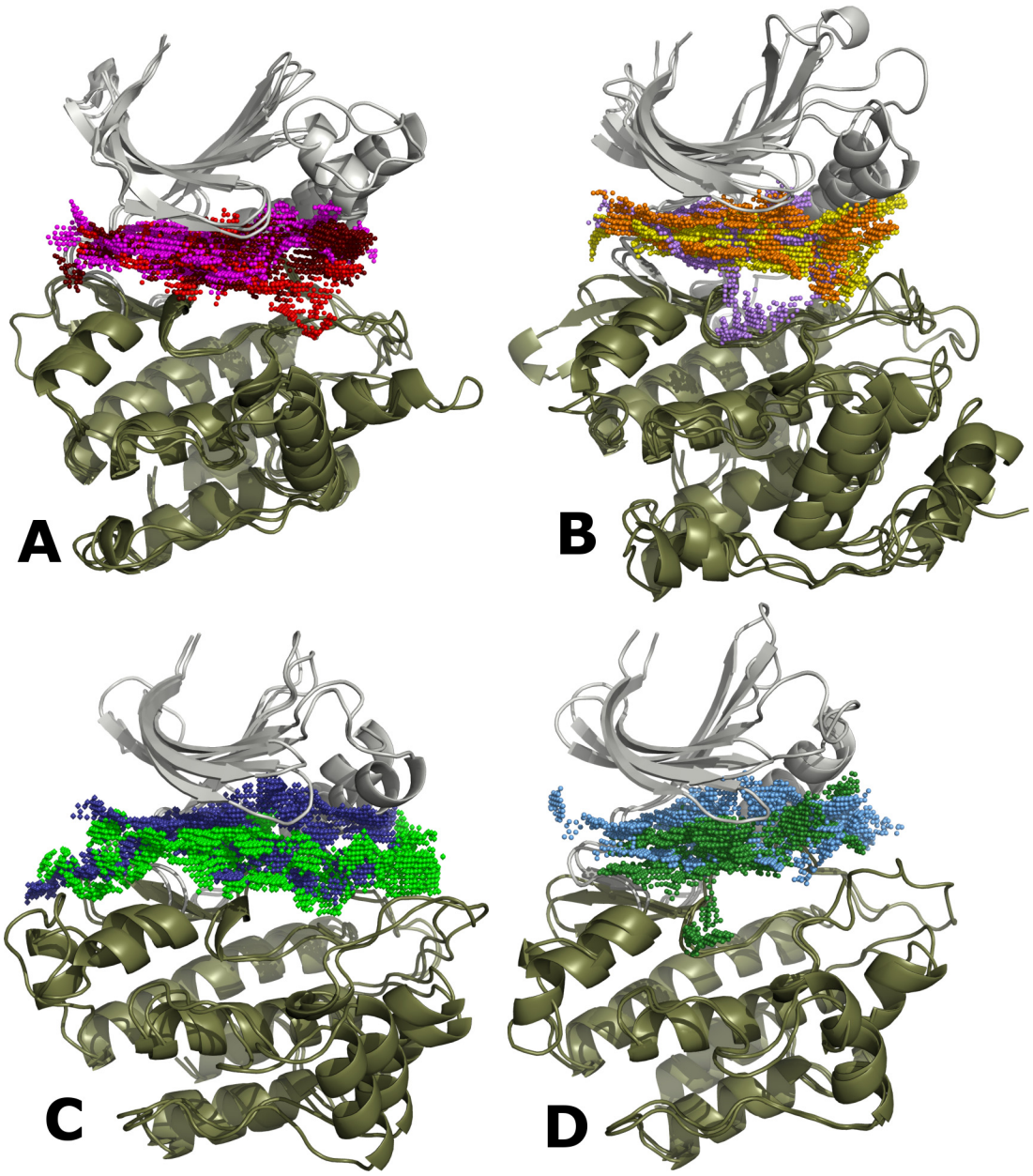

Figure 4-3. Active Site Cleft Detail, by Substrate.
A, B. Active sites for serine/threonine kinases. A. DAPK (maroon), PHK (magenta), PKA (red) B. Sky1p (yellow), CDK (orange), ERK2 (purple). C, D. Active sites for tyrosine kinases. C. IRK (green), LCK (dark blue). D. CSK (deep green), c-Src (light blue).

Section 4.02 Substrate Specificity

There is a computational difference between tyrosine and serine/threonine kinases outside of the active site cleft where phosphorylation substrates bind. Table 4-1 shows the results of the clustering computation. Two different clusters of pockets were identified, one cluster with only serine/threonine kinases and the other with only tyrosine kinases. PHK has two sites in the cluster because Crevasse separated the points into two small, adjacent sites rather than one larger site.

Table 4-3. Substrate Specificity Clusters
Computational results for the substrate recognition site clusters.

Serine/Threonine Cluster		Tyrosine Cluster	
Kinase	Distance (Å)	Kinase	Distance (Å)
CDK	0.82	CSK	3.28
DAPK	0.76	IRK	2.29
ERK2	2.49	LKC	3.86
PHK	4.87	c-Src	3.94
PHK	1.85		
PKA	4.33		
Sky1P	1.88		

The two sets of clustered pockets are depicted in

Figure 4-4. To find out what occupies the pockets, two kinases with substrate co-crystallized were identified. The substrates and pockets are depicted in

Figure 4-5. The PKA crystal structure used in the experiment has PKI in the full structure, so PKI was overlaid onto the pocket. The pocket is occupied by the P-2 and P-6 arginines. The P-3 arginine falls into the large active site cleft pocket. When the full IRK structure is overlaid with the core used in the computation, the backbone of the peptide substrate falls into the computationally identified pocket. The pocket

serves to extend the active site cleft so that the substrate peptide backbone is positioned for the bulkier tyrosine residue to fit into the phosphorylation site. The geometric fit of the peptide backbone and tyrosine residue may be part of the specificity determinant between serine/threonine and tyrosine kinases.

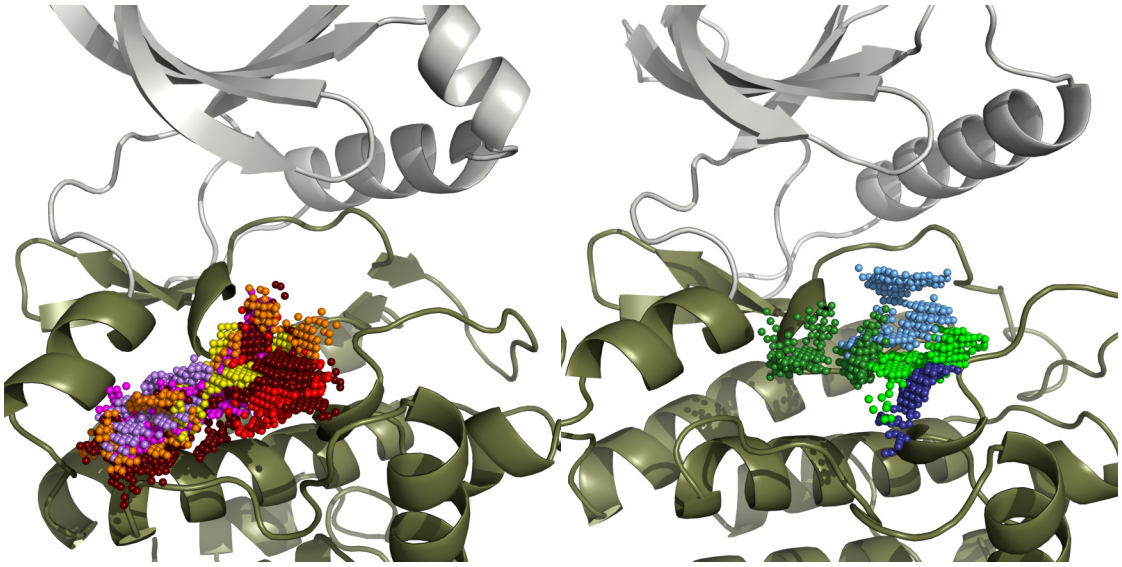

Figure 4-4. Substrate Specificity Determinants
Results of the Crevasse computation. Left: Cluster identified on serine/threonine kinases, pictured on PKA ribbons. Right: Cluster identified on tyrosine kinases, pictured on IRK ribbons.

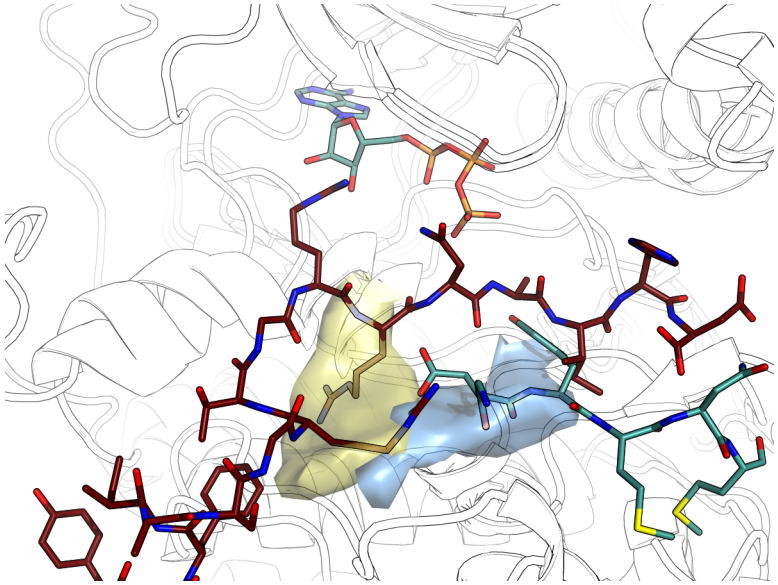

Figure 4-5. Kinases With Bound Substrate
PKA and IRK, co-crystallized with a peptide in the substrate site. PKI is shown in dark red, and the computed pocket in gold. The P-2 and p-6 arginines fall into the pocket. A substrate peptide co-crystallized with IRK is shown in teal, with the computed pocket in blue. The backbone of the P and P-1 residues are in the pocket.

Section 4.03 C-Helix Stabilization

Pockets were identified on multiple kinases at the N-lobe and C-lobe surfaces of the C-helix. The points from the two computational clusters are pictured in figure Figure 4-6. There are pockets on both surfaces in PKA, ERK2, IRK, and DAPK. There are also single pockets on c-Src, CSK, and PHK. Since the PKA FxxF motif falls into the pocket on the PKA core, the other kinases were examined for similar interactions in the full crystal structures.

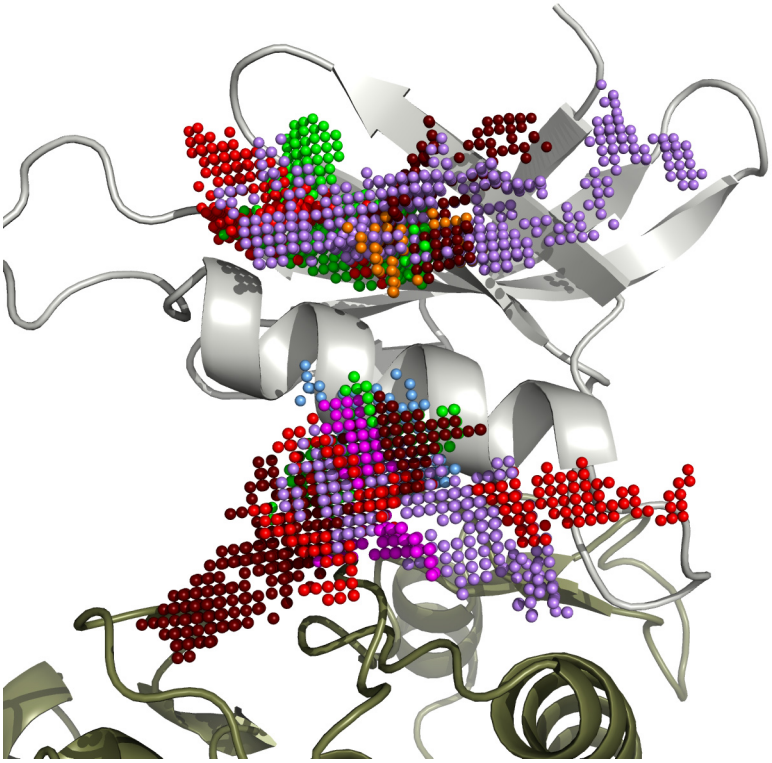

Figure 4-6. C-lobe and N-lobe Surface Pocket Points
Pockets were identified on the N-lobe and C-lobe surfaces of the C-helix on PKA (red), IRK(green), DAPK (dark red), PHK (magenta), ERK2 (purple), CDK (orange), and c-Src (light) blue. Pockets are pictured on DAPK ribbons.

