

PROTEIN MECHANICS THROUGH X-RAY CRYSTALLOGRAPHY

APPROVED BY SUPERVISORY COMMITTEE

Rama Ranganathan, M.D., Ph.D.

Luke Rice, Ph.D.

Michael Rosen, Ph.D.

Hongtao Yu, Ph.D.

DEDICATION

For Susan.



PROTEIN MECHANICS THROUGH X-RAY CRYSTALLOGRAPHY

by

KRISTOPHER IAN WHITE

DISSERTATION / THESIS

Presented to the Faculty of the Graduate School of Biomedical Sciences

The University of Texas Southwestern Medical Center at Dallas

In Partial Fulfilment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

The University of Texas Southwestern Medical Center at Dallas

Dallas, Texas

May, 2016

Copyright

by

Kristopher Ian White, 2016

All Rights Reserved

PERTURBATION STUDIES OF THE PDZ DOMAIN FAMILY

Publication No. _____

Kristopher Ian White, Ph.D.

The University of Texas Southwestern Medical Center at Dallas, 2016

Supervising Professor: Rama Ranganathan, M.D., Ph.D.

Proteins are dynamic entities which often cycle through a variety of conformational states as they carry out their functions. Despite the success of biophysical methods in determining the physical structures of proteins—that is, the precise three-dimensional configuration of all of their constituent atoms—no comprehensive physical models exist which accurately describe or predict the conformational cycling of proteins. For such models to be built, comprehensive knowledge of the energetics of intramolecular interactions in model proteins is essential. Here, we present three approaches which begin to address this problem, each through a different form of perturbation to a series of members of the PDZ protein family. First, we show that cycles of mutagenesis coupled with X-ray crystallography can reveal an anisotropic, distributed pattern of physical interactions in a PDZ domain, PSD-95 PDZ3. This pattern is functionally important and deeply connected to the evolution of PDZ domains in general. Second, we present a new approach for identifying essential dynamical features of proteins from relatively conventional X-ray diffraction data. Through combined analysis of nine different PDZ domain diffraction data sets, we show that collective features can be extracted and averaged, yielding a consensus picture of dynamics in the PDZ domain family. Finally, we report the development of a novel pump-probe method for directly inducing and reading out motions in proteins through the combined use of strong electric fields and time-resolved X-ray crystallography. We show that the method can be used to drive functionally relevant motions in a PDZ domain, LNX2 PDZ2, and provide a foundation for future efforts designed to directly probe the energetic architecture of proteins.

Table of Contents

1	Proteins as soft machines.....	1
2	Protein energetics.....	2
2.1	Proteins fold into marginally stable, three-dimensional structures.....	2
2.2	The distribution of energetic interactions in proteins is non-trivial.....	4
3	Energetics are linked to function through protein dynamics	7
3.1	Myoglobin as a prototypical system for studying protein dynamics	7
3.2	The conformational dynamics of enzymes	9
3.3	First-principles models of protein dynamics.....	12
4	Observing the “energetic architecture” of proteins.....	13
4.1	Thermodynamic mutant cycles reveal energetic coupling between residues	13
4.2	An alternative approach	15
5	Conclusions.....	16
6	Works cited.....	19
1	Introduction.....	22
1.1	Evolutionary constraints on protein fold and function.....	22
1.2	Allostery mediates long-range interactions in proteins.....	23
1.3	Analysis of amino acid coevolution in the several protein families appears to reveal allosteric pathways	25
1.4	A PDZ-ligand interaction as a model system for studying the biophysical constraints on adaptation in proteins	25
2	Results	28
2.1	The G330T mutation is conditionally neutral	28
2.2	Mutational paths to new specificity in PDZ3.....	30
2.3	Structural basis for class-neutral binding.....	33
2.4	Spatial distribution of conditional neutrality	35
2.5	The protein sector as the origin of adaptive mutations	36
2.6	Measuring structural couplings with X-ray crystallography.....	37
2.7	The pattern of structural coupling is consistent with allosteric and evolutionary models.....	40

2.8	Saturation mutagenesis in the background of H372A supports conclusions from structural coupling	41
3	Conclusions	43
3.1	A structural model for protein adaptation	43
3.2	Implications for protein engineering	44
3.3	Origins of allostery	45
3.4	Structural couplings and the physical origins of the sector	46
4	Experimental procedures	47
4.1	Global analysis of PDZ ligand specificity	47
4.2	Construction of the ligand library	48
4.3	Expression and purification of PSD-95 PDZ3 proteins	49
4.4	Protein crystallization and X-ray data collection	50
4.5	X-ray data reduction, model building, and refinement	51
5	Figures	53
6	Tables	69
7	Works cited	75
1	Introduction	78
2	Results and discussion	83
2.1	Consistent patterns of conformational heterogeneity can be extracted from independent X-ray datasets collected from PSD-95 PDZ3 protein crystals	83
2.2	Collective fluctuations can be extracted from the PDZ3-CRIPT coupling data	86
2.3	Collective fluctuations and simple models for protein physics	88
2.4	Averaging of mutual information across homologous PDZ domains reveals patterns of conserved collective fluctuations	90
2.5	Conclusions and future directions	93
3	Materials and methods	96
3.1	Construct acquisition	96
3.2	Protein purification and crystal growth	96
3.3	Peptide synthesis	98
3.4	Protein crystallization	98
3.5	X-ray diffraction data collection	99

3.6	Data reduction	100
3.7	Model building	100
3.8	Ensemble refinement and construction of super-ensembles	101
3.9	Calculation of correlation coefficient matrices and PCA	102
3.10	Calculation of elastic network models	103
3.11	Calculation of residue-by-residue mutual information from super-ensembles	104
3.12	Identification of collective modes from individual super-ensembles and from consensus data for the PDZ family	107
3.13	Contact probability matrix calculation	109
4	Figures.....	111
5	Tables	136
6	Works cited.....	141
1	Introduction.....	145
1.1	The function and evolution of proteins are deeply connected to mechanics	145
1.2	Current biophysical methods cannot provide the data required to create adequate mechanical models of proteins	145
1.3	EFX: a new method for revealing protein mechanics in atomic detail.....	146
2	Results	147
2.1	Theoretical and practical considerations.....	147
2.2	Molecular dynamics simulations suggest that external electric field will induce significant structural changes	148
2.3	System design	149
2.4	Application of EFX to a PDZ domain	150
2.5	The biological relevance of EFX-induced motions	155
2.6	From structure to mechanics	156
3	Methods.....	159
3.1	Molecular dynamics simulation protocol.....	159
3.2	Electronics.....	161
3.3	Safety considerations	162
3.4	Electrode construction.....	162
3.5	Protein expression, purification, and crystallization.....	163
3.6	Crystal mounting.....	164

3.7	Data collection and reduction	164
3.8	Refinement (C2 OFF models).....	165
3.9	Internal difference maps.....	166
3.10	Refinement against extrapolated structure factors.....	168
3.11	Comparison to homologous PDZ domains.....	169
3.12	Statistics	170
4	Figures.....	172
5	Tables	188
6	Works cited.....	199
1	In summary	204
1.1	The response of the PDZ proteins to perturbation is complex.....	204
2	Future directions.....	205
2.1	Development of a mechanical model for conformational change in the PSD-95 PDZ3 mutant cycle.....	205
2.2	Towards a simple mechanical model from EFX.....	207
2.3	Comparison of mechanical models to patterns of coevolution.....	208
3	Wrapping up	211
4	Works cited.....	212
1	Code.....	213
1.1	delta_r.....	213
1.1.1	delta_r_analysis.py	213
1.2	multiconf_tools	226
1.2.1	prep_pdb.py.....	226
1.2.2	batch_ensemble_refine.py.....	232
1.2.3	submit_batch_ensemble_refine_slurm-frac.sh.....	238
1.2.4	MIxnyn.py	239
1.2.5	cmap.py	240
1.2.6	analyze_superensemble.py.....	251
1.2.7	create_mi_dict.py	293
1.2.8	SymNMF.py.....	304
1.2.9	analyze_ermi.py	307

1.2.10	multiconf_utilities.py	378
1.3	arduino_control_center_v2	388
1.3.1	arduino_control_center_v2.ino.....	388
1.3.2	arduino_control_center.py.....	396

Chapter I

Introduction

1 Proteins as soft machines

The precise nature of the emergence, propagation, and ultimate self-awareness of living matter is among the most beautiful and perplexing topics in science. Given the hierarchical nature of living things, first-principles efforts to study their properties must ultimately begin with the study of their most basic components—the heritable genetic material and the molecular machines that carry out its instructions. As these components are physical systems composed of atoms, the development of accurate, predictive physical models describing their complex interactions represents a critical first step in understanding living systems from the bottom up.

Here, we focus on the proteins, although mechanics of the genetic material is obviously of great importance as well. Proteins are polymers, typically a chain of some combination of 20 amino acids, and the function of a given protein depends upon the lowest energy states occupied by this chain and the energetics of the conformational landscape surrounding it. As it navigates this landscape in carrying out some function, such a protein might bring to mind the concept of the machine¹. This is certainly an appealing analogy—formally speaking, a machine can simply be defined as “...a combination of resistant bodies so arranged that by their means the mechanical forces of nature can be compelled to do work accompanied by certain determinate motions”². Consistent with this definition, proteins—particularly those with precise three-dimensional conformations, or folds—often appear to harness some energy source (derived from thermal agitation, ATP hydrolysis, etc.) to perform stereotyped action along a relatively low-dimensional coordinate in a repeatable fashion (that is, low-dimensional in the context of total degrees of

freedom of the molecule). Proteins can thus be cast as the machines of the cell, invoking the need for a theory for protein mechanics.