Amino acids occupy the pockets in four of the serine/threonine kinases, PKA, DAPK, Cyclin, and ERK2. The C-helixes from PKA, ERK2, CDK, and DAPK are pictured in Figure 4-7. As shown in Chapter 3, the pockets on PKA are occupied by the C-terminal and N-terminal tails. In ERK2, a long extension of the C-terminal tail occupies both the N-lobe surface and C-lobe surface pockets. Residues F327, M331 and D334 occupy the N-lobe surface, and an α-helix formed by residues 341-350 lies along the C-lobe surface of the C-helix. Residues in the pocket are L341, I345, T349, and F352. T349 forms a hydrogen bond to R89; the rest of the interactions within the computationally defined pocket are nonpolar. In DAPK, the short N-terminal tail

folds over the N-lobe and into the pocket. Amino acids in the pocket are T1 and F3. In CDK, there is a similar interaction, only the C-helix pockets are occupied by amino acids from cyclin. The N-lobe surface pocket is occupied by cyclin residues L299 and F304. Residue H296 is not in the computational pocket, but it is next to the C-helix. Cyclin residues H296 and F304 are structurally homologous PKA residues F346 and F350. The C-lobe surface pocket is occupied by cyclin residues F267, E268, and I270. PHK has nothing occupying the pocket, but PHK has very few amino acids in the crystal structure beyond the kinase core. There is no crystal packing near the C-helix, and water molecules are in the pockets. The pocket is absent from the computation in Sky1P. In IRK, the pocket on the N-lobe surface of the C-helix is filled in the crystal by the N-terminus of an adjacent molecule.

A common feature in the amino acids packed against the N-lobe surface of the C-helix is the presence of at least one phenylalanine residue.

Figure 4-8 shows the overlaid C-helices and the N-lobe surface residues. The alignment is still on the F-helix, but the positioning of the C-helix is similar. The residues at the N-lobe surface are highlighted, and the phenylalanine residues in each structure are clearly visible.

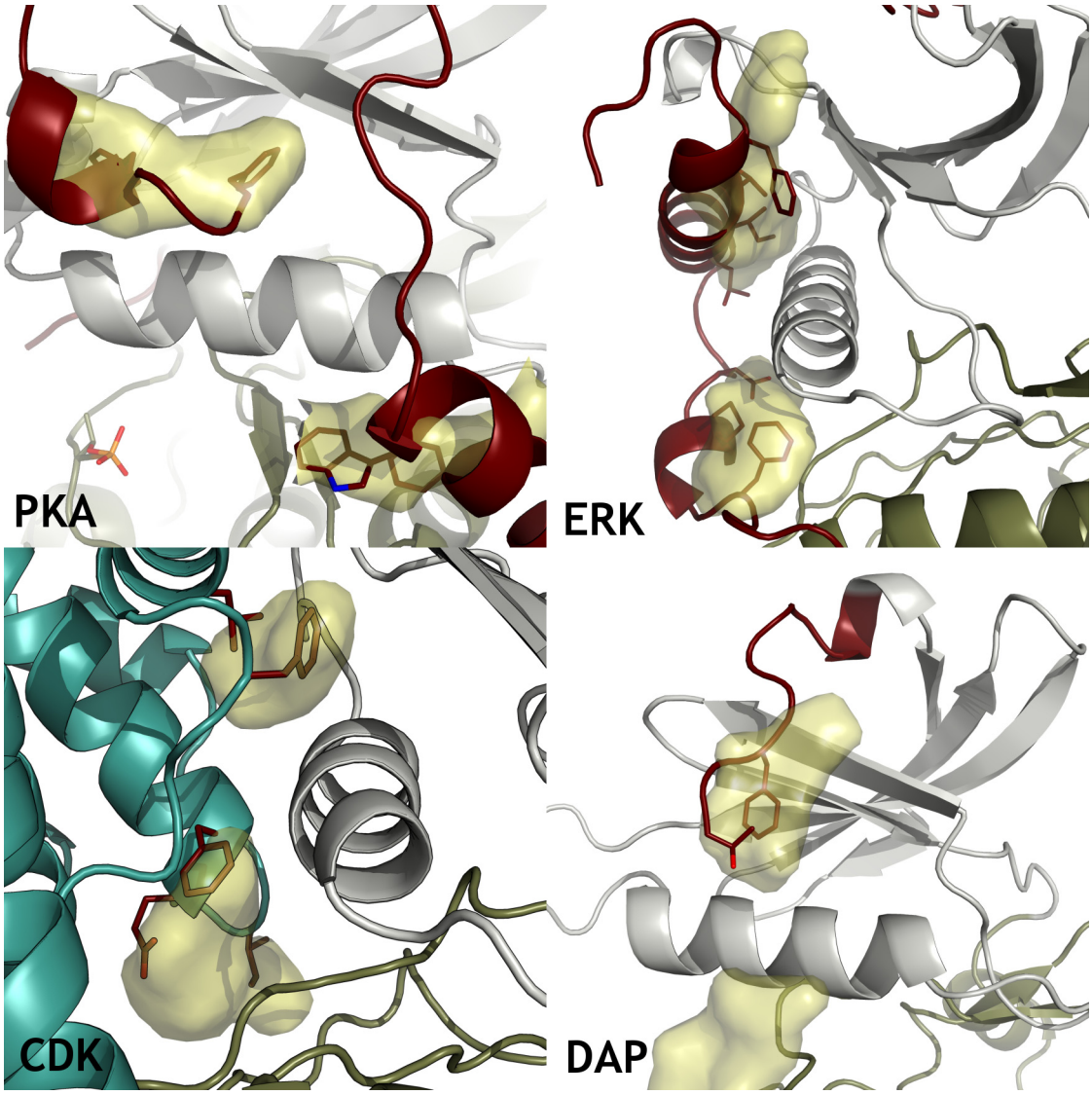

Figure 4-7. C-helix Pockets and Amino Acids Filling Them
The computed pockets are in gold, N-lobe in grey, and amino acids outside the kinase core are shown in red. Cyclin is in teal. ERK2 and CDK are in a different orientation to show the phenylalanine rings clearly.

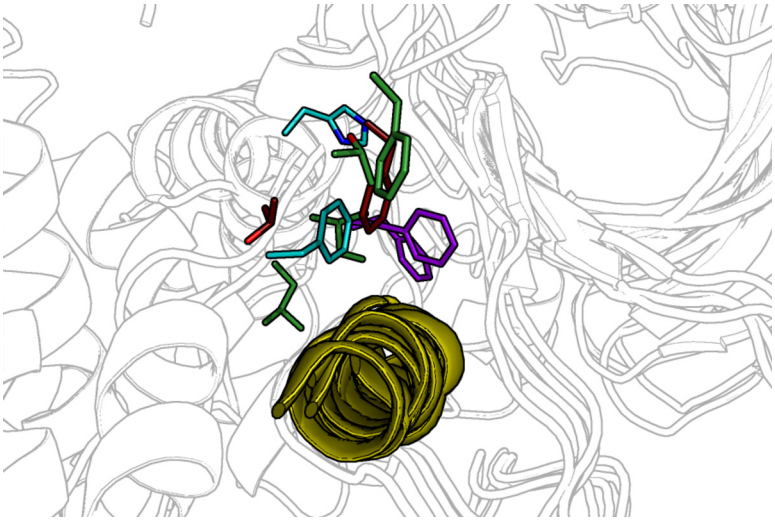

Figure 4-8. Overlaid Residues and C-helix on Serine/Threonine Kinases
The overlaid C-helix is in gold. Residues shown are PKA F347, F350 (purple), Cyclin H296, F304 (teal) DAPK T2, F4 (dark red), and ERK2 L341, I345, T348, F352 (green).

A computationally distinct cluster was present at the c-terminal end of the C helix on two tyrosine kinases, c-Src and IRK. A second computation was run with a 4Å length cutoff for the major axis of the pockets rather than 5Å and a third pocket was found on LCK. CSK shares a homologous tryptophan, but no computational pocket. The computation on c-Src and IRK is pictured in Figure 4-9. In c-Src, W260 is positioned at the c-terminal end of the C-helix with two hydrogen bonds to D258, and E97 from the SH3 domain. Active c-Src is shown overlaid with inactive c-Src (PDB ID 2SRC)

Figure 4-10 to illustrate the role of W260 in C-helix positioning. In inactive c-Src, the C-helix is moved away from the kinase core, and W260 is positioned between the C-helix and the kinase core. There is no hydrogen bond to the tryptophan nitrogen atom.

When the four tyrosine kinases are aligned on the C-helix rather than the F-helix, the conserved tryptophan at the c-terminal end of the helix is tightly structurally conserved in c-Src, IRK, and LCK. All three kinases also have a nearby glutamic acid residue within hydrogen bonding distance of the tryptophan backbone. The position of the CSK tryptophan is slightly different, and the glutamic acid residue is not present in the structure. Instead, F183 is packed next to the C-lobe surface of the C-helix similar to the serine/threonine kinases.

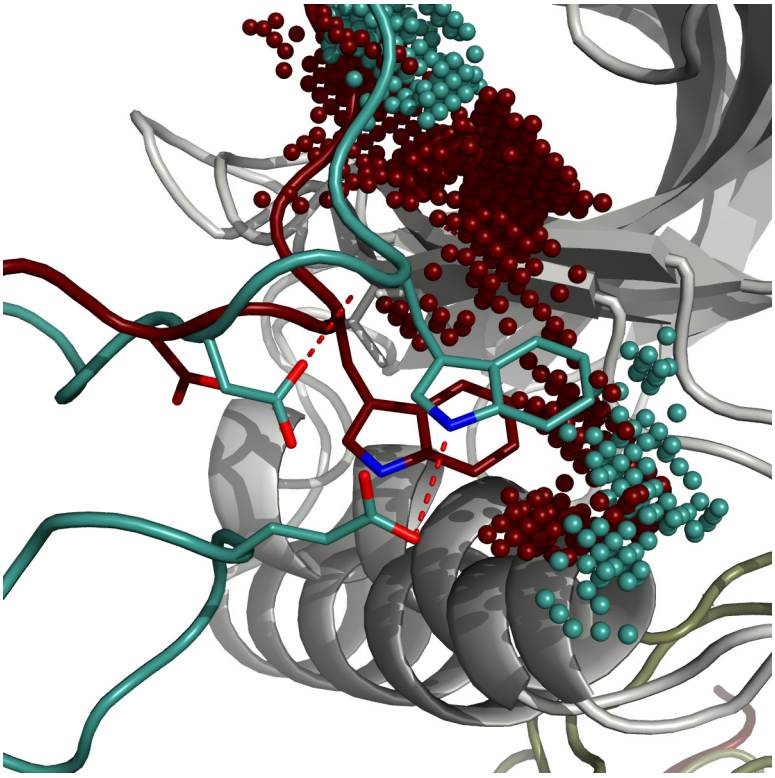

Figure 4-9. IRK and c-Src Conserved Tryptophan
IRK with points from the computation is pictured in dark red, c-Src is pictured in green. c-Src W260 hydrogen bonds to D258, and E97 from the SH3 domain. IRK W989 is within hydrogen bonding distance of the homologous D987, but the hydrogen bond is not formed.

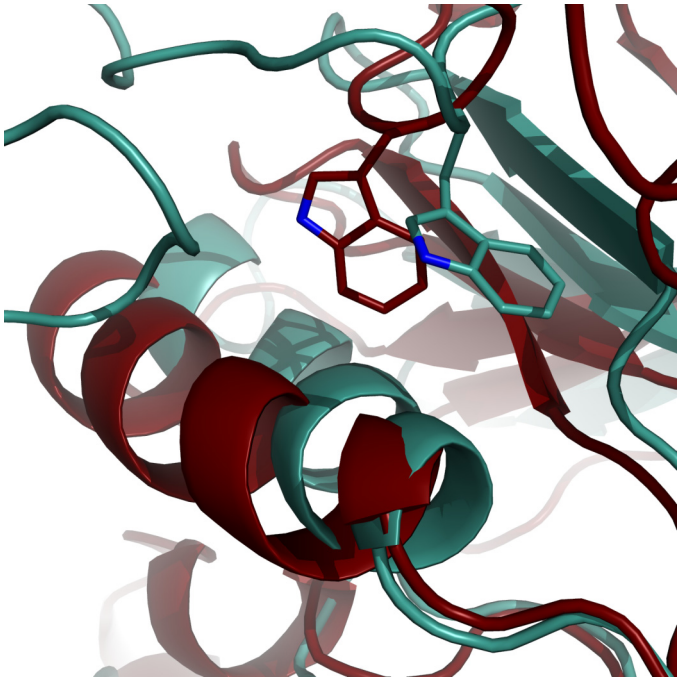

Figure 4-10. C-helix and W260 on Active and Inactive C-Src.
Active is in teal, and inactive in dark red. Proteins are aligned on the F-helix. The C-helix on inactive c-Src is shifted away from the kinase core and W260 is positioned differently.

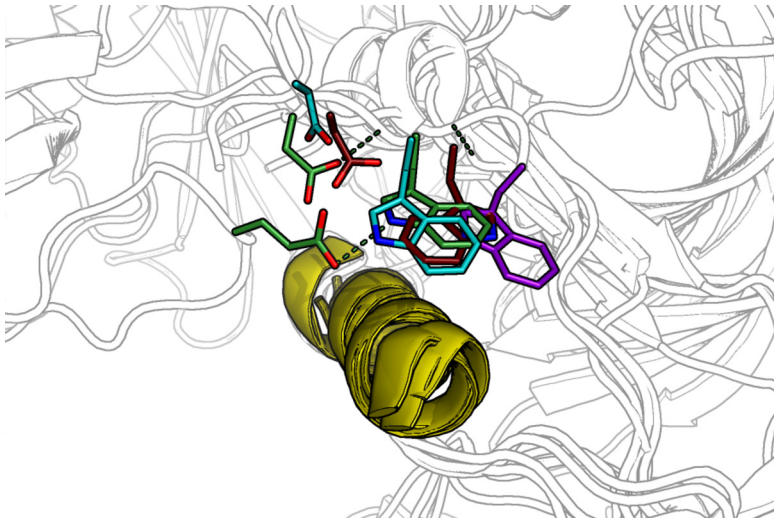

Figure 4-11. Conserved Tryptophan in Four Tyrosine Kinases
Kinases are aligned on the C-helix. Highlighted residues are c-Src E97, D258, W260 (green) IRK D987, W990 (cyan), LKC D236, W238 (dark red), CSK W188 (purple)

Section 4.04 N-lobe Cap

Another shared pocket present on all ten kinases is at the top of the N-lobe. The pocket was shown on PKA in Chapter 3. Table 4-4 shows the computational cluster. The site was split into two smaller sites for LCK and c-Src. Different tuning of the Crevasse connectivity threshold could combine the pockets if necessary.

Table 4-4. N-lobe Cap Cluster
Computatonal cluster for the N-lobe Cap. The pocket on LCK and c-Src was split into two separate small clusters.

Kinase	Distance (Å)
CDK	1.98
CSK	4.63
DAPK	1.01
ERK2	2.51
IRK	1.34
LCK	7.27
LCK	3.52
PHK	0.79
PKA	1.13
SLY	2.41
c-Src	3.77
c-Src	5.85

The points from the Crevasse computation are depicted in
Figure 4-12. Each set of points is depicted in a different color. Differences in the position of the pocket are partly reflective of subtle differences in the position of the glycine loop, since the kinases were aligned on the F-helix rather than the N-lobe.

When the kinases are examined, all of them have between one and four N-terminal residues that occupy the computationally defined pocket. The structures are shown individually in Figure 4-13. Residues in red were removed from the computation as they are not considered to be part of the classical kinase core. The first spine residue, catalytic lysine, and any ligands are shown as well. The pocket tends to

be above the beta sheet with the catalytic lysine and the kinase spine. The structural motif of residues filling the top of the N-lobe is a new feature of the kinase core. Table 4-5 shows the kinase sequences, aligned by homology on Subdomain I to sequences in Hanks and Hunter (1996). The residues on a black background in the table had atoms in the computational pocket. There is no clear sequence homology between the residues outside of the closely related sequences like IRK and LCK; the N-lobe Cap motif is purely structural.

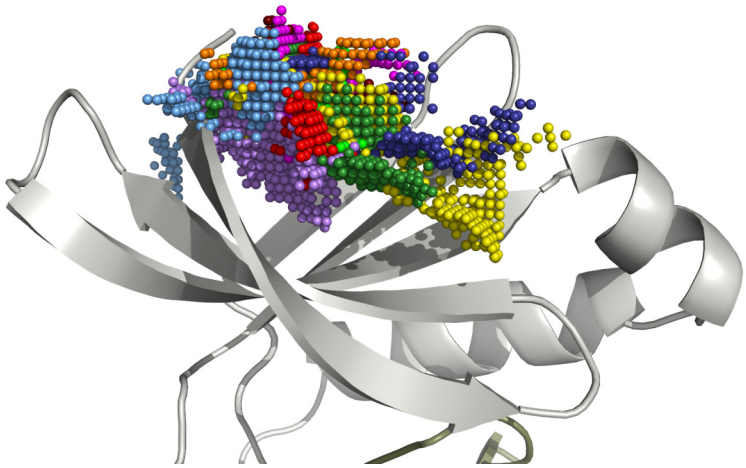

Figure 4-12. N-lobe Cap Cluster on all Ten Kinases
N-lobe Cap on all ten kinases. Points are shown on PKA ribbons. DAPK (maroon), PHK (magenta), PKA (red), Sky1p (yellow), CDK (orange), ERK2 (purple), IRK (green), LCK (dark blue), CSK (deep green), c-Src (light blue).

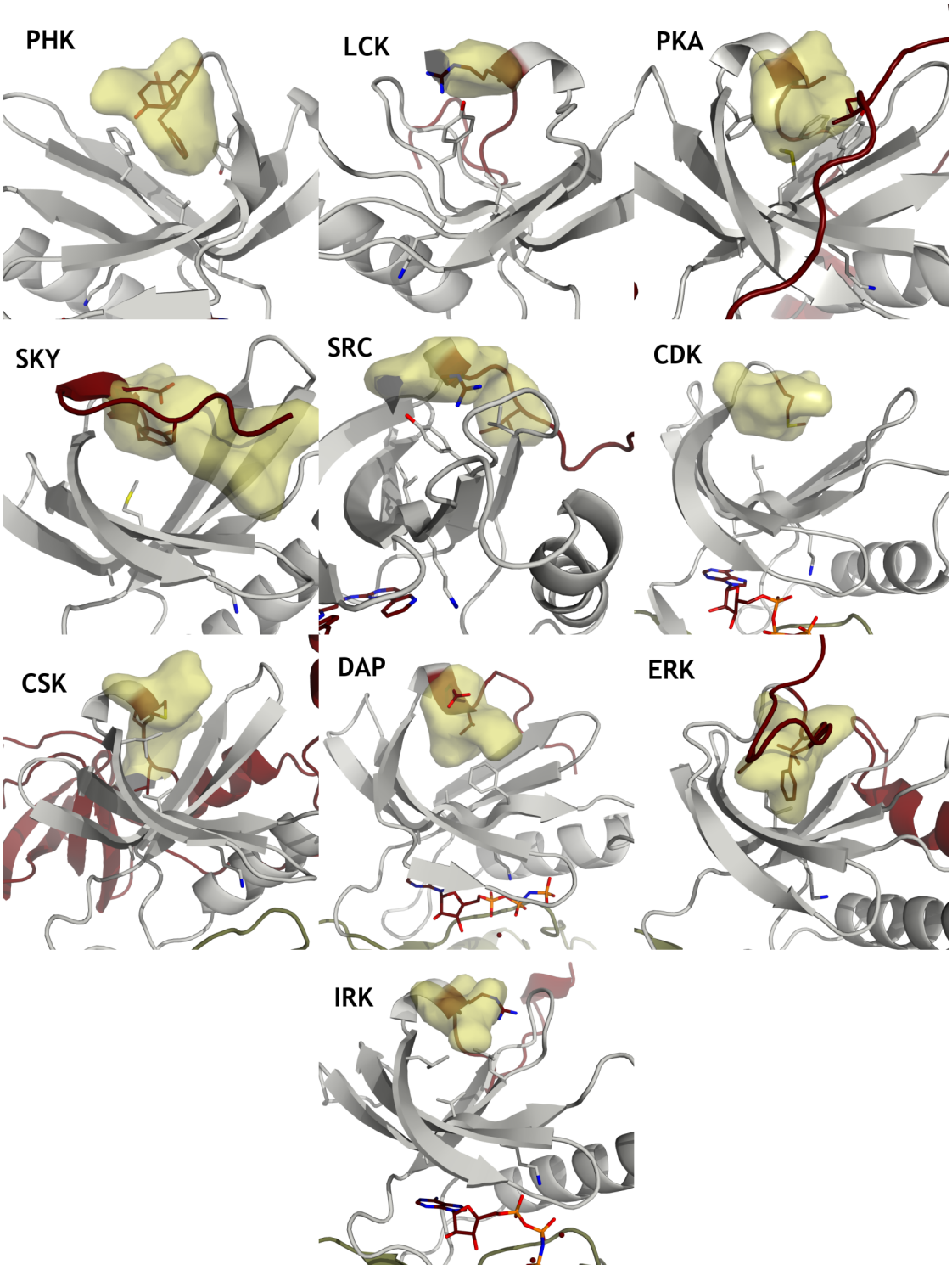

Figure 4-13. N-lobe Caps on all Kinases
The N-lobe cap is shown on all kinases. Amino acids that were not considered part of the kinase core and removed in the computation are shown in dark red. The pocket is shown as a gold surface.

Table 4-5. N-lobe Cap Sequences
The kinase sequences are shown with Subdomain I marked. The residues in white on black have atoms that fall into the computationally identified pocket.

Protein	Residue	Subdomain I
CDK	1	MENFQKVEKIGEGTYGVVYKARNKL
ERK2	17	FDVGPRYTNLSYIGEGAYGMVCSAYDNL
DAPK	8	NVDDYYDTGEELGSGQFAVVKKCREKS
PHK	15	FYENYEPKEILGRGVSSVVRRCIHKP
SRC	262	IPRESLRLEVKLGQGCFGEVWMGTWNG
LCK	240	VPRETLKLVERLGAGQFGEVWMGYYNG
CSK	189	ALNMKELKLLQTIGKGEFGDVMLGDYRG
IRK	991	VSREKITLLRELGQGSFGMVYEGNARDIIKGE
PKA	40	LDQFDRIKTLGTGSFGRVMLVKHKE
Sky1P	153	EPYARYILVRKLGWGHFSTVWLAKDMV

Section 4.05 Conserved Pockets with Unknown function

There are some conserved pockets on the kinase core with unknown function. Some are present on all the kinase cores in the survey, and others show some selectivity. Figure 4-14 shows a cluster of pockets over the αC-β4 loop and the N-lobe and C-lobe linker region that is present on all ten kinases. None of the crystal structures have anything occupying the pocket.

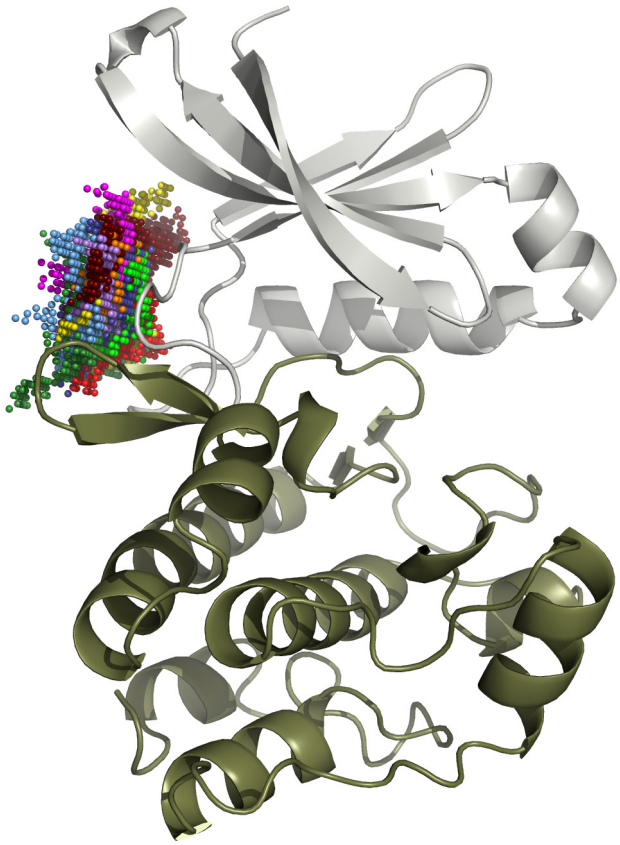

Figure 4-14. Hinge Region Pocket
Pocket present in all ten kinases. The pocket overlays the αC-β4 loop and the linker between the N and C lobes.

There are two pockets common to most of the kinases on the C-lobe as well. The pocket shown on the left in Figure 4-15 is located over the loop between the D and E helices, and at the C-terminal end of the F helix. This pocket, the DEF site, is present in nine of the ten kinases, and is missing only in PKA. There is nothing crystallized at that particular site except in c-Src. In c-Src, the last amino acid on the C-terminal tail, L533, folds into the identified site. The function of the site and the reason it is missing on PKA are difficult to determine from these data.