While this analogy has didactic value in making a very foreign world more familiar, is there any deeper value in such a comparison? Indeed, if the principles of protein mechanics parallel those of the macroscopic machines, a deep and remarkably successful intellectual framework can be applied in understanding the molecules of life. In the macroscopic world, one does not need to know the positions and velocities of all atoms in an engine to describe or build a functional vehicle—kinematic and thermodynamic rules reduce the complexity of the problem to something manageable. Such a framework would be of great value in the realm of protein physics, enabling greater understanding of extant proteins while providing a platform for protein engineering. Unfortunately, however, materials at the nanoscale can behave very differently than their macroscale counterparts, and the complex nature of proteins only blurs this picture. Before a general theory for protein mechanics can be established, we must better understand the raw material.

2 Protein energetics

2.1 Proteins fold into marginally stable, three-dimensional structures

The transcription and translation of a given gene ultimately lead to the stepwise formation of a nascent chain of amino acids at the ribosome. The emerging polypeptide then navigates a vast, hyper-dimensional energy landscape, ultimately converging on some relatively low-energy topology, or fold³. The nature of this search is an area of extensive research and is outside of the scope of this work, but the general properties of the folded, globular protein are of particular interest here.

What are these properties? The elaborate architecture of many proteins—that is, the precise location of each atom in three dimensions—arises from a large number of weak interactions⁴ and is established by a heterogeneously-distributed pattern of hydrophobic and hydrophilic amino acids⁵ upon hydrophobic collapse. This process, first proposed by Tanford⁶ and, here, restated by Chothia, as the process by which “... the gain in hydrophobic energy [upon folding], as a result of the reduction in the number of non-polar contacts with water, compensates for the loss of chain configurational entropy; and that the polar groups on the interior of the protein form hydrogen bonds”⁷. A number of biophysical methods—X-ray crystallography, nuclear magnetic resonance (NMR) spectroscopy, electron microscopy, etc.—yield information on the spatial distribution of atomic interactions following this process. Typically, the resulting structure is characterized by a series of secondary structure elements (e.g., α -helices, β -sheets) which coalesce to form a tertiary structure, or fold, characterized by a hydrophilic exterior and a well-packed hydrophobic core. The fold is maintained by a large number of weak, mostly non-covalent interactions such as van der Waals interactions, Coulombic interactions, and hydrogen bonds, with energies on the order of 1–5 kcal mol⁻¹ (8). Together, these enthalpic contributions must be sufficient to offset the substantial entropic penalty associated with increasing the order of the system upon folding.

The energetics of this process have been studied extensively. In bulk, observations of the enthalpy and entropy of folding suggest that the thermal properties of globular proteins are largely independent of three-dimensional structure or amino acid composition. In an insightful review, Robertson and Murphy compiled thermodynamic data for 63 globular proteins and found that the greatest predictor of folding enthalpy, ΔH , entropy ΔS , and heat capacity, ΔC_p , is simply the length of the polypeptide chain, and that the relationship appears to be linear⁹. More recent work by Ghosh and Dill builds upon this and provides a simple model for the estimating the thermodynamic

properties of globular proteins¹⁰. Drawing upon the data compiled by Robertson and Murphy and other sources, the authors explicitly model protein stability as a balance between chain entropy, which opposes protein folding, and the degree of amino acid burial (i.e., the solvation entropy), which drives it. Denaturant concentration and temperature are also taken into account. This model thus describes the folding free energy, ΔG , of an “ideal thermal protein” for which ΔH and ΔC_p increase linearly with protein chain length. An expanded model also accounts for the influence of ionic strength, pH, and spatial confinement on stability. In a subsequent paper, the authors go on to use this model to compute the average stability of the *E. coli* proteome at 37°C, noting that the distribution of ΔG values is heavy-tailed, with $\langle \Delta G \rangle = 7.1 \pm 4.3$ kcal mol⁻¹ and around 15% of the proteome being < 4 kcal mol⁻¹ stable (i.e., around one protein in a thousand is unfolded at 37°C)¹¹. While there are certainly caveats to such calculations, the message is simple—the average protein is marginally stable.

2.2 The distribution of energetic interactions in proteins is non-trivial

While the overall ΔG of folding may follow general trends across the proteome, the distribution of the energetic terms for any given protein—that is, the precise energies of all atomic interactions that sum to the ΔG of folding—are hardly well-understood. Why? While the atomic structure of a protein may illustrate the precise locations of all of its constituent atoms in three-dimensional space, and thus the types of interactions that contribute to its characteristic fold, the structure does not tell us the energetics of those interactions. Studies designed to probe the energetic architecture of proteins illustrate this, while generating important clues about the nature of interatomic interactions in proteins. The advent of effective tools for performing site-directed mutagenesis in the early 1980’s¹² effectively provided a platform from which to perform “protein engineering”—

that is, to iteratively tinker with the identity of amino acids at different positions in the protein, and to then try to rationalize their effects through structural and biochemical studies¹³⁻¹⁷.

While these efforts initially yielded a great deal of atomic detail in support of general models for protein stability and function, researchers soon realized that the biophysical effects of mutations were often hard to predict, and even harder to rationalize. Classic work in phage T4 lysozyme demonstrated—contrary to expectation at the time—that proteins are remarkably tolerant to mutation, with more than half of positions in the protein tolerating mutation as assessed by phage plaque formation¹⁸. Important structural studies by Eriksson et al. on some of the viable mutations subsequently demonstrated that even the most seemingly profound mutations—mutations resulting in the loss of hydrophobic atoms from the core of the protein—were accommodated by the native fold^{19,20}. While values of difference in folding free energy, $\Delta\Delta G$ observed for “cavity-creating” mutants correlated reasonably well with the size of the cavity introduced by mutation (the larger the cavity, the greater the destabilization), they were hardly predictable. In a review written following this work in 1993, Brian Matthews wrote that “[t]here is every reason to expect that it will be possible to rationalize the stabilities of mutant proteins from accurate knowledge of their structures”⁸. More recently, in 2010, he summarized nearly 20 years of progress, writing that “the relatively modest objective of accurately predicting the changes in stability and structure associated with... mutations within the core is yet to be met”²¹. To date, an entire bioinformatics subfield remains preoccupied with this goal²².

While Eriksson et al. were busy with their phage T4 lysozyme mutants, others were focused on understanding the role of amino acid identity in catalysis by seeking to convert the chemical specificity of trypsin to that of chymotrypsin. Despite having nearly superimposable atomic structures, each of these serine proteases cleaves peptides with different properties—chymotrypsin

cleaves sequences with large hydrophobic residues, while trypsin has specificity for sequences with lysine or arginine. Initially, Gráf et al. attempted to simply convert a key active site residue of trypsin to the equivalent amino acid in chymotrypsin (i.e., the mutation of trypsin aspartate 189 to serine), assuming that the key to the difference in specificity was the structurally “obvious” one²³. This strategy did not lead to a trypsin with specificity for hydrophobic residues; on the contrary, it produced a poor, generalist enzyme. Eventually, however, Hedstrom et al. managed to engineer a trypsin with chymotrypsin-like specificity²⁴. Importantly, this strategy involved mutating positions in seemingly unimportant loops of the protein, far removed from the active site. While the authors succeeded in their explicit goal, they were left to only speculate about the roles of the loops in contributing to specificity, noting that some unintuitive and long-range propagation of conformational fluctuation must be contributing to the change.

Several years later, studies of the extracellular domain of the human growth hormone (hGHbp) and site 1 of human growth hormone (hGH) began to suggest that the energetics of key interactions are distributed in a structurally non-obvious way. In a landmark study, Clackson and Wells individually mutated 32 residues on the part of the 1300 Å² surface of the receptor which becomes buried upon hGH binding and assessed the difference in binding free energy, $\Delta\Delta G$, between the mutant and the wild-type hGHbp²⁵. Interestingly, researchers found that less than half of the mutations caused a substantial loss of binding affinity, and that the greatest energetic penalties upon mutation were localized to only a handful of residues contributing to a small fraction of the total accessible surface area at the interface. Furthermore, the identities of the residues contributing the greatest to binding were not well-correlated with structural parameters for side chains such as buried surface area, number of van der Waals contacts, *B*-factors, or solvation. While this result was important from the perspective of understanding the energetics of binding interactions, the

observations can perhaps be understood more generally in terms of generic proteins and the interactions that contribute to their fold and function.

From the discussed studies and numerous others, we can draw two essential conclusions about folded proteins. First, proteins appear to be highly robust, with many mutations having no discernable effect on the ΔG of folding or function. Second, the pattern of interactions required to promote the formation of a productive inter- or intramolecular interface or chemical reaction appears to be distributed throughout the structure in a heterogeneous and non-local manner. As we will see, this has substantial implications with respect to the mechanical properties of proteins.

3 Energetics are linked to function through protein dynamics

As the energetics discussed are on the scale of thermal agitation (i.e., they are on the order of kT), it is natural to assume that proteins must be in constant motion at physiological temperature, and that thermal energy must drive many aspects of protein folding and function. While protein motions have long been hypothesized to contribute to protein function, the precise nature of these contributions remains a matter of ongoing study^{26,27}.

3.1 Myoglobin as a prototypical system for studying protein dynamics

The crystal structure of the oxygen transporter myoglobin—the first protein crystal structure ever solved—marked a beginning in the quest to understand the interplay between protein fold, dynamics, and function²⁸. While the protein was known to bind O₂ or CO, the structure revealed a heme active site buried far from solvent. Subsequent work showed binding was reversible and fast²⁹, invoking the need for a dynamic model for the protein which would permit access by the ligand to the free heme. While the number of degrees of freedom accessible to all of the atoms in

myoglobin is massive, seminal studies by Austin et al. demonstrated that only a handful of dynamical processes actually relate to the function of the protein as it binds and releases O₂ or CO^{30,31}. The latter of these studies, published in 1975, represents an essential step in understanding the effects of dynamics on protein function.

Building upon initial work by Gibson²⁹ and others, Austin et al. studied the binding of ligand—either O₂ or CO—to sperm whale myoglobin using a flash photolysis technique. After saturating myoglobin with ligand, a pulse of 590 nm laser light was used to initiate its dissociation from the heme group of the protein. Ligand-free myoglobin has an absorption band at 434 nm, while the absorption bands of liganded myoglobin are blue-shifted by around 10 nm; as a result, the fraction of ligand-free myoglobin can be followed as a function of time by monitoring sample absorbance near 434 nm. Using this system, Austin et al. studied ligand rebinding as a function of temperature (from 40–350 K in steps of 10 K), time (from $\sim 10^{-6}$ – 10^1 s), and dynamic range (i.e., the mobility of the protein in different solvents). Critically, in the temperature regime below 200 K, rebinding kinetics for both CO and O₂ were found to be non-exponential in time, described instead by complex power laws requiring the summation of multiple exponentials. Put another way, these measurements suggest that the overall rate of rebinding is the sum of a continuously distributed series of rates; this observation thus implies that the protein can occupy a range of conformational substates, each with a slightly different rate of rebinding. After increasing the temperature above 200 K, Austin et al. observed another interesting series of phenomena—the presence of three additional non-exponential kinetic phenomena, or “processes”, present over different temporal regimes. Together, the data argued for a hierarchy of conformational substates in myoglobin, separated by timescale. Each process, they argued, corresponded to a relatively large scale conformational change between two substates with a characteristic rate of conversion, necessary

to permit diffusion of the ligand into the protein. Within each state, a spectrum of nearly isoenergetic conformations, or microstates, contributed to the non-exponential rate of rebinding of a given substate.

So, what physical model could explain the kinetic data? Frauenfelder et al. attempted to this question with a structural study of the effects of temperature on the structure of myoglobin³². They hypothesized that the states observed kinetically in previous work might be directly observable in the structure of the protein, and thus analyzed atomic displacements as a function of data collection temperature using X-ray crystallography. While they observed displacements at critical residues surrounding the heme, the results did not provide discrete resolution of different substates—they could not visualize a path by which ligand might bind the heme. Only with the development of time-resolved X-ray diffraction methods would the structural basis for these phenomena begin to clarify, at least in the case of myoglobin^{33,34}. Briefly, the consensus picture is that fs dynamics dominate the local environment around and within the heme group; these ultrafast motions are then coupled to slower, global motions throughout the protein, consistent with models for vibrational energy relaxation in proteins³⁵. The interplay between the surface of the protein and the solvation shell is thought to be critical as well³⁶.

3.2 The conformational dynamics of enzymes

Myoglobin is a special case, however—most proteins lack a photoactivatable chromophore, and many of the most interesting ones are enzymes, that is, they catalyze chemical reactions. Another model system, dihydrofolate reductase (DHFR), has served as a model system for understanding the role of protein motions in catalysis for quite some time. DHFR catalyzes the reduction of dihydrofolate (DHF) to tetrahydrofolate (THF) in an NADPH-dependent manner³⁷.

The catalytic mechanism for *E. coli* DHFR has been known for quite some time, and it predicts the existence of five chemical intermediates over the course of the reaction³⁸. Substantial effort has thus gone into the structural characterization of each step of this reaction, with X-ray crystal structures available for each catalytic intermediate showing substantial conformational change along the cycle^{39 40}. While these structures represent static snapshots of the protein along the reaction coordinate, subsequent work linked them by NMR through analysis of Carr-Purcell-Meiboom-Gill (CPMG)-based R_2 relaxation dispersion experiments⁴¹. This key study demonstrated that, at each step of the catalytic cycle, DHFR sampled weakly-populated conformational states which were frequently related to the next step in catalysis, and that interconversion occurred on the ms time scale. Subsequent studies of mutants in key regions of the protein revealed that disrupting residues involved in these exchange processes could interfere with the motion of the domain during the functional cycle and thus impair catalysis⁴².

Around the same time, relaxation dispersion experiments suggested a similar story in the case of the prolyl *cis-trans* isomerase cyclophilin A (CypA), which catalyzes the isomerization of prolyl peptide bonds⁴³. Eisenmesser et al. found that a two-state network conformational exchange on the ms timescale dominated during catalysis, and that the equilibrium was shifted far towards one of the two substates in the absence of substrate. Unlike the case of DHFR, however, subsequent analysis of all available crystal structures of CypA in a variety of states failed to reveal any visible conformational heterogeneity which was consistent with the relaxation dispersion data. Subsequent work by Fraser et al. proved transformative⁴⁴. Initially thinking that the lack of evident substates in the CypA was the consequence of low-resolution diffraction data, the authors solved a high-resolution cryogenic structure of the protein. This also failed to produce visible evidence of alternative conformations consistent with the NMR data, but application of a method for detecting

weakly populated rotameric states in electron density, Ringer⁴⁵, suggested that some of the features—at least, the ones closest to the active site—were indeed present at very low occupancy, evident in electron density contoured below commonly-used thresholds (0.3–1.0 σ). Inspired by the studies on the quenching of conformational fluctuations upon cryocooling^{46,47}, Fraser et al. collected a CypA dataset at room temperature. Intriguingly, alternative conformations were identified not only at the active site, but also at distant sites. Based on this observation, the authors proposed a model wherein the dynamic network from the NMR data are explained by a coupled switching of rotameric states between two sterically compatible sets of conformations during turnover. To test this hypothesis, they mutated a key serine residue in the network—but over 14 Å from the active site—to a threonine, mimicking the two states observed in the electron density at once and locking a nearby rotamers into one of the two sets of conformations. Crystal structures of the mutant showed the network was indeed locked in a single state, although NMR experiments suggested a less dramatic, but still significant, shift in the two-state equilibrium relative to wild-type. This mutation was found to disrupt the catalytic efficiency of the enzyme, reducing it 300-fold overall, and specifically reducing the rate of the bidirectional isomerization step of the enzyme by around 70-fold.

Together, the studies of *E. coli* DHFR and CypA offer complementary models for the role of dynamics in catalysis. In the case of DHFR, loop motions dominate the observed motions near the active site, while, in the case of CypA, the dynamic network of residues appears to propagate largely through steric occlusion of rotameric states. Both studies suggest that an equilibrated ensemble of states is a more natural way of thinking about the catalytic cycles of their respective proteins (discussed at length by Benkovic et al.⁴⁸). Finally, both studies illustrate the fundamental importance of integrating information from multiple techniques in order to better understand

dynamic processes, and motivate technological development required to discover truly coupled motion from structural data.

3.3 First-principles models of protein dynamics

So, given the decades of research into the dynamical nature of proteins, can any computational method predict the specific conformational dynamics that appear so fundamental to protein function? Even in the as-of-yet unrealized limit where molecular dynamics force fields are accurate without constraint⁴⁹⁻⁵² and can be used to computationally sample the conformational space of a protein in accordance with its true distribution, one hopes that some relatively simple rules will be sufficient to describe the motions of proteins. Indeed, the studies discussed thus far in this section—the observation of only a few kinetic transitions in myoglobin, the presence of consistent conformational substates across DHFR catalysis, and the existence of a two-state process in CypA—all seem to suggest that the vast conformational hypersurface accessible to a protein can possibly be reduced to only a few relevant dimensions which describe the functionally-relevant collective motions of its atoms.

What type of approach might permit such a reduction? Early on, the molecular dynamics community turned to normal mode analysis to simplify the interpretation of all-atom simulations of proteins⁵³. While these models for protein dynamics could sometimes reproduce dynamical information consistent with experiment, the computational overhead was immense, particularly in the early days of all-atom simulation. This practical consideration largely motivated the development of a new class of dynamical models for proteins based on a simple Hookian potential, where atoms in the protein are conceptualized as nodes separated by springs⁵⁴. These so-called elastic network models (ENMs) require no molecular dynamics simulation and simplify

assumptions about the vibrational dynamics of proteins even further, using only a single-parameter potential to produce results on par with both full NMA of MD trajectories⁵⁵.

Although there are fairly dramatic limitations to ENMs—proteins are certainly not purely harmonic in their dynamics—the spirit of this approach is appealing as it simplifies the apparent complexity of protein motions in a dramatic fashion, reducing the complex energy landscape of the protein to a few simple harmonic potentials. More importantly, ENMs provide a testable quantitative model for the coupling between atoms in the protein, as pairwise couplings derived from ENMs can be directly compared to those determined through other analytical or experimental means.

4 Observing the “energetic architecture” of proteins

Understanding proteins thus begs for a means to directly measure the degree of energetic coupling between any pair of atoms in a given protein and to relate this somehow to their motions. What types of experiments could accomplish this? Ideally, one would experimentally map out the potential surface for every atom in the protein. While efforts along these lines will be discussed in Chapter IV, such experiments are only beginning to be possible. In the meantime, we will review alternative approaches to get at this information.