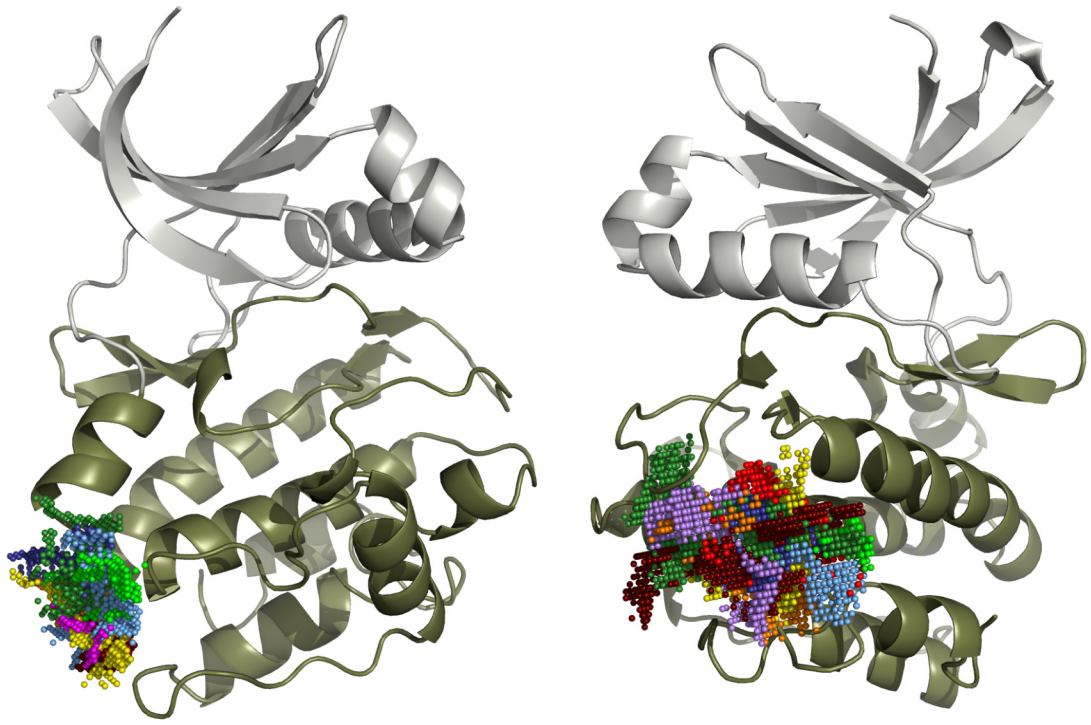

Figure 4-15. Common C-Lobe Sites, the DEF and EF pockets
Left. Pocket present in all kinases except PKA. The pocket is located at the DE loop, and the C-terminal end of the F helix. Right. Pocket in all kinases but PHK. The pocket is at the N-terminal end of the F-helix.

The second shared pocket in Figure 4-15 , the EF site, is on the other side of the C-lobe. It is located at the N-terminal end of the F helix and the C-terminal end of the E helix. The pocket is near where the PKA regulatory subunit binds in the holoenzyme, but is not occupied by regulatory subunit amino acids in any structure crystallized thus far. The EF site in Sky1P is filled by the Sky1P C-terminal tail, shown in Figure 4-16. The pocket is occupied by I729, W732, and part of F733. The Sky1P C-terminal tail is part of the activation of the kinase (Nolen et al., 2001). Deletion of the tail results in a loss of the constitutive activity.

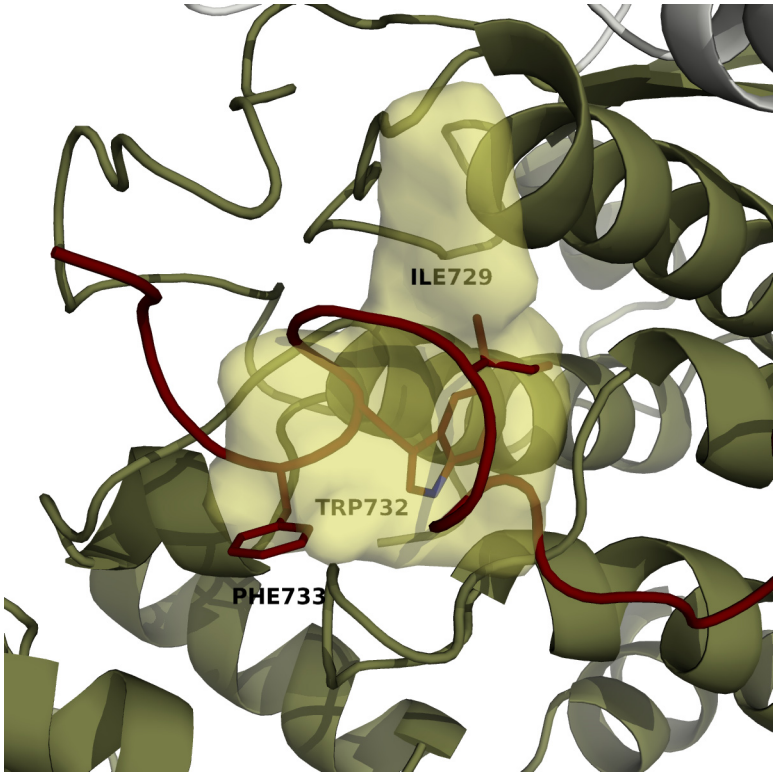

Figure 4-16. Sky1P SR Protein Kinase C-terminal Tail
The EF pocket in Sky1P, occupied by I729, W732, and F733.

Section 4.06 Conclusions

The combination of FADE and Crevasse can be used to make comparisons and find new features within a related group of structurally homologous proteins. In the kinase family, clustering crevasse results shows both similarities and differences between the serine/threonine and tyrosine kinases. The tight active site cleft cluster both validates the pocket clustering methodology and demonstrates the strong degree of conservation of active kinase catalytic cores. The relationship between the active site cleft and the F-helix has been tightly conserved in these ten kinases.

The cluster of pockets at the kinase C-helix shows how this method can pick out similar mechanisms that do not necessarily share sequence homology. Four of the six serine/threonine kinases in this survey have something packed tightly against the C-helix, presumably to maintain the active conformation. The pocket is conserved, but the mechanisms to fill it are quite diverse. PKA has both C- and N-terminal tails packed against the helix. DAPK has "solved" the problem with an N-terminal tail, while ERK has an extended C-terminal tail. CDK has a fourth mechanism, with cyclin packed in against the helix. There is specificity from kinase to kinase, as cyclin does not activate kinases other then CDK kinase tails do not promiscuously activate other kinases.

There are pockets next to the C-helix on PHK, IRK and c-Src as well. The pocket on the N-lobe surface of the IRK C-helix is filled by the symmetry mate. The catalytic core of IRK is only part of the full insulin receptor, so the crystal packing results are difficult to interpret. The pocket on the N-lobe side of the c-Src C-helix is completely empty as is the pocket on the C-lobe surface of the PHK C-helix. The results with W260 suggest that c-Src has a different mechanism of C-helix stabilization. PHK is the smallest kinase in this study, with no N- or C-terminal tails. Again, there is no suggestion as to whether another protein might interact at that site. Finally, Sky1p has no pockets at the C-helix. Sky1P is a constitutively active protein kinase. Its C-helix is unusually long and anchored in an active position by a different method (Nolen et al., 2001).

The other part of the C-helix identified by the algorithm is the C-terminal end in tyrosine kinases. The highly conserved tryptophan residue pictured in Figure 4 11 has been previously identified as important in the tyrosine kinase family. In Hck, a Src family kinase, mutation of the homologous tryptophan to alanine increases activity and disrupts the activation by ligand binding to the SH2 and SH3 domains. All that is required for full activity is autophosphorylation (LaFevre-Bernt et al., 1998). In contrast, mutation of W362 in c-Raf inactivates the kinase (McPherson et al., 2000).

The pocket at the top of the N-lobe is a new finding for the kinase family. The pocket is occupied in all ten of the active structures studied. It is tempting to speculate that the N-lobe cap plays a role in N-lobe stabilization because of its proximity to the conserved lysine (K72 in PKA) and the residue at the top of the R-spine. Packing at the top of the N-lobe could help maintain the orientation of these residues in the active conformation. It would be relatively simple to construct truncated versions of cyclin and PHK to see whether the first few amino acids are important for kinase activity. More kinases should be studied to see whether the top of the N-lobe is consistently occupied in kinase structures. If so, the classical definition of the kinase catalytic core should be revised to include the amino acids that fold onto the top of the N-lobe.

Another new finding is the difference in shape at the substrate recognition site and peptide docking site. This algorithm shows an extra surface at the peptide docking site that positions the bulkier tyrosine residue at the phosphorylation site. It also shows a conserved pocket at the PKA substrate recognition site in serine/threonine kinases. This is the first method that has shown potential for

differentiating between serine/threonine and tyrosine kinases by purely geometric methods. More structures would have to be studied to see whether the difference is consistent, and whether it is maintained in open conformations and inactive kinases.

The large, conserved pockets with unknown functions are also potential areas of study. There is already unpublished research underway that suggests a protein-protein interaction at the αC-β4 loop on a kinase not included in this survey. The EF site is another interesting pocket. It has a clear role in Sky1p, and is close to where the PKA regulatory subunit binds. There are not full structures of all of the PKA holoenzymes available yet, so it remains to be seen whether the site is occupied in any of the PKA holoenzymes. The DE loop pocket shows a difference between PKA and the rest of the kinases studied. It is a rather large pocket in the nine other kinases, but the surface on PKA is quite flat at that site.

Overall, this method is an interesting way of comparing protein surfaces independent of amino acid sequence. The method has identified some of the important motifs in the kinase family, shown a possible way of discriminating between serine/threonine and tyrosine kinases, and highlighted potential regions of interest on the molecules.

The results in Chapter 4 will be published as a new method for comparing protein structures. The dissertation author will be the primary investigator and author. Dr. Kannan Natarajan kindly provided me with F-helix sequences for aligning the kinases in Chapter 4, and both he and Dr. Alexandr Kornev helped me design and

interpret the experiment. Both will be co-authored along with Dr. Susan Taylor and Dr. Lynn Ten Eyck.

DeLano, WL The PyMOL Molecular Graphics System (2002) DeLano Scientific, Palo Alto, CA, USA. http://www.pymol.org

Hanks SK, Hunter T. 1995. Protein kinases 6. The eukaryotic protein kinase superfamily: kinase (catalytic) domain structure and classification. *FASEB J. 9*(8):576-96.

Kornev AP, Haste NM, Taylor SS, Ten Eyck LF. 2006. Surface comparison of active and inactive protein kinases identifies a conserved activation mechanism. *PNAS 103* (47):17783-17788.

LaFevre-Bernt M, Sicheri F, Pico A, Porter M, Kuriyan J, Miller WT. (1998) Intramolecular Regulatory Interactions in the Src Family Kinase Hck Probed by Mutagenesis of a Conserved Tryptophan Residue. *J Biol Chem, 273(48)*, 32129-32134.

Nolen B, Yun CY, Wong CF, McCammon JA, Fu X-D, Ghosh G. (2001) The structure of Sky1p reveals a novel mechanism for constitutive activity. *Nature Structural Biology 8:*176-183.

McPherson RA, Taylor MM, Hershey ED, Sturgill TW (2000) A different function for a critical tryptophan in c-Raf and Hck. *Oncogene 19(32)*: 3616-3622.

Mitchell JC, Kerr R, Ten Eyck LF. 2001. Rapid atomic density methods for molecular shape characterization. *J. Mol Graph Model. 19*(3-4):325-30, 388-90.

Word JM, Lovell SC, Richardson JS, Richardson DC. 1999. Asparagine and glutamine: using hydrogen atom contacts in the choice of sidechain amide orientation. *J. Mol. Biol. 285*, 1735-1747.

Chapter 5. Further Studies

The methodology outlined here could be extended in a number of ways. As outlined here, it is a valuable method of mapping single protein surfaces and comparing the surfaces of small groups of proteins. It would be interesting to examine other families of proteins, expand the computation to other molecular properties, and map larger groups of proteins. It would also be interesting to try clustering pockets determined with other computational methods.

Perhaps the most obvious follow-up study would be adding more kinase structures to the group of ten. There are over 400 kinase crystal structures in the PDB, and the computation is fast enough to compute pockets across them all overnight. There would be a number of scientific and technical problems to address on the larger computation. First, the proteins have to be aligned in a meaningful fashion to compare pockets. The proteins chosen for this study were closed, active kinases. The experiment worked because alignment on the structurally conserved F-helix also gave a reasonable alignment of the N-lobes of the kinases. All of the pockets on the surface could be clustered without regard to their relationship to other parts of the kinase core. Running this methodology on dissimilar structures, like a mixture of inactive and active kinase structures or a mixture of open and closed form kinases, would require a different approach.

The approach for structures that are not crystallized in a similar conformation would be a computation over local alignments of smaller features of the protein. For kinases, it might be possible to align the beta strands, the C-helix, and the F-helix

separately and run the computation multiple times. Then pockets near the aligned feature could be compared. Doing so would require that the conserved structural features be identified in each of the proteins in the study. For this study, the F-helix sequences were kindly provided by Dr. Kannan Natarajan, who has extensive knowledge of domain homology across the kinase family. Similar lists are available for other conserved features of the kinase core. Studying a different family of proteins would require either similar domain knowledge, or a tool to do substructure alignments. Thresholding would become important in a larger study. First, FADE and Crevasse would need to be tuned. Then there would also have to be a definition of what is "near" enough to a conserved feature to cluster, and how to handle boundary pockets.

Sifting through the computational results across 400 structures would also require some automation. Even with only ten structures, the pockets did not cluster perfectly. Inserts, small differences in the sequence, and differences in side chain conformations produce some noise in the results. The study has examples of pockets split into two parts that fell in the same computational cluster, and one interesting pocket that only appeared when the threshold length for defining a pocket was changed. K-means clustering algorithms are not stable on noisy data, so the clustering would have to be run multiple times or with a different algorithm. Pairwise comparisons with a canonical structure would be one way to reveal similarities and differences. It is also possible to construct a tree of pocket distances based on the clustering that could be compared to the sequence similarity maps.