4.1 Thermodynamic mutant cycles reveal energetic coupling between residues

A thermodynamic framework to systematically probe interaction energies between amino acids using mutant cycles was developed and employed by some to this end, although in a somewhat limited sense^{56,57}. These thermodynamic mutant cycles provide a useful framework with which to conceptualize and study the degree of energetic coupling between sets of amino acids. In

the simplest case, such cycles can be used to study pairwise interaction between mutations at two positions in a protein. More formally, to measure the coupling energy between mutations a and b at positions i and j in a protein, one would measure some series of thermodynamic observables relative to wild type (any state quantity; here, free energy) for each individual mutant and the double mutant. Explicitly, three values of ΔG must be calculated relative to wild-type— ΔG_i^a , ΔG_j^b , and $\Delta G_{i,j}^{a,b}$. A simple expression then reports on the thermodynamic independence of the mutations:

$$\Delta\Delta G_{i,j}^{a,b} = \Delta G_{i,j}^{a,b} - (\Delta G_i^a + \Delta G_j^b)$$

In the limit that there is no coupling between the two mutations, the sum of the energetic effects of the individual mutations equals the energetic effects of both mutations relative to the wild-type protein, $\Delta\Delta G_{i,j}^{a,b}$ is zero, and the mutational effects are then said to be additive. Conversely, if the sum of the effects taken independently does not equal the effect of both mutations at the same time, $\Delta\Delta G_{i,j}^{a,b}$ is non-zero, and the effects are said to be non-additive.

While in principle this approach represents a powerful means to understand the distribution of interaction energies in a protein and their distribution throughout the physical structure, several key issues limit their applicability in the general sense. While useful for dissecting the roles of a handful of residues and their contribution to a specific phenomenon, thermodynamic mutant cycles rapidly become unwieldy when it comes to measuring pairwise interactions for all residues in the protein. For a single protein, measuring all pairwise interactions between two residues requires measurement of some property for four different proteins. To extend this pairwise measurement

to all residues in a protein with r amino acids, the total number of measurements, n , needed is given by

$$n = \frac{1}{2}(r^2 - r) + r + 1$$

The average protein in *E. coli* is 267 residues long⁵⁸, so mapping all pairwise couplings would thus require 35,779 independent measurements in the case of mutation to only one other residue (e.g., an alanine scan). Furthermore, this only applies for measuring second order interactions; to measure the thermodynamic coupling of order p , the number of measurements for each set scales as 2^p . As a relevant upper bound for p is currently the matter of ongoing research, the number of experiments one might need to perform in order to understand the energetic structure of a protein rapidly becomes unfeasible. Finally, it is worth pointing out that the interpretation of even a single cycle—that is, figuring out how the energy of $\Delta\Delta G$ is actually accounted for by the protein structure—remains non-trivial as well.

4.2 An alternative approach

Motivated by observations such as these, Lockless and Ranganathan proposed a method called statistical coupling analysis (SCA) for inferring the interaction energies between the amino acids in a protein using a multiple sequence alignment (MSA)⁵⁹. Statistical coupling analysis (SCA) measures the conservation-weighted degree of coëvolution between amino acids in the MSA for a given protein family. In this work, Lockless and Ranganathan hypothesized that the interactions between amino acids in a protein family must be encoded in some way in the statistics of evolution. As the evolution of a protein fold is the result of many cycles of mutation and selection for protein function, the assumption is that residues not linked to a given function will not coëvolve, while two residues which contribute to that same function will by necessity coëvolve. The identification

of coëvolving residues thus provides a direct link to the function of a protein family, and (although indirectly) to the physics of the interactions between amino acids. To test this hypothesis, they performed SCA on a family of small, modular peptide ligand-binding domains—the PDZ proteins. Remarkably, the authors found a functionally-relevant pattern of statistical coupling that, when mapped to the structure of a representative PDZ domain, encompasses a spatially-contiguous group of amino acids distributed throughout the protein structure, that is, a group of residues that form the ligand binding region and that proceeds out to several relatively distant surface sites.

Subsequent work by Ranganathan and colleagues expanded and refined the conceptual framework underlying the SCA approach. Critically, application of the SCA method to different proteins reiterated initial observations; coëvolving sets of amino acids were discovered in other families as well, with three key properties⁶⁰. First, they are distributed throughout the structure, linking different regions of the protein. Second, these networks are spatially contiguous, that is, they contact one another to form a physical subset of the protein which traverses multiple elements of secondary structure. Finally, they are sparse; they comprise only a limited subset of the protein, typically on the order of 20% of the total number of positions. Further work demonstrated their functional relevance, suggesting that functional properties of proteins are more likely to be embedded in these networks of amino acids, and that they appear to act cooperatively to carry out this function⁶¹⁻⁶³. Ultimately, they were given a name—sectors—and it was demonstrated that multiple sectors could be identified in a protein family corresponding to different aspects of function⁶⁴.

5 Conclusions

From the perspective of understanding the distribution of physical interactions in proteins, the sector hypothesis provides several key conjectures with respect to protein mechanics. While the studies discussed in §1.2–3 suggest that intramolecular interactions are heterogeneous in space and time for only a handful of proteins, the results of SCA of many protein families argue that heterogeneity is a general feature of proteins, and that protein families may thus have stereotyped sets of interactions critical to their functions. Furthermore, sectors appear to be cooperative units, and, when multiple sectors are found in a given protein, they appear to act independently. Physically, this brings to mind the concept of a limited number of essential, superposed mechanical modes—the idea that heterogeneous patterns of interactions might give rise to the mechanical operation of subsets of the protein along independent coordinates. This is also consistent with the apparent dimensionality of many molecular processes.

So, where to begin? Work from our lab and others suggests that the key to understanding protein physics is to establish the distribution of energy throughout the various atomic interactions of the protein as well as the relative fluctuations of those energies over some relevant functional coordinate. With this in mind, my graduate work has focused on better understanding the nature and energetics of amino acid interactions through three types of perturbations:

1. Mutagenic perturbation. Proteins evolve through a stepwise process of mutation—what biophysical properties characterize mutants favored by natural selection?
2. Thermal perturbation. Proteins typically exist at some energetic ground state, and deviate from that state in a manner proportional to the degree of thermal agitation they experience. What general properties characterize these fluctuations, and to what degree are they conserved over different proteins with similar folds and functions?

3. Direct perturbation. By applying an electric field and directly exerting forces of known magnitude and direction on the various charges distributed throughout a protein, can we directly observe its “energetic architecture”?

Together, these approaches share a common theme—to identify the collective features which reduce the apparent complexity of protein function.

6 Works cited

1. Alberts, B. The cell as a collection of protein machines: preparing the next generation of molecular biologists. *Cell* **92**, 291–294 (1998).
2. Reuleaux, F. The Kinematics of Machinery. (1876). at <https://books.google.com/books?id=uMiEAAAIAAJ>
3. Anfinsen, C. B. Principles that govern the folding of protein chains. *Science* **181**, 223–230 (1973).
4. Pauling, L., Campbell, D. H. & Pressman, D. The nature of the forces between antigen and antibody and of the precipitation reaction. *Physiol Rev* (1943).
5. Kauzmann, W. Some factors in the interpretation of protein denaturation. *Adv. Protein Chem.* **14**, 1–63 (1959).
6. Tanford, C. Contribution of Hydrophobic Interactions to the Stability of the Globular Conformation of Proteins. *J. Am. Chem. Soc.* **84**, 1892–1896 (1962).
7. Chothia, C. Structural invariants in protein folding. *Nature* **254**, 304–308 (1975).
8. Matthews, B. W. Structural and genetic analysis of protein stability. *Annu Rev Biochem* **62**, 139–160 (1993).
9. Robertson, A. D. & Murphy, K. P. Protein structure and the energetics of protein stability. *Chem. Rev.* (1997).
10. Ghosh, K. & Dill, K. A. Computing protein stabilities from their chain lengths. *Proc Natl Acad Sci USA* **106**, 10649–10654 (2009).
11. Ghosh, K. & Dill, K. Cellular Proteomes Have Broad Distributions of Protein Stability. *Biophys J* **99**, 3996–4002 (2010).
12. Smith, M. In Vitro Mutagenesis. *Annual Review of Genetics* **19**, 423–462 (1985).
13. Shih, H. H., Brady, J. & Karplus, M. Structure of proteins with single-site mutations: a minimum perturbation approach. *Proc Natl Acad Sci USA* **82**, 1697–1700 (1985).
14. Craik, C. S. *et al.* Redesigning trypsin: alteration of substrate specificity. *Science* **228**, 291–297 (1985).
15. Fersht, A. R. *et al.* Hydrogen bonding and biological specificity analysed by protein engineering. *Nature* **314**, 235–238 (1985).
16. Estell, D. A. *et al.* Probing steric and hydrophobic effects on enzyme-substrate interactions by protein engineering. *Science* **233**, 659–663 (1986).
17. Alber, T. *et al.* Contributions of hydrogen bonds of Thr 157 to the thermodynamic stability of phage T4 lysozyme. *Nature* **330**, 41–46 (1987).
18. Rennell, D., Bouvier, S. E., Hardy, L. W. & Poteete, A. R. Systematic mutation of bacteriophage T4 lysozyme. *J Mol Biol* **222**, 67–88 (1991).
19. Eriksson, A. E. *et al.* Response of a protein structure to cavity-creating mutations and its relation to the hydrophobic effect. *Science* **255**, 178–183 (1992).
20. Eriksson, A. E., Baase, W. A., Wozniak, J. A. & Matthews, B. W. A cavity-containing mutant of T4 lysozyme is stabilized by buried benzene. *Nature* **355**, 371–373 (1992).
21. Baase, W. A., Liu, L., Tronrud, D. E. & Matthews, B. W. Lessons from the lysozyme of phage T4. *Protein Sci* **19**, 631–641 (2010).
22. Potapov, V., Cohen, M. & Schreiber, G. Assessing computational methods for predicting protein stability upon mutation: good on average but not in the details. *Protein Eng Des Sel* **22**, 553–560 (2009).