Another solution for comparing pocket centroids across a large set of proteins would be to find the sets of pockets with the largest common pointset (LCP). Since the structures are not identical, what is actually needed is the largest common pointset within some acceptable degree of tolerance. The LCP problem is NP-hard, but there are fast approaches designed for structural biology problems. Choi and Goyal (2006) described one approach called tolerant hashing or T-hashing. T-hashing is a geometric hashing method that allows a predetermined distance between matched points. Because it is based on geometric hashing, the T-hashing algorithm finds sets of points independent of geometric translation and rotation. The output is a series of 3D transformations that overlay the input sets to align the most points. Depending on the size of the geometric invariants relative to the point set, the alignment can be either local or global. I have written software that implements a T-hashing algorithm and tested it on structural alignments. I am able to align subdomanins of proteins where there is large interdomain movement by finding sets of common points between the structures. Similarly, it would be possible to find sets of pocket centroids around common subdomains in a given protein. The advantage to using T-hashing to match sets of pockets is that no structural alignment of the proteins is required. The limitation is that the T-hashing algorithm works on pairs of point sets, rather than clustering like K-means.

There is data from the current computation that could be of use to compare pockets across more proteins. The FADE grid points do not fill in the pockets, so they do not provide a direct measure of volume. However, the number of grid points in a

pocket is a reasonable metric for the surface area of a pocket. Crevasse also outputs an object-oriented bounding box for each of the pockets. For pockets that are not irregularly shaped, the volume of the box provides a measure of the size of the pocket. It may also be possible to use the magnitude of the raw FADE scores to compare the geometry of different pockets. A deep, narrow pocket should have a different distribution of FADE scores than a broad, wide one. Currently, the FADE scores are used only to choose points to input into Crevasse. Metrics to consider would be the maximum score, the mean score, and the standard deviation of the scores across the pocket.

The computation can be viewed as a proof of concept for segmenting output from a grid-based computational method. FADE can integrate any function that can be mapped onto a grid. For example, rather than mapping atomic centers to the grid, the computation could be run with an electrostatic potential grid. The integration limits are also flexible. FADE currently integrates the grid with respect to a discrete ball, but another function such as a Gaussian could be used.

Finally, another interesting approach would be to map protrusions instead of pockets. FADE is unique among "pocket-finding" programs because it computes a score that is sensitive to the local surface curvature. By taking a range of low FADE scores instead of high scores, FADE and Crevasse could map protrusions on protein surfaces for clustering. Pockets are more predictive for small molecule binding sites, but protrusions often complement pockets in protein-protein interactions.

Choi V, Goyal N (2006) An Efficient Approximation Algorithm for Point Pattern Matching Under Noise. arXiv:cs/0506019v4

Appendix A FADE Changelog

Elaine Thompson Changes

- Fixed algebraic grid indexing error that sometimes returned incorrect z coordinates for the output routines.
- Set up bitmask to store user-requested output types, replacing if/then tests.
- Rewrote output routines to write all files simultaneously on one pass through the grid. This fixed a bug where if the user requested a file with the full grid, grid points that should be null were set to zeros in memory and incorrectly written to other files.
- Changed PDB output of FADE to conform to PDB2 standards. The PDB file now contains “residues” of 100 sequentially numbered hydrogen atoms.
- Added grid information to .fad output files for Crevasse.

Mike Pique Changes

Memory management improvements

- Improved error message on memory allocation failure
- Implemented safe (checking) alloc and malloc, adapted from DOT "dotmem.c". Replaced all malloc/calloc calls with malloc_t/calloc_t to check for allocation failures.
- Corrected freeing of computation grid memory and improved allocation of grids, moved messages to proper places
- Allocated "Count" array big enough, simplified firstPos logic and made action agree with comments.
- Made sure all variables properly initialize in error conditions

Speed improvements

- Replaced three calls to fround() with simple macro iround, for speed.
- Made xyzSize "double" to simplify code, removed some unnecessary runtime calls to "floor()" in "int" assignments
- Removed a "sqrt()" call for trivial speedup.

I/O Improvements

- Made log messages to to stdout, error messages to stderr.
- Added fflush so convolution integral progress report appears in a timely way.
- Function writeFiles: moved 'indx' call slightly inside loop for speed, inserted white space into all writeFiles output fields.
- In 'indx' function, replaced floats with doubles to reduce precision problems noted on big grids, where the Z value could be incorrectly calculated.
- In writeFiles, added checking that indices (a[0], a[1], a[2]) are correctly computed, rearranged "if goodPoint" logic for speedup and increased clarity.

Appendix B Crevasse Source Code

Makefile

```
CXX        =        g++
CXXFLAGS =          -Wall -g3

OBJS       = Crevasse.o CrevasseMap.o Grid.o

INCLUDES = -I/scratch/slocal/usr/local/include

all: crevasse

crevasse: $(OBJS)
        g++ -o $@ $(OBJS)

.cpp.o:
        $(CXX) $(CXXFLAGS) $(INCLUDES) -c $*.cpp -o $@


clean:
        /bin/rm *.o
```

Crevass.cpp

```
#include "Grid.h"
#include "CrevasseMap.h"

using namespace crevasse;

int main(int argc, const char *argv[]) {
        if (argc < 2) {
                // no filename, same message but abnormal termination
                cerr << "Error: Missing input filename" << endl;
                cerr << "Usage crevasse filename, optional parameters" << endl;
                cerr << "Run crevasse -h to see options" << endl;
                return 1;
        }

        string help = argv[1];
        // see if user typed -h or --help
        if (help.find("-h") < help.npos ) {
                cerr << "Usage is crevasse filename" << endl;
                cerr << "Add \"parameter space value\" pairs to customize."<< endl;
                cerr << "The following options are recognized..." << endl << endl;
                cerr << "-o output base filename (extension will be added) " << endl;
                cerr << "-n neighbors to keep a point, defaults to 2" << endl;
                cerr << "-s size of clusters to keep, defaults to 80" << endl;
                cerr << "-l minimum length of longest box axis, defaults to 0" << endl;
                cerr << "-6point (no value for this parameter) overrides default " << endl;
                cerr << "behavior and does not explore diagonals" << endl;
                return 0;
        }

//  string filename = argv[1];
```

```
//		int neighbors = 2;  // set a default
//		if (argc > 2) {
//			neighbors = atoi(argv[2]);
//		}

		string filename;
		string outfile;
		bool haveOutfile = false;
		unsigned int neighbors = 2;
		unsigned int numToCrawl = 26;
		float minLength = 0;
		unsigned int clustSize = 80;

		for (int i = 1; i < argc; i++) {
			string theArg = argv[i];
			// read the size
			if (theArg.find("-s") < theArg.npos) {
				i++;
				clustSize = atoi(argv[i]);
				if (clustSize == 0) {
					cerr << "Error: Couldn't read size argument" << endl;
					return 1;
				}
			} else if (theArg.find("-n") < theArg.npos)  {
				i++;
				neighbors = atoi(argv[i]);
				if (neighbors == 0) {
					cerr << "Error: Couldn't read number of neighbors" << endl;
					return 1;
				}
			} else if (theArg.find("-l") < theArg.npos)  {
				i++;
				minLength = atof(argv[i]);
				if (minLength == 0) {
					cerr << "Error: Couldn't read length of box axis" << endl;
					return 1;
				}
			} else if (theArg.find("-o") < theArg.npos) {
				i++;
				outfile = argv[i];
				haveOutfile = true;
			} else if (theArg.find("-6point") < theArg.npos) {
				numToCrawl = 6;
			} else {
			// not a parameter so it's the required filename
				filename = theArg;
			}
		}

		if (haveOutfile == false) {
			outfile = filename;
		}

		cout << "Running clustering on " << filename << endl;
		cout << "Neighbors = " << neighbors << endl;
		cout << "Cluster size = " << clustSize << endl;
		cout << "Minimum length for longest box axis = " << minLength << endl;
		cout << "Writing base file name " << outfile << endl << endl;
		cout << "Crawling " << numToCrawl << " neighboring points" << endl;

		Grid* ptrGrid = new Grid(neighbors);
		try {
			if (filename.find(".grd") < filename.npos) {
				ptrGrid->readFadeGrid(filename);
			} else {
				ptrGrid->readFade(filename);
```

```
                }
        } catch (std::exception) {
                cerr << "Couldn't find or open input file " << filename << endl;
                delete ptrGrid;
                return 1;
        }
        cerr << "Read input file." << endl;

        if (numToCrawl == 6) {
                ptrGrid->cluster6();
        } else {
                ptrGrid->cluster26();
        }
        cerr << "Completed depth-first search." << endl;
        CrevasseMap * ptrCrevasseMap = new CrevasseMap(clustSize);
        ptrGrid->fillMap(*ptrCrevasseMap);
        cerr << "Considering " << ptrCrevasseMap->getSize() << " points for clusters" << endl;
        ptrCrevasseMap->siftPoints();
        ptrCrevasseMap->calcOOB();
        cerr << "Writing pdb file " << outfile << ".pdb" << endl;
        ptrCrevasseMap->writePDB(outfile);
        // ptrCrevasseMap->writeBoxes(outfile, minLength);
        // ptrCrevasseMap->writeMap(outfile, ptrGrid->getSize(),
                // ptrGrid->getCenter(), ptrGrid->getSpacing());
        ptrCrevasseMap->writeSummary(outfile, ptrGrid->getSize(), ptrGrid->getSpacing(),
                filename, minLength, neighbors);
        ptrCrevasseMap->writeXyzq (outfile);

        delete ptrGrid;
        delete ptrCrevasseMap;
        cerr << "Finished, freeing memory " << endl << endl;
        return 0;
}
```

Grid.h header file

```
#ifndef GRID_H_
#define GRID_H_
#include <vector>
#include <string>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <math.h>
#include "CrevasseMap.h"


using namespace std;
namespace crevasse {

// Small data class for gridpoints.

class Gridpoint {
public:
        Gridpoint() {
                hasData_ = 0;
                flag_ = 0;
                fade_ = 0;
        }
        Gridpoint(const float & inFade) { //!< construct with fade value
```

```
                fade_ = inFade;
                hasData_ = 1;
                flag_ = -1;
        }
        Gridpoint(const int & inFlag) { //!< construct with flag value
                fade_ = 0;
                hasData_ = 1;
                flag_ = inFlag;
        }

        const int& hasData() const { return hasData_; }  //!< return hasData
        const int& flag() const { return flag_; } //!< return flag
        const float& fade() const { return fade_; } //!< return fade value
        const void flag (const int& inFlag) { flag_ = inFlag; } //!<set flag
        const void fade (const float & inFade) { //!< set fade value and update class
                fade_ = inFade;
                hasData_ = 1;
                flag_ = -1;
        }
        virtual ~Gridpoint() {}
private:
        int hasData_;
        int flag_;
        float fade_;

};

/*
Grid class.  Knows how fill itself from files, search itself, and describe itself
*/

class Grid
{
public:
        Grid (const int & neighbors);
        const void readFade ( const string& filename );
        const void readFadeGrid ( const string& filename );
        const void readCrev ( const string& filename );
        const void cluster26();
        const void cluster6();
        const void search26(const int & i, const int & j, const int & k, const unsigned long & clust);
        const void search6(const int & i, const int & j, const int & k, const unsigned long & clust);
        const unsigned long entry(const int & i, const int & j, const int & k);
        const void indx(const unsigned long & n, int &i, int &j, int &k);
        const void fillMap (CrevasseMap & inMap);
        // Accessors
        const vector<float> & getCenter() const  { return myCenter; }
        const vector<int> & getSize() const { return mySize; }
        const float & getSpacing() const { return mySpacing; }
        const Gridpoint & getPoint( const int & i, const int & j, const int & k) const {
                return myData[i][j][k]; }
        virtual ~Grid();
private:
        Array3D<Gridpoint> myData;
        vector<float> myCenter;
        vector<int> mySize;
        float mySpacing;
        int myNeighbors;  //neighbors specified by the user + 1
};

};

#endif /*GRID_H_*/
```

Grid.cpp C file

```
#include "Grid.h"

namespace crevasse {

Grid::Grid(const int & neighbors) {
	myNeighbors = neighbors;
}


// read in a fade grid, .fad.grd
const void Grid::readFadeGrid ( const string& filename ) {
	ifstream inFile(filename.c_str());
	if (!inFile.is_open()) {
		exception err;
		throw err;
		return;
	}
	float data;
	string line;
	long counter = 0;
	int i, j, k;
	while (!inFile.eof()) {
		line.erase();
		getline(inFile,line);

		// ignore blank lines and read the header
		if (line.empty()) {continue;}
		if (line.compare(0,1,"#")== 0) { // comment or header line

//  Sample fade header
//#  FADE output grid    1STC.fad.grd
//#
//#  Xsize = 80, Ysize = 80, Zsize = 96
//#  center = (8.000000, 21.000000, 25.000000)
//#  spacing = 1.000000
//#
					// handle the grid size field
					if (line.find("Xsize") < line.npos) {
						string temp;
						int comma;
						line.erase(0,3);
						// find the first comma
						for (int i=0; i<3; i++) {
							comma = line.find_first_of(",");
							temp = line.substr(8, (comma-8));
							mySize.push_back(atoi(temp.c_str()));
							//now chop off to one past the comma
							line.erase(0, comma+2);
						}
						//now set up the grid with the sizes we read in
						myData = TNT::Array3D<Gridpoint>(mySize.at(0), mySize.at(1), mySize.at(2));
					}

					// read in the centers
					if (line.find("center") < line.npos) {
						string temp;
						int right_paren;
						int comma;
						line.erase(0,13);
						// chop off the right parenthesis for safety
						right_paren = line.find(")");
						line.erase(right_paren, (line.length()-right_paren) );
```

```
                                        for (int i=0; i<3; i++) {
                                                comma = line.find_first_of(",");
                                                temp = line.substr(0, comma);
                                                myCenter.push_back(atof(temp.c_str()));
                                                //now chop off to one past the comma
                                                line.erase(0, comma+2);
                                        }
                                }
                                // get the grid spacing
                                if (line.find("spacing") < line.npos) {
                                        line.erase(0,13);
                                        mySpacing = (atof(line.c_str()));
                                }

                        continue;
                        }
                        // done reading header
                        istringstream input(line, istringstream::in);
                        input >> data;
                        if (data != 0) {
                                indx(counter, i, j, k);
                                // debug
                                //if (i > mySize[0]) {cerr << "Bad x index of " << i << endl; continue;}
                                //if (j > mySize[1]) {cerr << "Bad y index of " << j << endl; continue;}
                                //if (k > mySize[2]) {cerr << "Bad z index of " << k << endl; continue;}
                                myData[i][j][k] = Gridpoint(data);
                        }
                        counter++;
                }
                inFile.close();
}

// read a .fad file

const void Grid::readFade ( const string& filename ) {
                ifstream inFile(filename.c_str());
                if (!inFile.is_open()) {
                        exception err;
                        throw err;
                        return;
                }
                string line;
                int i, j, k;
                float x, y, z, dist, ex;
                while (!inFile.eof()) {
                        line.erase();
                        getline(inFile,line);

                        // ignore blank lines and read the header
                        if (line.empty()) {continue;}
                        if (line.compare(0,1,"#")== 0) { // comment or header line

//  Sample fade header
//#  FADE output grid    1STC.fad.grd
//#
//#  Xsize = 80, Ysize = 80, Zsize = 96
//#  center = (8.000000, 21.000000, 25.000000)
//#  spacing = 1.000000
//#
                                // handle the grid size field
                                if (line.find("Xsize") < line.npos) {
                                        string temp;
                                        int comma;
                                        line.erase(0,3);
                                        // find the first comma
                                        for (int i=0; i<3; i++) {
```

```
                                        comma = line.find_first_of(",");
                                        temp = line.substr(8, (comma-8));
                                        mySize.push_back(atoi(temp.c_str()));
                                        //now chop off to one past the comma
                                        line.erase(0, comma+2);
                                }
                                //now set up the grid with the sizes we read in
                                myData = TNT::Array3D<Gridpoint>(mySize.at(0), mySize.at(1), mySize.at(2));
                        }

                        // read in the centers
                        if (line.find("center") < line.npos) {
                                string temp;
                                int right_paren;
                                int comma;
                                line.erase(0,13);
                                // chop off the right parenthesis for safety
                                right_paren = line.find(")");
                                line.erase(right_paren, (line.length()-right_paren) );

                                for (int i=0; i<3; i++) {
                                        comma = line.find_first_of(",");
                                        temp = line.substr(0, comma);
                                        myCenter.push_back(atof(temp.c_str()));
                                        //now chop off to one past the comma
                                        line.erase(0, comma+2);
                                }
                        }
                        // get the grid spacing
                        if (line.find("spacing") < line.npos) {
                                line.erase(0,13);
                                mySpacing = (atof(line.c_str()));
                        }

                continue;
                }
                // done reading header
                istringstream input(line, istringstream::in);
                input >> x >> y >> z >> dist >> ex;

                i = (int)(((x - myCenter[0])/mySpacing) + mySize[0]/2);
                j = (int)(((y - myCenter[1])/mySpacing) + mySize[1]/2);
                k = (int)(((z - myCenter[2])/mySpacing) + mySize[2]/2);
                // debug
                if (i > mySize[0]) {cerr << "Bad x index of " << i << endl; continue;}
                if (j > mySize[1]) {cerr << "Bad y index of " << j << endl; continue;}
                if (k > mySize[2]) {cerr << "Bad z index of " << k << endl; continue;}
                myData[i][j][k] = Gridpoint(ex);
        }
        inFile.close();
}

// read a file from this software back in
const void Grid::readCrev ( const string& filename ) {
        ifstream inFile(filename.c_str());
        if (!inFile.is_open()) {
                exception err;
                throw err;
                return;
        }
        string line;
        int i, j, k;
        float x, y, z;
        int flag;
        while (!inFile.eof()) {
                line.erase();
```

```
                getline(inFile,line);

                // ignore blank lines and read the header
                if (line.empty()) {continue;}
                if (line.compare(0,1,"#")== 0) { // comment or header line

//  Sample fade header
//#  FADE output grid    1STC.fad.grd
//#
//#  Xsize = 80, Ysize = 80, Zsize = 96
//#  center = (8.000000, 21.000000, 25.000000)
//#  spacing = 1.000000
//#
                        // handle the grid size field
                        if (line.find("Xsize") < line.npos) {
                                string temp;
                                int comma;
                                line.erase(0,3);
                                // find the first comma
                                for (int i=0; i<3; i++) {
                                        comma = line.find_first_of(",");
                                        temp = line.substr(8, (comma-8));
                                        mySize.push_back(atoi(temp.c_str()));
                                        //now chop off to one past the comma
                                        line.erase(0, comma+2);
                                }
                                //now set up the grid with the sizes we read in
                                myData = TNT::Array3D<Gridpoint>(mySize.at(0), mySize.at(1), mySize.at(2));
                        }

                        // read in the centers
                        if (line.find("center") < line.npos) {
                                string temp;
                                int right_paren;
                                int comma;
                                line.erase(0,13);
                                // chop off the right parenthesis for safety
                                right_paren = line.find(")");
                                line.erase(right_paren, (line.length()-right_paren) );

                                for (int i=0; i<3; i++) {
                                        comma = line.find_first_of(",");
                                        temp = line.substr(0, comma);
                                        myCenter.push_back(atof(temp.c_str()));
                                        //now chop off to one past the comma
                                        line.erase(0, comma+2);
                                }
                        }
                        // get the grid spacing
                        if (line.find("spacing") < line.npos) {
                                line.erase(0,13);
                                mySpacing = (atof(line.c_str()));
                        }

                continue;
                }
                // done reading header
                istringstream input(line, istringstream::in);
                input >> x >> y >> z >> flag;

                i = (int)(((x - myCenter[0])/mySpacing) + mySize[0]/2);
                j = (int)(((y - myCenter[1])/mySpacing) + mySize[1]/2);
                k = (int)(((z - myCenter[2])/mySpacing) + mySize[2]/2);
                // debug
                if (i > mySize[0]) {cerr << "Bad x index of " << i << endl; continue;}
                if (j > mySize[1]) {cerr << "Bad y index of " << j << endl; continue;}
```

```
                        if (k > mySize[2]) {cerr << "Bad z index of " << k << endl; continue;}
                        myData[i][j][k] = Gridpoint(flag);
                }
                inFile.close();
}


const void Grid::cluster26() {
                int i, j, k;
                long clust;
                clust = 1;
                for (i=0; i<mySize[0]; i++) {
                        for (j=0; j<mySize[1]; j++) {
                                for (k=0; k<mySize[2]; k++){
                                        search26 (i,j,k,clust);
                                        clust++;
                                }
                        }
                }
}

// cluster on six points, rather than 26

const void Grid::cluster6() {
                int i, j, k;
                long clust;
                clust = 1;
                for (i=0; i<mySize[0]; i++) {
                        for (j=0; j<mySize[1]; j++) {
                                for (k=0; k<mySize[2]; k++){
                                        search6 (i,j,k,clust);
                                        clust++;
                                }
                        }
                }
}

// this routine searches all 26 neighbors

const void Grid::search26(const int & i, const int & j, const int & k, const unsigned long & clust) {
                Gridpoint aPoint, anotherPoint;
                aPoint = myData[i][j][k];
                // zero flag = empty, positive flag = searched
                // we only want to search points with flags of -1
                if (aPoint.flag() >= 0) {return;} // empty or already searched

                int x,y,z; // indexes for moving around the point of interest
                // we always count the search point, so start at -1 to give a zero result
                // if there are no neighbors
                int sumData = -1;

                // check to see whether we're in a cluster and have any neighbors
                for (x = (i-1); x < (i+2); x++) {
                        if (x < 0) {continue;}
                        if (x >= mySize[0]) {continue;}
                        for (y = (j-1); y < (j+2); y++) {
                                if (y < 0) {continue;}
                                if (y >= mySize[1]) {continue;}
                                for (z = (k-1); z < (k+2); z++) {
                                        if (z < 0) {continue;}
                                        if (z >= mySize[2]) {continue;}
                                        anotherPoint = myData[x][y][z];
                                        sumData += anotherPoint.hasData();
                                }
                        }
                }
```

```
		// check neighbors and set the flag before moving on - this is key!

//		if (sumData < myNeighbors) {
//			aPoint.flag(0);
//			myData[i][j][k] = aPoint;
//			return;
//		} else {
//			aPoint.flag(clust);
//			myData[i][j][k] = aPoint;
//		}

		// EET 21Dec06 Changed to add "hair" when neighbors is high

		if (sumData == 0) {
			// point has no connectivity and is a total dud
			aPoint.flag(0);
			myData[i][j][k] = aPoint;
			return;
		}
		if (sumData < myNeighbors) {
			// point has connectivity.  Add to cluster but do not explore.
			aPoint.flag(clust);
			myData[i][j][k] = aPoint;
			return;
		}
		// point has threshold connectivity.  Explore.
		aPoint.flag(clust);
		myData[i][j][k] = aPoint;


		// search neighbors recursively
		for (x = (i-1); x < (i+2); x++) {
			if (x < 0) {continue;}
			if (x >= mySize[0]) {continue;}
			for (y = (j-1); y < (j+2); y++) {
				if (y < 0) {continue;}
				if (y >= mySize[1]) {continue;}
				for (z = (k-1); z < (k+2); z++) {
					if (z < 0) {continue;}
					if (z >= mySize[2]) {continue;}
					// recursive call
					search26(x,y,z,clust);
				}
			}
		}
}

// search only nearest six neighbors
// add fuzz by including all connected points in the cluster

const void Grid::search6(const int & i, const int & j, const int & k, const unsigned long & clust) {
		Gridpoint aPoint, anotherPoint;
		aPoint = myData[i][j][k];
		int x = i;
		int y = j;
		int z = k;
		int sumData = 0;
		// zero flag = empty, positive flag = searched
		// we only want to search points with flags of -1
		if (aPoint.flag() >= 0) {return;} // empty or already searched