23. Gráf, L. *et al.* Electrostatic complementarity within the substrate-binding pocket of trypsin. *Proc Natl Acad Sci USA* **85**, 4961–4965 (1988).
24. Hedstrom, L., Szilagyi, L. & Rutter, W. J. Converting trypsin to chymotrypsin: the role of surface loops. *Science* **255**, 1249–1253 (1992).
25. Clackson, T. & Wells, J. A. A hot spot of binding energy in a hormone-receptor interface. *Science* **267**, 383–386 (1995).
26. Frauenfelder, H., Sligar, S. G. & Wolynes, P. G. The energy landscapes and motions of proteins. *Science* **254**, 1598–1603 (1991).
27. Karplus, M. & Kuriyan, J. Molecular dynamics and protein function. *Proc Natl Acad Sci USA* **102**, 6679–6685 (2005).
28. Kendrew, J. C. *et al.* A three-dimensional model of the myoglobin molecule obtained by x-ray analysis. *Nature* **181**, 662–666 (1958).
29. Gibson, Q. H. An apparatus for flash photolysis and its application to the reactions of myoglobin with gases. *J. Physiol. (Lond.)* **134**, 112–122 (1956).
30. Austin, R. H. *et al.* Dynamics of carbon monoxide binding by heme proteins. *Science* **181**, 541–543 (1973).
31. Austin, R. H., Beeson, K. W., Eisenstein, L., Frauenfelder, H. & Gunsalus, I. C. Dynamics of ligand binding to myoglobin. *Biochemistry* **14**, 5355–5373 (1975).
32. Frauenfelder, H., Petsko, G. A. & Tsernoglou, D. Temperature-dependent X-ray diffraction as a probe of protein structural dynamics. *Nature* **280**, 558–563 (1979).
33. Schotte, F. *et al.* Watching a Protein as it Functions with 150-ps Time-Resolved X-ray Crystallography. *Science* **300**, 1944–1947 (2003).
34. Barends, T. R. M. *et al.* Direct observation of ultrafast collective motions in CO myoglobin upon ligand dissociation. *Science* **350**, 445–450 (2015).
35. Fujisaki, H. & Straub, J. E. Vibrational energy relaxation in proteins. *Proc Natl Acad Sci USA* **102**, 6726–6731 (2005).
36. Frauenfelder, H. *et al.* A unified model of protein dynamics. *Proc Natl Acad Sci USA* **106**, 5129–5134 (2009).
37. Schnell, J. R., Dyson, H. J. & Wright, P. E. Structure, Dynamics, and Catalytic Function of Dihydrofolate Reductase. *Annual Rev. Biophysics* **33**, 119–140 (2004).
38. Fierke, C. A., Johnson, K. A. & Benkovic, S. J. Construction and evaluation of the kinetic scheme associated with dihydrofolate reductase from *Escherichia coli*. *Biochemistry* **26**, 4085–4092 (1987).
39. Sawaya, M. R. & Kraut, J. Loop and Subdomain Movements in the Mechanism of *Escherichia coli* Dihydrofolate Reductase: Crystallographic Evidence^{†,‡}. *Biochemistry* **36**, 586–603 (1997).
40. Venkitakrishnan, R. P., Zaborowski, E. & McElheny, D. Conformational changes in the active site loops of dihydrofolate reductase during the catalytic cycle. *Biochemistry* (2005). doi:10.1021/bi0580052
41. Boehr, D. D., McElheny, D., Dyson, H. J. & Wright, P. E. The dynamic energy landscape of dihydrofolate reductase catalysis. *Science* **313**, 1638–1642 (2006).
42. Bhabha, G. *et al.* A Dynamic Knockout Reveals That Conformational Fluctuations Influence the Chemical Step of Enzyme Catalysis. *Science* **332**, 234–238 (2011).
43. Eisenmesser, E. Z. *et al.* Intrinsic dynamics of an enzyme underlies catalysis. *Nature* **438**, 117–121 (2005).
44. Fraser, J. S. *et al.* Hidden alternative structures of proline isomerase essential for catalysis.

- Nature* **462**, 669–U149 (2009).
45. Lang, P. T. *et al.* Automated electron-density sampling reveals widespread conformational polymorphism in proteins. *Protein Sci* **19**, 1420–1431 (2010).
 46. Rasmussen, B. F., Stock, A. M., Ringe, D. & Petsko, G. A. Crystalline ribonuclease A loses function below the dynamical transition at 220 K. *Nature* **357**, 423–424 (1992).
 47. Halle, B. Biomolecular cryocrystallography: structural changes during flash-cooling. *Proc Natl Acad Sci USA* **101**, 4793–4798 (2004).
 48. Benkovic, S. J., Hammes, G. G. & Hammes-Schiffer, S. Free-energy landscape of enzyme catalysis. *Biochemistry* **47**, 3317–3321 (2008).
 49. Piana, S., Lindorff-Larsen, K. & Shaw, D. E. How robust are protein folding simulations with respect to force field parameterization? *Biophys J* **100**, L47–9 (2011).
 50. Piana, S., Klepeis, J. L. & Shaw, D. E. Assessing the accuracy of physical models used in protein-folding simulations: quantitative evidence from long molecular dynamics simulations. *Curr Opin Struct Biol* **24**, 98–105 (2014).
 51. Vitalini, F., Mey, A. S. J. S., Noé, F. & Keller, B. G. Dynamic properties of force fields. *J Chem Phys* **142**, 084101 (2015).
 52. Pitera, J. W. & Chodera, J. D. On the Use of Experimental Observations to Bias Simulated Ensembles. *J Chem Theory Comput* **8**, 3445–3451 (2012).
 53. Brooks, B. & Karplus, M. Harmonic dynamics of proteins: normal modes and fluctuations in bovine pancreatic trypsin inhibitor. *Proc Natl Acad Sci USA* **80**, 6571–6575 (1983).
 54. Tirion, M. Large Amplitude Elastic Motions in Proteins from a Single-Parameter, Atomic Analysis. *Phys. Rev. Lett.* **77**, 1905–1908 (1996).
 55. Atilgan, A. R. *et al.* Anisotropy of fluctuation dynamics of proteins with an elastic network model. *Biophysj* **80**, 505–515 (2001).
 56. Carter, P. J., Winter, G., Wilkinson, A. J. & Fersht, A. R. The use of double mutants to detect structural changes in the active site of the tyrosyl-tRNA synthetase (*Bacillus stearothermophilus*). *Cell* **38**, 835–840 (1984).
 57. Horovitz, A. Double-mutant cycles: a powerful tool for analyzing protein structure and function. *Fold Des* **1**, R121–6 (1996).
 58. Brocchieri, L. & Karlin, S. Protein length in eukaryotic and prokaryotic proteomes. *Nucleic Acids Res.* **33**, 3390–3400 (2005).
 59. Lockless, S. W. & Ranganathan, R. Evolutionarily conserved pathways of energetic connectivity in protein families. *Science* **286**, 295–299 (1999).
 60. Süel, G. M., Lockless, S. W., Wall, M. A. & Ranganathan, R. Evolutionarily conserved networks of residues mediate allosteric communication in proteins. *Nat Struct Biol* **10**, 59–69 (2003).
 61. Hatley, M. E., Lockless, S. W., Gibson, S. K., Gilman, A. G. & Ranganathan, R. Allosteric determinants in guanine nucleotide-binding proteins. *Proc Natl Acad Sci USA* **100**, 14445–14450 (2003).
 62. Shulman, A. I., Larson, C., Mangelsdorf, D. J. & Ranganathan, R. Structural determinants of allosteric ligand activation in RXR heterodimers. *Cell* **116**, 417–429 (2004).
 63. Ferguson, A. D. *et al.* Signal transduction pathway of TonB-dependent transporters. *Proc Natl Acad Sci USA* **104**, 513–518 (2007).
 64. Halabi, N., Rivoire, O., Leibler, S. & Ranganathan, R. Protein sectors: evolutionary units of three-dimensional structure. *Cell* **138**, 774–786 (2009).

Chapter II

Perturbing with mutagenesis: Insights into protein evolution and allostery

1 Introduction

1.1 Evolutionary constraints on protein fold and function

Proteins display the capacity to fold, often into well-packed three dimensional structures, and to carry out biologically essential activities such as catalysis, signal transmission, and allosteric regulation. The amino acid sequence reflects the constraints arising from these basic properties, and considerable prior work has focused on understanding how sequence encodes folding and biochemical function¹⁻⁴. Indeed, studies mapping functional properties to the structures of proteins form the basis for most general concepts of how proteins work and why they are built the way they are. However, it has been appreciated for decades that there may be non-trivial pressures on proteins that come not just from the physics of folding and function, but also from the process of evolution itself⁵⁻⁹. For instance, proteins must be possible through the random, iterative, stepwise process of mutation and selection, and they must be capable of adaptive variation as conditions of fitness vary in the environment. These considerations may place unique and yet unknown “design” constraints on evolved proteins, a missing aspect of our current understanding. An example of such a constraint might be functional connectivity in adaptive paths—requirement that adaptation between fitness peaks proceed through intermediates that maintain function above a threshold of selection⁷.

Recent work has begun to elucidate general properties of functional adaptation that are likely critical for the evolution of biological systems^{6,8-13}. One such property is conditional neutrality, a

special case in which mutations have no significant effect in the existing genetic or environmental background but have a significant effect upon subsequent changes in either the genome or environment^{9,11-13}. Such variations are “cryptic” in the sense that they hide their effects on fitness until exposed in the right setting, and can therefore accumulate and pre-exist in populations as standing genetic variation¹⁴. Since conditionally neutral mutations arise without selection and only express their fitness advantages upon subsequent events, they are said to be pre-adaptive (or “exaptive”¹⁵), and represent a pool of variants that can facilitate the emergence of novel adaptive phenotypes. Indeed, conditional neutrality has been convincingly demonstrated to facilitate adaptation both theoretically¹² and experimentally¹¹, and conceptually, represents a key link between the two major driving forces for genetic variation in populations—neutral drift and selection¹³. Understanding the prevalence and structural principles of conditional neutrality in protein molecules represents a key step in linking biophysical variation at the molecular level to evolutionary viability.

1.2 Allostery mediates long-range interactions in proteins

The connection of these concepts to allosteric phenomena is also of interest. Allostery is often an essential feature of biological systems at the atomic scale. First envisioned in the context of two models to explain cooperativity in oxygen binding in hemoglobin^{16,17}, allostery has come to represent any case in which a protein exhibits some global conformational change in response to a local perturbation such as ligand binding—a mechanism for information propagation¹⁸. What physical models can account for this phenomenon?

In one limit, allostery can be simply viewed as the conformational switching in a frustrated physical system, the product of simply propagating the resolution of steric conflict. Evidence for

such processes certainly exist; the work of Fraser et al., for example, reveals a dynamic conformational network with two discrete conformations evident in X-ray data of the peptidyl-prolyl *cis-trans* isomerase cyclophilin A¹⁹. As discussed in Chapter I, disruption of this network far from the active site substantially decreases the rate of enzyme catalysis. The allosteric pathway is thus inferred from the X-ray data and biochemical analysis.

In the other limit, however, conformational change is not a necessity; rather, the behavior of the system is grounded in subtler phenomena. This is the essence of a proposition by Cooper and Dryden, which suggests that some allosteric effects will propagate without visible conformational change²⁰. The essence of the argument is simple; the authors point out that while the mean positions of atoms in a protein may remain fixed upon ligand binding, the frequencies and amplitudes of the thermal fluctuations of atoms could change. This model was consistent with the general observation that the heat capacity of a given protein bound to its ligand was often observed to be less than that of the free protein, implying a reduction in thermal energy through the damping of atomic vibrations. As such, allostery in this limit would primarily be an entropic effect. While this proposal is challenging to test, recent work suggests that it is a plausible model for allosteric phenomena²¹. In this study, Frederick and colleagues explored the internal dynamics of calmodulin (CaM) when bound to one of six different peptide ligands. While the affinities of each peptide for CaM were found to be roughly equivalent, the thermodynamic parameters defining the free energy of binding were dramatically different. Intriguingly, the authors found that the estimated entropy of each CaM-ligand system scaled relatively linearly with the entropic component of the free energy of binding. While not a direct demonstration of Cooper and Dryden's model for allostery, this observation suggests that ligand binding does indeed modulate the entropy of the system as a whole.

1.3 Analysis of amino acid coevolution in the several protein families appears to reveal allosteric pathways

In either case, the direct and unambiguous experimental observation of conformational propagation or changes in conformational dynamics remains elusive, and will likely remain so until the development of general, time-resolved biophysical techniques to probe the energetics of proteins structures with atomic resolution (e.g., EFX; see Chapter IV). In the absence of such methods, Ranganathan and colleagues have developed a method, termed statistical coupling analysis (SCA), to infer the the relative pattern of interaction energies between amino acids in a protein family^{2,22}. As discussed in §1.4.2, SCA reveals one or more sparse, distributed, and spatially contiguous networks—or sectors—of coevolving amino acids in a protein family. These sectors encompass groups of residues which are functionally coupled, and, in several cases, they have been found to link distant functional sites in a manner which invokes the concept of an allosteric network^{4,22-25}. This is certainly reasonable—while sectors are statistical entities, the evolutionary coupling between their amino acids must be grounded in physical laws. In light of this and the physical models for allostery discussed in the previous section, it is thus sensible to expect some degree of correspondence between functionally-relevant allosteric networks and sectors. So, given that we cannot directly observe allosteric networks in real time, how can we at least assess the degree of mechanical coupling between amino acids? Pairwise and higher-order mutagenesis combined with structural analysis offers a strategy to “see” the mechanical basis for the collective network that, when averaged over the protein family, yields the sector.

1.4 A PDZ-ligand interaction as a model system for studying the biophysical constraints on adaptation in proteins

To explore these questions, we turned our attention to a model system for studying allostery in proteins. PDZ (Post-synaptic density, Discs large, Zona occludens) domains are small (~90 amino acids), modular protein-protein interaction domains which, with exception²⁶, bind the C-termini of other proteins^{27,28}. Here, we focus on the interaction between Post Synaptic Density protein 95 (PSD-95) PDZ3 and its cognate peptide ligand, derived from the C-terminus of Cysteine-Rich Interactor of PDZ Three (CRIPT; Ac-TKNYKQTSV-COOH)²⁹. The core PDZ3 domain is not particularly remarkable; like other PDZ domains, its architecture consists of a six-strand β -sandwich decorated with a pair of α helices (Figure 1 A). The CRIPT ligand binds to PDZ3 in a cleft between the β_2 strand and the α_2 helix of the domain by β -sheet augmentation, with the carboxy terminus forming a series of hydrogen bonds with a conserved region of the protein at the end of the so-called carboxylate binding loop (positions 322–325). The majority of the binding energy arises from this interaction and from interaction between the -2 position of the ligand (in the case of CRIPT, a threonine) and the residues it contacts on the α_2 helix. While not entirely accurate in general²⁸, one may conceptually group PDZ domains into two classes based on differences in ligand binding; class I domains typically bind ligands with sequence -X-T/S-X- Φ (where Φ is a hydrophobic residue, T is threonine, S is serine, and X is any residue), whereas class II domains bind ligands with sequence -X- Φ -X- Φ . In the case of PDZ3, the domain achieves class I specificity with a histidine at position 372; this amino acid is large and polar, making it ideal for interacting with T₋₂ of CRIPT (Figure 1 B).

In a previous study, McLaughlin et al. performed saturation mutagenesis on PDZ3 and assayed the function of each mutant for binding each of the two classes of peptides, CRIPT ligand and a mutant form with a bulky hydrophobic residue at the -2 position, T₋₂F³⁰. While PDZ3 binds the native CRIPT ligand with a K_D of $0.8 \pm 0.1 \mu\text{M}$, it only binds T₋₂F with a K_D of $36.0 \pm 2.1 \mu\text{M}$ (a

45-fold difference). In light of this, the authors examined the mutagenesis data for possible evolutionary paths to switch the specificity of the domain. To accomplish this, they first made the single mutation to PDZ3 found to confer the greatest affinity increase for T₂F, G330T (Figure 1 B). Despite being located relatively far from -2 position of the ligand, this mutation increased specificity of the domain for T₂F, to an affinity of $1.8 \pm .3 \mu\text{M}$. At the same time, the affinity for the wild-type ligand was largely preserved, with an affinity of $1.9 \pm 0.3 \mu\text{M}$. The G330T mutant was thus found to be a generalist—a protein capable of functioning in the context of either class I or class II constraints. To switch the specificity, the authors then created a double-mutant, G330T-H372A, as the H372A mutation was found to increase the specificity for T₂F as well. This switch essentially happens at the “active site” of the domain, and the rationale for the change in affinity is relatively obvious (Figure 1 B). The H372A mutation effectively truncates the sidechain to C_α, creating additional free space in the binding cleft around T₂ while eliminating an enthalpically-favorable hydrogen bonding interaction—an interaction thought to be key to recognition of S/T at -2³¹. This mutation is complementary in the case of binding to the T₂F peptide, however; the atoms removed upon mutation of the protein are effectively replaced with atoms on the ligand in the same space, and the lack of charge at either -2 or 372 promotes hydrophobic interaction. Importantly, changes correspond to a direct change to an interaction between the domain and the ligand involved in specificity. Subsequent analysis revealed that G330T-H372A does indeed have inverted substrate specificity relative to wild-type, with an affinity for the CRIPT of $22.1 \pm 2.6 \mu\text{M}$ and an affinity for T₂F of $0.5 \pm 0.1 \mu\text{M}$.

2 Results

2.1 The G330T mutation is conditionally neutral

While the effect on specificity resulting from the H372A mutation is easy to rationalize, the role of G330T is not. From the binding assay described above, it appears that the G330T mutant is a functional generalist, that is, it can bind two ligands without regard for amino acid identity at the -2 position. This is reminiscent of previous work on so-called conditionally neutral, or cryptic, mutations, which confer new function without affecting existing function in some context^{9,11,12,32}. Is the G330T mutation consistent with these types of mutations?

To test comprehensively test this, we assessed the degree of binding of each of four proteins—wild-type, G330T, H372A, and G330T-H372A—to a library of all possible peptide ligands with the final four positions randomized ($20^4 = 160,000$ total ligands). This assay was performed using a quantitative bacterial two-hybrid (BTH) assay for PDZ function, in which the transcription of a gene conferring antibiotic resistance is tuned to be linearly proportional to the binding free energy of PSD-95 PDZ3 to a set of ligands. The binding profile for a given PDZ domain can thus be assessed by growing bacteria carrying the BTH system in the presence of antibiotic and comparing the frequencies of alleles before and after selection. The degree to which a PDZ domain binds a given ligand x is proportional to the relative enrichment E_x , where $E_x = \log(f_x^s / f_x^u)$, where f_x^u is the frequency of observing the ligand in the unselected library and f_x^s is the frequency of observing it in the selected library. Of the 160,000 possible ligands, we observed 154,521 with sufficient counting statistics for all proteins. The full experimental details of the assay itself are summarized elsewhere^{30,33}.