		// check to see whether we're in a cluster and have any neighbors
		// I can't think of a nice way to do this with loops so it's written out
		--x;
		if ( x >= 0) {
			anotherPoint = myData[x][y][z];
```

```
                sumData += anotherPoint.hasData();
}
x+=2;
if (x < mySize[0]) {
                anotherPoint = myData[x][y][z];
                sumData += anotherPoint.hasData();
}
x=i;
--y;
if (y >=0) {
                anotherPoint = myData[x][y][z];
                sumData += anotherPoint.hasData();
}
y+=2;
if (y < mySize[1]) {
                anotherPoint = myData[x][y][z];
                sumData += anotherPoint.hasData();
}
y=j;
--z;
if (z >=0) {
                anotherPoint = myData[x][y][z];
                sumData += anotherPoint.hasData();
}
z+=2;
if (z < mySize[2]) {
                anotherPoint = myData[x][y][z];
                sumData += anotherPoint.hasData();
}
z=k;

// check neighbors and set the flag before moving on - this is key!
if (sumData == 0) {
                // point has no connectivity and is a total dud
                aPoint.flag(0);
                myData[i][j][k] = aPoint;
                return;
}
if (sumData < myNeighbors) {
                // point has connectivity.  Add to cluster but do not explore.
                aPoint.flag(clust);
                myData[i][j][k] = aPoint;
                return;
}
// point has threshold connectivity.  Explore.
aPoint.flag(clust);
myData[i][j][k] = aPoint;

// search neighbors recursively
--x;
if ( x >= 0) {
                search6(x,y,z,clust);
}
x+=2;
if (x <= mySize[0]) {
                search6(x,y,z,clust);
}
x=i;
--y;
if (y >=0) {
                search6(x,y,z,clust);
}
y+=2;
if (y <= mySize[1]) {
                search6(x,y,z,clust);
}
```

```
		y=j;
		--z;
		if (z >=0) {
			search6(x,y,z,clust);
		}
		z+=2;
		if (z <= mySize[2]) {
			search6(x,y,z,clust);
		}
		z=k;
}


const void Grid::fillMap (CrevasseMap & inMap) {
		int i, j, k;
		float x, y, z;
		Gridpoint aPoint;
		for (i=0; i<mySize[0]; i++) {
			for (j=0; j<mySize[1]; j++) {
				for (k=0; k<mySize[2]; k++){
					aPoint = myData[i][j][k];
					if (aPoint.flag() > 0) {
					// calculate coordinates
					// important:  Fade flips the signs of the grid center when it
					// writes the grid file header ADD rather than SUBTRACT the offsets
						x = mySpacing*((float) (i - mySize[0]/2)) + (float) myCenter[0];
						y = mySpacing*((float) (j - mySize[1]/2)) + (float) myCenter[1];
						z = mySpacing*((float) (k - mySize[2]/2)) + (float) myCenter[2];
						inMap.addPoint(aPoint.flag(), x, y, z, aPoint.fade());
					}
				}
			}
		}
}


/* ------------------------------------------------------
* Subroutine entry
* 3D index to 1D index
* Author: Julie C. Mitchell, mitchell@sdsc.edu
* Last revision: 10-24-06
*
* Notes:
* ------------------------------------------------------ */

const unsigned long Grid::entry(const int & i, const int & j, const int & k) {
		long longIndex;
		longIndex = (long) (mySize[2])*mySize[1]*i + (mySize[2])*j + k;
return longIndex;
}

/* ------------------------------------------------------
* Subroutine indx
* 1D index to 3D index
* Author: Julie C. Mitchell, mitchell@sdsc.edu
* Revised: Elaine E. Thompson, eethomp@sdsc.edu
* Last revision: 10-24-06
*
* ------------------------------------------------------ */

const void Grid::indx(const unsigned long & n, int &i, int &j, int &k){

		float  a, b, c;
		float  temp1,temp2;

		temp1	= (float) floor((float) (n / (mySize[2])));
```

```
		c			= (float) n - (float) (mySize[2])*temp1;

		temp2	= (float) floor(temp1 / (float) mySize[1]);
		b			= temp1 - (float) (mySize[1])*temp2;

		a			= (n - b * (float) (mySize[2]) - c) / (float) mySize[1] / (float) (mySize[2]);

//changed to floorf and added explicit casts
		i = (int)floorf(a);
		j = (int)floorf(b);
		k = (int)floorf(c);

return;
}

Grid::~Grid()
{
}

}


-----------------------------------------
CrevasseMap.h header file
-----------------------------------------

#ifndef CREVASSEMAP_H_
#define CREVASSEMAP_H_
#include <vector>
#include <map>
#include <iostream>
#include <fstream>
#include <tnt/tnt.h>
#include <tnt/jama_eig.h>

using namespace std;
using namespace TNT;
namespace crevasse {

class Point {
public:
		Point() {}
		//! constructor: from components
   Point( const float& x, const float& y, const float& z, const float& fade ) : x_(x), y_(y), z_(z), fade_(fade) {}
   const float& x() const { return x_; }  //!< return x component
   const float& y() const { return y_; }  //!< return y component
   const float& z() const { return z_; }  //!< return z component
   const float& fade() const {return fade_; } //!< return fade score
   const void x(const float& x) { x_ = x; }
   const void y(const float& y) { y_ = y; }
   const void z(const float& z) { z_ = z; }
   const void fade(const float& fade) { fade_ = fade; }
   const void xyz (const float& x, const float& y, const float& z, const float& fade) {
		x_ = x;
		y_ = y;
		z_ = z;
		fade_ = fade;
   }
		virtual ~Point() {}
private:
		float x_;
		float y_;
		float z_;
		float fade_;
};
```

```
class OOB {
public:
	OOB() {
		// set the size of the arrays upon instantiating the class
		covar = TNT::Array2D<double>(3,3,0.0);
		eVec = TNT::Array2D<double>(3,3,0.0);
		// eVal = TNT::Array1D<double>(3,0.0);
		means = TNT::Array1D<double>(3,0.0);
		extents = TNT::Array1D<double>(3,0.0);
		corners = TNT::Array2D<float>(8,3,0.0);
		volume = 0;
	}
	TNT::Array2D<double> covar;
	TNT::Array2D<double> eVec;
	// TNT::Array1D<double> eVal;
	TNT::Array1D<double> means;
	TNT::Array1D<double> extents;
	TNT::Array2D<float> corners;
	float volume;
	virtual ~OOB() {}
};


class CrevasseMap {
public:
	CrevasseMap(unsigned int & clusterSize) { myClusterSize = clusterSize; }
	const void addPoint(const unsigned long& index, const Point & inPoint);
	const void addPoint(const unsigned long& index, const float& x, const float& y,
		const float& z, const float& fade);
	const void calcOOB();
	const void siftPoints();
	const void writePDB(const string& filename);
	const void writeBoxes(const string & filename, const float & minLength);
	const void writeMap(const string& filename, const vector<int> & gridSize,
		const vector<float>& gridCenter, const float& gridSpacing, const float & minLength);
	const void writeSummary(const string& filename, const vector<int> & gridSize,
		const float& gridSpacing, const string & fadeFile, const float & minLength, const int & neighbors);
	const void writeXyzq(const string& filename);
	const int getSize() const {return myCrevasses.size();}
	virtual ~CrevasseMap() {}

private:
	multimap<const unsigned long, Point> myCrevasses;
	vector<OOB> myBoxes;
	unsigned int myClusterSize;
};

};

#endif /*CREVASSEMAP_H_*/
-----------------------------------------
CrevasseMap.cpp source code
-----------------------------------------

#include "CrevasseMap.h"

/*
 * From: http://www.gamedev.net/community/forums/topic.asp?topic_id=328157
 * Scale an OOB by finding the extremes in the new basis formed by the
 * orthogonal eigenvectors
box.extents.Zero();
for (int i = 0; i < points.size(); ++i)
{
   Vector3 diff = points[i] - box.center;
```

```
    for (int j = 0; j < 3; ++j)
    {
        float dot = fabs(diff.Dot(box.axis[j]));
        if (dot > box.extents[j])
            box.extents[j] = dot;
    }
}
*/

/*
 * http://www.gamedev.net/community/forums/topic.asp?topic_id=290417
 * C is the box center, A[] are the axes, and E[] are the extents.
corner[0] = C + E[0] * A[0] + E[1] * A[1] + E[2] * A[2];
corner[1] = C - E[0] * A[0] + E[1] * A[1] + E[2] * A[2];
corner[2] = C + E[0] * A[0] - E[1] * A[1] + E[2] * A[2];
corner[3] = C + E[0] * A[0] + E[1] * A[1] - E[2] * A[2];
corner[4] = C - E[0] * A[0] - E[1] * A[1] - E[2] * A[2];
corner[5] = C + E[0] * A[0] - E[1] * A[1] - E[2] * A[2];
corner[6] = C - E[0] * A[0] + E[1] * A[1] - E[2] * A[2];
corner[7] = C - E[0] * A[0] - E[1] * A[1] + E[2] * A[2];

where p[][] are the vertices.
p[0][0] = ev[0][0]*hl[0] + ev[1][0]*hl[1] + ev[2][0]*hl[2] + c[0];
p[0][1] = ev[0][1]*hl[0] + ev[1][1]*hl[1] + ev[2][1]*hl[2] + c[1];
p[0][2] = ev[0][2]*hl[0] + ev[1][2]*hl[1] + ev[2][2]*hl[2] + c[2];

*/



namespace crevasse {

const void CrevasseMap::addPoint(const unsigned long& index, const Point & inPoint) {
        myCrevasses.insert(pair<const int, Point>(index, inPoint));
}

const void CrevasseMap::addPoint(const unsigned long& index, const float& x,
        const float& y, const float& z, const float& fade) {
        Point aPoint(x, y, z, fade);
        myCrevasses.insert(pair<const int, Point>(index, aPoint));
}

const void CrevasseMap::siftPoints() {
        // see whether this is a waste of time
        if (myCrevasses.size() < myClusterSize) {
                cerr << "Not enough points to consider.  No clusters in this protein." << endl;
                myCrevasses.clear();
                return;
        }
        // OK, we may have a crevasse so sort the points and organize
        // keys in multimaps are always const so have to copy from one map to another

        multimap<const unsigned long, Point> tempMap; // new list of good points
    multimap<const unsigned long, Point>::const_iterator lower;
    multimap<const unsigned long, Point>::const_iterator upper;


        // get the first key
        lower = myCrevasses.begin();
        long currKey = lower->first;
        long counter = 1;

        do {
                lower = myCrevasses.lower_bound(currKey);
      upper = myCrevasses.upper_bound(currKey);
```

```
                        // check to see if there's enough and copy
                        if (myCrevasses.count(currKey) >= myClusterSize) {
                        while (lower != upper) {
                        tempMap.insert(pair<const unsigned long, Point>(counter, lower->second));
                        lower++;
                        }
                        counter++;
                        }

                        // now get the next key if we're not at the end of the map;
                        if (upper != myCrevasses.end()) {
                                currKey = upper->first;
                        }
                } while (upper != myCrevasses.end());

                // replace the old map with the new, smaller one
                cerr << "Finished sifting. Keeping " << counter-1 << " clusters." << endl;
                myCrevasses = tempMap;
}


const void CrevasseMap::calcOOB() {

                // make sure we have at least one data set to work on
                if (myCrevasses.size() == 0 ) {
                        return;
                }

   multimap<const unsigned long, Point>::const_iterator lower;
   multimap<const unsigned long, Point>::const_iterator upper;
                Point aPoint;
                // set the first key
                long currKey = 1;

                do {
                        OOB anOOB;
                        lower = myCrevasses.lower_bound(currKey);
      upper = myCrevasses.upper_bound(currKey);
      int nSize = myCrevasses.count(currKey);
      int i, j;
      // array of points
      Array2D<double> myData(nSize, 3);
      Array2D<double> myTranspose(3, nSize);

      // fill the array with the raw point data
      int counter = 0;
      while (lower != upper) {
                aPoint = lower->second;
                // copy the data into a TNT array
                myData[counter][0] = aPoint.x();
                myData[counter][1] = aPoint.y();
                myData[counter][2] = aPoint.z();

                // accumulate totals for means
                anOOB.means[0] += aPoint.x();
                anOOB.means[1] += aPoint.y();
                anOOB.means[2] += aPoint.z();
                lower++;
                counter++;
      }
      // calculate the means
      anOOB.means[0] /= nSize;
      anOOB.means[1] /= nSize;
      anOOB.means[2] /= nSize;

      // subtract means from the point data
```

```
for (i = 0; i < nSize; ++i) {
        for (j = 0; j < 3; ++j) {
                myData[i][j] -= anOOB.means[j];
        }
}
// transpose the points
for (i = 0; i < nSize; ++i) {
        for (j = 0; j < 3; ++j) {
                myTranspose[j][i] = myData[i][j];
        }
}
// calculate the covariance matrix, eigenvectors
anOOB.covar = matmult(myTranspose, myData);
JAMA::Eigenvalue<double> ev(anOOB.covar);
                // ev.getRealEigenvalues(anOOB.eVal);
                ev.getV(anOOB.eVec);

                // normalize the eigenvectors, remembering that they're in the COLUMNS
                // EET TODO Jama seems to normalize the eigenvectors in its output so
                // this step may be unnecessary.  BUT, if the vectors are ever not
                // normalized, the entire calculation breaks badly.

                for (i = 0; i < 3; ++i) {
                        double dist = 0;
                        dist += (anOOB.eVec[0][i] * anOOB.eVec[0][i]);
                        dist += (anOOB.eVec[1][i] * anOOB.eVec[1][i]);
                        dist += (anOOB.eVec[2][i] * anOOB.eVec[2][i]);
                        dist = sqrt(dist);
                        anOOB.eVec[0][i] /= dist;
                        anOOB.eVec[1][i] /= dist;
                        anOOB.eVec[2][i] /= dist;
                }

                // find the extents of the box
                for (i = 0; i < nSize; ++i) {
                for (j = 0; j < 3; ++j) {
                        // check the dot product of the data minus means
                        // against each axis in turn
                double dot = 0;
                dot += (myData[i][0] * anOOB.eVec[0][j]);
                dot += (myData[i][1] * anOOB.eVec[1][j]);
                dot += (myData[i][2] * anOOB.eVec[2][j]);
                dot = abs(dot);
                if (dot > anOOB.extents[j]) {
                anOOB.extents[j] = dot;
                }
                }
                }
                // calculate the box volume as it may come in handy
                anOOB.volume = (float)((anOOB.extents[0]*2) * (anOOB.extents[1]*2) *
                                                                        (anOOB.extents[2]*2));

                // calculate the eight box corners
                // calculate the extents * axes and put in TNT structures
                TNT::Array1D<double> xLength(3);
                TNT::Array1D<double> yLength(3);
                TNT::Array1D<double> zLength(3);
                TNT::Array1D<double> corner(3);

                for (i=0; i<3; ++i) {
                        xLength[i] = anOOB.eVec[i][0] * anOOB.extents[0];
                        yLength[i] = anOOB.eVec[i][1] * anOOB.extents[1];
                        zLength[i] = anOOB.eVec[i][2] * anOOB.extents[2];
                }

                // corners are in the order specified by SLIDE
```

```
                        corner = anOOB.means - xLength - yLength - zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[0][i] = corner[i];
                        }
                        corner = anOOB.means + xLength - yLength - zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[1][i] = corner[i];
                        }
                        corner = anOOB.means - xLength + yLength - zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[2][i] = corner[i];