What do the data reveal? Clustering analysis of some of the most representative peptides from across the four experiments reveals a pattern consistent with expectation (Figure 2 A); the wild-type protein largely binds class I ligands with T or S at -2, while H372A and G330T-H372A clearly bind ligands with a hydrophobic residue at -2. The binding profile of G330T is also consistent with expectation—ligand sequences consistent with either class are enriched. This analytical method ignores a large portion of the data, however. To more completely assess the binding profiles of each protein, we used principal components analysis (PCA) to visualize the data. For each PDZ domain variant, the 2,359 peptides with predicted affinity of 15 μ M or better for at least one domain variant were projected onto a two dimensional space (Figure 2 C). The assumed dimensionality is justified, as the first two principal components (V_1 , V_2) describe nearly 97% of the total variance in the data (Figure 2 B). From a biochemical perspective, these modes are highly meaningful. Analysis of the first component reveals the pattern hinted at by clustering—ligands cluster to the left of the origin are dominated by either T or S at the -2 position, while ligands to the right of the origin typically have F, Y, I, or L at the -2 position. The locations of CRIPT and T₂F are consistent with this. The second component then splits the ligands by identity at both the -2 and -3 positions.

So, how does each PDZ domain contribute to the projected pattern? This question can be addressed by projecting the data for each onto the same two-dimensional space with the same cutoff for binding (Figure 3). As expected, wild-type PDZ3 binds a subset of peptides (including CRIPT) in a manner consistent with its class I specificity (Figure 3 A). At the same time, both H372A and G330T-H372A bind ligands (including T₂F) in a manner consistent with class II specificity (Figure 3 C,D). This reiterates the idea that the H372A mutation is responsible for changing the class specificity of the domain from I to II. Most notably, however, the G330T protein spans these two spaces, consistent with the idea of it being a conditionally neutral mutation (Figure

3 C). Importantly, G330T does not bind all ligands—it only binds 854 with specificity better than 15 μM —but rather only a subset that spans the bottom region of the space.

2.2 Mutational paths to new specificity in PDZ3

We thus took this system as a framework for understanding the biophysical constraints on stepwise adaptation of proteins to new function. Formally, this series of mutations can be conceptualized as one of several along the edges of a cube, with each stepwise path from the wild-type/CRIPT vertex to the G330T-H372A T₂F vertex representing a possible sequence of mutations required to switch specificity (Figure 1 B). Consider the first mutation to occur in a hypothetical evolutionary trajectory; the two possible mutations are clearly different in the way they tune the specificity of the domain for each peptide. On one hand, the H372A mutation results in a PDZ domain with limited affinity for CRIPT but with nearly-native affinity for T₂F. On the other hand, the G330T mutant retains affinity for CRIPT while gaining affinity for T₂F. What can be said about the viability of each path, beginning with a mutation in the PDZ domain?

Lacking the means to explore this problem experimentally, we performed evolutionary dynamics simulations over a range of mutation rates and ligand-switching rates to better understand the conditions that might favor different paths of mutations, from wild-type to the double-mutant. Each simulation begins with a 1000-individual population with the wild-type PDZ3 genotype (Figure 4 A). At each generation, single mutations between genotypes are generated with a probability μ and double-mutations with probability μ^2 . Additionally, the target ligand switches between the class I CRIPT peptide and the class II T₂F peptide at a rate τ , and fitness at every generation is defined as proportional to the ligand fraction-bound as determined from the experimentally defined equilibrium dissociation constants (Figure 1 B). The total ligand

concentration is set to 10 μM , a value in the middle of the specificity range of wild-type PSD-95 PDZ3—the relevant regime for non-trivial dynamics. In essence, this simulation gives an opportunity to study how the flux between the two paths to the double-mutant state depends on both internal parameters (mutation rate and population size) and external parameters (environmental switching between the two class-distinct ligands).

A representative simulation trajectory at one particular mutation and ligand-switching rate illustrates properties of the adaptive process (Figure 4 B). In this case, $\tau = 500$ and $N\mu = 1$, meaning that the ligand switches every 500 generations and one single mutation is made, on average, at every generation. As expected, the wild-type genotype (black trace) is the most fit in the presence of the class I CRIPT ligand, with a small fraction of other genotypes stochastically occurring in the population according to the mutation rate and on their fitness relative to wild-type. Switching to the T₂F ligand causes the population to ultimately switch to the double mutant state (blue), the genotype that is most fit for the T₂F ligand under these simulation conditions. However, the path of switching can show considerable trial-by-trial variability with regard to intermediates. For example, in this trajectory, G330T (green) is more prevalent in trial 1 and H372A (red) more prevalent in trial 2. Averaged over many trials of switching (~ 500 events) from CRIPT to T₂F, we find that G330T is by far the preferred path of adaptation to the double mutant state given the selected mutation and ligand-switching rates (Figure 4 D).

How can we understand this result mechanistically? Since both G330T and H372A can bind the T₂F ligand about equally well (Fig. 1B), the path simply depends on the relative availability of these genotypes in the population at the moment of switching (Figure 4 B, insets). This property in turn depends on the fitness of G330T and H372A while in the CRIPT environment, a factor that heavily favors G330T over H372A (Fig. 1B, $K_D^{\text{G330T}} = 2.2 \pm 0.3 \mu\text{M}$ and $K_D^{\text{H372A}} = 26.9 \pm 6.3 \mu\text{M}$).

As a consequence, G330T typically comprises the majority of the cryptic genetic variation in the CRIPT environment, more likely to be present and able to support transition to the double mutant state when the environment switches to T₂F.

How does this result depend on mutation rate and the switching rate of target ligand? Simulations show that conversion to the double mutant state is only achieved over a certain regime of ligand-switching rate. This makes sense; if ligand switches too rapidly to permit fixation of the double-mutant, the population converges to the only genotype that is fit for the average of both ligand environments—G330T. However, in any regime of ligand-switching rate in which the double mutant fixes in the population, the simulations show that G330T is always more preferred than H372A in mediating adaptation (Figure 4 C–E). This is true when mutations are rare ($N\mu \ll 1$, Figure 4 C), and when mutations are abundant ($N\mu \gg 1$, Fig. 4E). Thus in general, it is the class-neutral genotype rather than the direct class-switching genotype that represents the likely intermediate in adaptation to new ligand class specificity.

But, how “neutral” does a mutation have to be in order to be statistically preferred over a class-switching mutant such as H372A in mediating adaptation? Indeed, even G330T is not strictly neutral in the CRIPT environment ($K_D^{\text{G330T}} = 2.2 \pm 0.3 \mu\text{M}$ and $K_D^{\text{WT}} = 0.8 \pm 0.1 \mu\text{M}$), the reason why it is considerably less competitive than wild-type (Figure 4 B). To study this, we carried out a series of simulations in which we examined the effect of varying the affinity of G330T for the CRIPT ligand from near wild-type (1 μM) to that of the class-switching mutant (26.9 μM , Figure 1 B). The data show that given the conditions of the simulation, affinities up to the limit of physiological PDZ binding ($< 15\mu\text{M}$) will be statistically preferred to H372A (Figure 4 F). This result relaxes the notion of conditional neutrality, defining a limit of protein function at which a

mutant can still contribute to the cryptic genetic variation and be distinguished in adaptive capacity from direct class-switching mutations.

2.3 Structural basis for class-neutral binding

The dominance of the class-neutral G330T mutation in adaptation to new ligand specificity is particularly interesting because it is not structurally obvious. Position 330 occurs on a surface loop (β 2- β 3) that lies behind the substrate binding pocket and makes no direct contact with ligand (Figure 1 A, C). How does mutation at this site create a dual-functional PDZ3 binding pocket capable of recognizing both class I and class II ligands? To address this issue, we solved high-resolution crystal structures of the PSD-95 PDZ3 variants (wild-type, G330T, H372A, and the double-mutant) in either the apo (ligand-free) state or bound to either CRIPT or T₂F ligands—a total of twelve structures (Figure 5 and Tables 2–5). All structures were solved under near-isomorphous conditions—in the same crystal form (P4₁32) with unit cell constants within 0.5% of each other—and models were refined to similarly high resolution (2.0 Å or better) with excellent statistics and geometry (Table 2). Thus, we are in a position to make statements about the mechanism of action of the mutations from comparative study of atomic structures.

The structure of PDZ3 bound to the class I CRIPT ligand—the wild-type complex—shows H372 in a rotameric state in which it can hydrogen bond with T₂, and a well-ordered β 2- β 3 loop that makes backbone hydrogen bonds with the H372 region (Figure 5 A, consistent with Doyle et al.³¹). Not surprisingly, mutation of position 372 to alanine abrogates class I recognition by elimination of the hydrogen-bonding partner for the Thr/Ser residue at the -2 position (Figure 5 B) while creating space to accommodate a bulky hydrophobic side chain at the ligand -2 position without steric clash (Figure 5 C). No other conformational changes of this magnitude are evident,

indicating that the direct class-switching phenotype in which effects of H372A is due to effects that are spatially localized to the site of adaptive challenge.