                        }
                        corner = anOOB.means + xLength + yLength - zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[3][i] = corner[i];
                        }
                        corner = anOOB.means - xLength - yLength + zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[4][i] = corner[i];
                        }
                        corner = anOOB.means + xLength - yLength + zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[5][i] = corner[i];
                        }
                        corner = anOOB.means - xLength + yLength + zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[6][i] = corner[i];
                        }
                        corner = anOOB.means + xLength + yLength + zLength;
                        for (i=0; i<3; ++i) {
                                anOOB.corners[7][i] = corner[i];
                        }

        // store the info and move on
        myBoxes.push_back(anOOB);
                        currKey++;

                } while (upper != myCrevasses.end());
                // OK. Now we've calculated and stored the eigenvalues and vectors

}


/* ------------------------------------------------------------
* Subroutine writeBoxes
* Write coordinates, distances and exponents
* Author: Elaine Thompson, eethomp@sdsc.edu
* Last revision: 6Jun07
*
* ------------------------------------------------------------ */


const void CrevasseMap::writeBoxes(const string & filename, const float & minLength) {
                int counter = 1;
                OOB anOOB;
                vector<OOB>::const_iterator iter;
                for (iter = myBoxes.begin(); iter!= myBoxes.end(); iter++) {
                        anOOB = *iter;

                        // find the longest edge
                        double maxExtent = 0;
                        for (int i = 0; i < 3; ++i) {
                                if (maxExtent < anOOB.extents[i]*2) {
                                        maxExtent = (float) anOOB.extents[i]*2;
                                }
```

```
                        }
                        if (maxExtent < minLength) {
                                ++counter;
                                continue;}

                        // jump through hoops to convert the counter int to a string
                        ostringstream stream;
                stream << counter;

                // make the filename with an increment since we're writing more than one
                        string nameXYZ = filename + stream.str();
                        nameXYZ += ".xyz";
                        ofstream outFile(nameXYZ.c_str());
                        for (int i = 0; i < 8; ++i) {
                                outFile << anOOB.corners[i][0] << " " << anOOB.corners[i][1] << " ";
                                outFile << anOOB.corners[i][2] << endl;

                        }
                        outFile.close();
                        ++counter;
                }
}



/* -------------------------------------------------------
* Subroutine writePDB
* Write coordinates and boxes as S atoms
* Author: Elaine Thompson, eethomp@sdsc.edu
* Last revision: 13Nov06
*
* Notes: Legacy C code mixed in
* ------------------------------------------------------- */

const void CrevasseMap::writePDB(const string& filename) {
   int counter;
                string type = "H";
                FILE *outPDB;
   multimap<const unsigned long, Point>::const_iterator lower;
   multimap<const unsigned long, Point>::const_iterator upper;
   unsigned long currKey = 1;
   Point aPoint;
                OOB anOOB;
                /* name and open output files */
                string namePDB = filename + ".pdb";
                outPDB = fopen(namePDB.c_str(), "w");

                /* print header */
                fprintf(outPDB,"REMARK  CREVASSE output file    %s\nREMARK\n",namePDB.c_str());
                // make sure we have some data before entering the loop
                if (myCrevasses.size() == 0) {
                        fclose(outPDB);
                        return;
                }

                /* return points */
   counter = 0;
                do {
                        OOB anOOB;
                        lower = myCrevasses.lower_bound(currKey);
      upper = myCrevasses.upper_bound(currKey);
      // loop through the points in this crevasse
      while (lower !=upper) {
                                aPoint = lower->second;
                        // write pdb
                                fprintf(outPDB,
```

```
                                        "ATOM  %5.0f  %s%02d UNK  %4d    %8.3f%8.3f%8.3f  %3.2f %5.2f\n",
                                        (float) counter+1, type.c_str(), counter%100, (int)currKey,
                                        aPoint.x(), aPoint.y(), aPoint.z(), 1.0, 10.0);
                                ++counter;
                                lower++;
                                // print out corners as sulfurs
                                // one box to a cluster so this should be safe
                                // debug
        }
        // write the bounding box
//                      debug
//                      if ((currKey-1) >= myBoxes.size()) {
//                              cerr << "Trying to overrun vector of boxes; bailing out on pdb file print" << endl;
//                              fclose(outPDB);
//                              return;
//                      }

                        anOOB = myBoxes[currKey - 1];
                        fprintf(outPDB, "REMARK box number %4d volume is %8.3f\n", (int)currKey, anOOB.volume);
                        fprintf(outPDB,"REMARK EXTENTS: %7.2f %7.2f %7.2f\n",
                                anOOB.extents[0]*2, anOOB.extents[1]*2, anOOB.extents[2]*2);

// Uncomment this block to write corners as sulfur atoms
//                      for (int i = 0; i < 8; ++i) {
//                              fprintf(outPDB,
//                                      "ATOM  %5.0f  S%02d UNK  %4d    %8.3f%8.3f%8.3f  %3.2f %5.2f\n",
//                                      (float) counter+1, i, (int)currKey, anOOB.corners[i][0],
//                                      anOOB.corners[i][1], anOOB.corners[i][2], 1.0, 10.0);
//                              ++counter;
//                      }
                        currKey++;
                } while (upper != myCrevasses.end());

                /* close file */
                fclose(outPDB);
                return;
}

/* ------------------------------------------------------------
* Subroutine writeMap
* Write coordinates in grid format
* Author: Elaine Thompson, eethomp@sdsc.edu
* Last revision: 3Apr07
* ------------------------------------------------------------ */


//  Sample fade header
//#  Xsize = 80, Ysize = 80, Zsize = 96
//#  center = (8.000000, 21.000000, 25.000000)
//#  spacing = 1.000000
//#

const void CrevasseMap::writeMap(const string& filename,
                const vector<int> & gridSize, const vector<float>& gridCenter,
                const float& gridSpacing, const float & minLength) {
   multimap<const unsigned long, Point>::const_iterator lower;
   multimap<const unsigned long, Point>::const_iterator upper;
   unsigned long currKey = 1;
   Point aPoint;
   int counter = 0;

                /* name and open output files */
                string crevFile = filename + ".map";
                ofstream outFile(crevFile.c_str());
                string summaryFileName = filename + ".sum";
                ofstream summaryFile(summaryFileName.c_str());
```

```
        /* print header */
        outFile << "#  CREVASSE output file " << crevFile << endl;
        outFile << "#  Xsize = " << gridSize.at(0) <<
                        ", Ysize = " << gridSize.at(1) <<
                        ", Zsize = " << gridSize.at(2) << endl;
        outFile << "#  center = (" << gridCenter.at(0) << ", " <<
                        gridCenter.at(1) << ", " << gridCenter.at(2) << ")" << endl;
        outFile << "#  spacing = " << gridSpacing << endl;

        summaryFile << "#  CREVASSE summary file " << summaryFileName << endl;
        summaryFile << "# Cluster size threshold: " << myClusterSize << endl;
        summaryFile << "# Wrote boxes with minimum length " << minLength << endl;
        summaryFile << "# crevasseID points volume boxVolume sizeX sizeY sizeZ" << endl;

        // make sure we have some data before entering the loop
        if (myCrevasses.size() == 0) {
                outFile.close();
                summaryFile.close();
                return;
        }
        //find the volume of a gridpoint - hack to avoid fiddly pow funcion
        // and math libraries
        // EET 21May07  This calculation is totally bogus!
        float volume = gridSpacing * gridSpacing * gridSpacing;
        /* return points */
        do {
                counter = 0;
                lower = myCrevasses.lower_bound(currKey);
    upper = myCrevasses.upper_bound(currKey);
    // loop through the points in this crevasse
    while (lower !=upper) {
                        aPoint = lower->second;
                // write xyz file
                        outFile << aPoint.x() << " " <<  aPoint.y() << " " <<
                                        aPoint.z() << " " << (int)currKey << endl;
                        lower++;
                        ++counter;
    }
    // write summary file
    // EET 21May07 Needed more info in summary
    OOB anOOB = myBoxes[currKey - 1];
    summaryFile << currKey << " " << counter << " " << counter * volume << " ";
    summaryFile << anOOB.volume << " " << anOOB.extents[0]*2 << " ";
    summaryFile << anOOB.extents[1]*2 << " " << anOOB.extents[2]*2 << endl;
                ++currKey;
        } while (upper != myCrevasses.end());

        /* close file */
        outFile.close();
        summaryFile.close();
        return;
}


/* ------------------------------------------------------
* Subroutine writeSummary
* Write a summary of the boxes we found
* Author: Elaine Thompson, eethomp@sdsc.edu
* Last revision: 3Apr07
* ------------------------------------------------------ */

const void CrevasseMap::writeSummary(const string& filename, const vector<int> & gridSize,
        const float& gridSpacing, const string & fadeFile, const float & minLength, const int & neighbors) {
   multimap<const unsigned long, Point>::const_iterator lower;
   multimap<const unsigned long, Point>::const_iterator upper;
```

```
    unsigned long currKey = 1;
    Point aPoint;
    int counter = 0;

            /* name and open output files */
            string summaryFileName = filename + ".sum";
            ofstream summaryFile(summaryFileName.c_str());

            /* print header */
            summaryFile << "CREVASSE summary file " << summaryFileName << endl;
            summaryFile << "Ran clustering on " << fadeFile << endl;
            summaryFile << "Neighbors = " << neighbors << endl;
            summaryFile << "Cluster size = " << myClusterSize << endl;
            summaryFile << "Minimum length for longest box axis = " << minLength << endl;
            summaryFile << "Grid info: Xsize = " << gridSize.at(0) <<
                                    ", Ysize = " << gridSize.at(1) <<
                                    ", Zsize = " << gridSize.at(2) <<
                                    ", spacing = " << gridSpacing << endl;
            summaryFile << "Wrote boxes to base file name " << filename << endl;
            summaryFile << "crevasseID points volume boxVolume sizeX sizeY sizeZ centerX centerY centerZ" << endl;

            // make sure we have some data before entering the loop
            if (myCrevasses.size() == 0) {
                    summaryFile << "Found no crevasses large enough " << endl;
                    summaryFile.close();
                    return;
            }
            //find the volume of a gridpoint - hack to avoid fiddly pow funcion
            // and math libraries
            // EET 21May07  This calculation is totally bogus, because crevasses are hollow
            // still writing because it will break scripts that run over this file
            float volume = gridSpacing * gridSpacing * gridSpacing;

            // count points in each crevasse
            do {
                    counter = 0;
                    lower = myCrevasses.lower_bound(currKey);
        upper = myCrevasses.upper_bound(currKey);
        // loop through the points in this crevasse
        while (lower !=upper) {
                                    aPoint = lower->second;
                                    lower++;
                                    ++counter;
        }
        // EET write box info summary
        OOB anOOB = myBoxes[currKey - 1];
        summaryFile << currKey << " " << counter << " " << counter * volume << " ";
        summaryFile << anOOB.volume << " " << anOOB.extents[0]*2 << " ";
        summaryFile << anOOB.extents[1]*2 << " " << anOOB.extents[2]*2 << " ";
        // EET 24Jan08 add COM to output
                    summaryFile << anOOB.means[0] << " " << anOOB.means[1] << " " << anOOB.means[2] << endl;
                    ++currKey;
            } while (upper != myCrevasses.end());

            // close file
            summaryFile.close();
            return;
}

/* ------------------------------------------------------------
* Subroutine writeXyzq
* Write coordinates as an xzyq style file
* File will be x, y, z, fade, segment
* Author: Elaine Thompson, eethomp@sdsc.edu
* Last revision: 19Feb08
* ------------------------------------------------------------ */
```

```
const void CrevasseMap::writeXyzq(const string& filename) {
   multimap<const unsigned long, Point>::const_iterator lower;
   multimap<const unsigned long, Point>::const_iterator upper;
   unsigned long currKey = 1;
   Point aPoint;

            /* name and open output file */
            string crevFile = filename + ".xyzq";
            ofstream outFile(crevFile.c_str());

            /* print header */
            outFile << "# CREVASSE output file " << crevFile << endl;
            outFile << "# X Y Z fade_score segment" << endl;

            // make sure we have some data before entering the loop
            if (myCrevasses.size() == 0) {
                        outFile.close();
                        return;
            }

            /* return points */
            do {
                        lower = myCrevasses.lower_bound(currKey);
      upper = myCrevasses.upper_bound(currKey);
      // loop through the points in this crevasse
      while (lower !=upper) {
                                    aPoint = lower->second;
                        // write to file
                                    outFile << aPoint.x() << " " <<  aPoint.y() << " " <<
                                                aPoint.z() << " " << aPoint.fade() << " " << (int)currKey << endl;
                                    lower++;
      }
                        ++currKey;
            } while (upper != myCrevasses.end());

            /* close file */
            outFile.close();
            return;
}



}
```