The origin of the class-neutral phenotype of G330T is qualitatively different and more complex. To explain, consider the effect of the T₂F ligand in binding to wild-type PDZ3, a low-affinity complex (Figure 1 B). Binding of the T₂F ligand involves a propagated structural perturbation in which the side-chain of H372 is forced to adopt a new rotamer state with two split conformations of roughly equal occupancy (presumably due to steric clash), and the β 2- β 3 loop (containing position 330) is in turn induced to partially adopt an alternate conformational state (Figure 5 D). The conformational heterogeneity at both H372 and the β 2- β 3 loop is consistent with the poor affinity of the wild-type protein for the T₂F ligand (Figure 1 C). How does G330T provide for high-affinity binding of both CRIPT and T₂F ligands? The G330T mutation stabilizes the β 2- β 3 loop in the non-native alternate conformation (Figure 5 E), a structural change that permits the H372 side-chain to adopt either rotamer state without steric penalty (Figures 5 F–G). Thus, the G330T variant can recognize both class I and class II ligands with high affinity, switching the rotameric state of H372 in a ligand-dependent manner (compare Figs. 5F and G). Finally, it is straightforward that addition of H372A in the background of G330T would (just like in the wild-type background) abrogate class I ligand recognition, completing the path of adaptation and resulting in the quantitative class II specificity observed in the double mutant (Figure 5 H). In summary, H372A works directly and locally at the binding pocket to simultaneously eliminate class I ligand binding and to promote class II ligand binding—the phenotype of direct switching. In contrast, G330T works allosterically to open up conformational plasticity at the binding pocket suitable for class I and class II recognition—the phenotype of class-neutral binding. The mutation opens up just one additional macroscopic conformational state (Figure 5 F–G) that is consistent

with high-affinity dual-class ligand specificity (Figure 3 B). Overall, these structures provide a model for the allosteric mechanism that underlies a class-neutral mutation in PDZ3.

2.4 Spatial distribution of conditional neutrality

The detailed study of G330T and H372A motivates a comprehensive analysis of all adaptive mutations to define the general structural rules. Such a study is made possible by a dataset comprising a total saturation mutagenesis of PSD-95 PDZ3³⁰ reporting the effect of every possible amino acid substitution at every position in the PDZ domain (1,598 total) on the binding of either the CRIPT ligand or the T₂F variant (Figure 6 A). This dataset permits enumeration of every mutation in PDZ3 that shows direct class-switching (e.g. H372A) or class-neutral ligand recognition (e.g. G330T). Using the 15 μ M cutoff for physiologically relevant binding, this analysis shows that while the vast majority of mutations are either neutral or destabilizing for both ligands, a subset of 44 mutations show gain-of-function for the T₂F ligand (shaded regions, Figure 6 A). Of these adaptive mutations, 12—like H372A—show loss-of-function for the CRIPT ligand (class-switching phenotype, red shade, Figure 6 A), and 32—like G330T—show near-neutrality for the CRIPT ligand (class-neutral phenotype, green shade, Fig. 6A). Mapping of the positions corresponding to these 44 mutations on the tertiary structure of PDZ3 shows the global spatial distribution of adaptation for the T₂F ligand (Figure 6 B) and leads to a simple conclusion. All class-switching mutations directly contact the site of adaptive challenge (T₂F), and all class-neutral mutations are invariably outside of the contact environment of T₂F (Figure 6 B). Only mutations to one residue (position 336) can generate both phenotypes, arguing that the distinction between class-switching and class-neutral phenotypes is typically a characteristic of the position rather than of the specific substitution at that position.

Interestingly, adaptive mutations are organized in the tertiary structure into a physically contiguous, wire-like network of residues linking the class-switching active site residues to class-neutral regions distributed throughout the protein structure (Figures 6 B, 7). The network is not isotropically organized in space around the T-2F site in a manner consistent with a simple model of spatial proximity to the site of adaptation. Instead it is an anisotropic network that fractures through the protein structure to include some distantly positioned residues at the expense of some more proximal ones (Figures 6 B, 7). For example, position 373 is in the immediate vicinity of T-2F but shows no mutations capable of adaptation. In contrast, position 362 is nearly 15 Å from ligand position -2, but has four mutations that create a binding pocket capable of recognizing both class I and class II ligands. Thus, the data argue that all class-neutral mutations are fundamentally allosteric in nature, working through a heterogeneous, epistatically-coupled network of amino acids within the protein structure to influence active site function from a distance.

2.5 The protein sector as the origin of adaptive mutations

As discussed previously, sectors represent a model for the relevant cooperative action of amino acid positions in proteins. We thus compared the pattern of adaptive mutations—both direct class-switching and class-neutral—with the pattern of coevolution in the PDZ domain family³⁰. The sector in the PDZ domain family (blue mesh, Figures 6 B, 7) comprises a group of 20 amino acid positions (~20% of total residues, default parameters, SCA 5.0) that form a network linking the ligand binding pocket to three regions: the β 2- β 3 loop, the α ₁ helix, and the end of the β ₄ strand (positions 362–363). Essentially all of the adaptive mutations, both directly class-switching and class-neutral, are contained within the sector (38/44 mutations at 8/12 positions, $p < 0.001$, Fisher Exact Test), and the six remaining mutations occur at four surface sites (358, 333, 371, 380), marked in yellow, that contact the peripheral edges of the sector (Figure 7). Importantly, keeping

number of top-scoring positions constant, neither spatial proximity to the site of adaptive challenge ($p = 0.47$, Fisher Exact Test) or the simpler analysis of position-specific conservation ($p = 0.07$, Fisher Exact Test) show such significant correlation with adaptive positions. Thus, at least for variation at the primary specificity site on the ligand, the capacity to adapt in the PDZ domain emerges from an evolutionarily ancient coevolving network of residues in the protein family. The coevolution of amino acids within the sector is consistent with the cooperativity and allosteric effect of class-neutral mutations, and generalizes the role of protein sectors as not only functional units of proteins, but as adaptive units of proteins.

2.6 Measuring structural couplings with X-ray crystallography

The presence of coupled set of conformations between the β_2 - β_3 loop and position 372 is evident from the structural data discussed. Could we extend the idea that collective motions lead to non-additivity in state functions to structural analysis itself? Since we have isomorphous crystal structures of each of the different PDZ3 variants, we can ask whether structural non-additivity can reveal the structural couplings underlying allostery, and, more generally, the sector in the PDZ family. In other words, is there a structurally-coupled path running along the binding cleft between the β_2 - β_3 loop and the α_1 helix (Figure 9 A)?

Our analysis builds on a method described previously for measuring structural coupling between related isomorphous structures³⁴, focusing on the G330T mutation to the PDZ3 and the T₂F mutation to the peptide. To quantitatively assess the degree of perturbation throughout each PDZ3 variant due to mutation—i.e., the first-order effect of mutation—we calculated error-weighted displacements of all non-hydrogen atoms. Explicitly, for a mutant protein m superposed

with a reference protein (here, using the wild-type structure), the normalized displacement of atom j of residue i relative to the reference is given by

$$\Delta r_{i,j}^m = \frac{|\vec{r}_{i,j}^m - \vec{r}_{i,j}^{\text{WT}}|}{\sqrt{\sigma_{i,j}^{m2} + \sigma_{i,j}^{\text{WT}2}}}$$

where $\vec{r}_{i,j}^m$ and $\vec{r}_{i,j}^{\text{WT}}$ are the centroid positions of each atom j in residue i in the wild-type and mutant structures and $\sigma_{i,j}^m$ and $\sigma_{i,j}^{\text{WT}}$ are the associated errors calculated based on the method of Stroud and Fauman for assessing the significance of atomic displacements³⁵. These errors are based on the crystallographic B -factor for the atom in each structure as well as the resolution of the diffraction data. The value of $\Delta r_{i,j}^m$ for an atom with a large B -factor is thus smaller than the value for an atom with a large B -factor. The average displacement over residue i is simply calculated by taking the average over all atoms J in the residue, i.e.,

$$\Delta r_i^m = \frac{1}{J} \sum_{j=0}^J \Delta r_{i,j}^m.$$

Now, consider the distribution of Δr for the G330T mutant relative to wild-type, in the presence of the CRIPT ligand. Previously, we demonstrated that biochemical and structural analysis reveal a role for the G330T mutation in tuning both local and long-range interactions within the protein structure. Can the underlying structural coupling be revealed through Δr analysis? Conformational heterogeneity is present at the β_2 - β_3 loop and around the α_1 helix (Figure 9 C), but little structural change occurs between the two regions.

To more completely study the effect of functional perturbation on structural coupling, we next considered the effect of the the T₂F mutation to the ligand. To do so, we calculated the

structural coupling parameter $\Delta\Delta r_{i,j}^{m,n}$. Given a thermodynamic cycle with combinations of two mutations, m and n , the structural coupling for atom j of residue i is given by the expression

$$\Delta\Delta r_{i,j}^{m,n} = \frac{|(\vec{r}_{i,j}^{m,n} - \vec{r}_{i,j}^n) - (\vec{r}_{i,j}^m - \vec{r}_{i,j}^{\text{WT}})|}{\sqrt{\sigma_{i,j}^{m,n^2} + \sigma_{i,j}^{m^2} + \sigma_{i,j}^{n^2} + \sigma_{i,j}^{\text{WT}^2}}}$$

As before, the average structural coupling was calculated over all atoms in a given residue.

We performed this calculation using the wild-type CRIPT, wild-type T₂F, G330T CRIPT, and G330T T₂F structures, as the T₂F mutation represents a direct functional challenge to PDZ3. Taking residues in the 80th percentile of the $\Delta\Delta r$ distribution as structurally coupled, we found coupling between a series of familiar regions—the β_2 - β_3 loop, the α_1 helix, and the end of carboxylate binding loop (CBL, β_1 - β_2 loop) near the ligand-binding site (Figure 10 A, B). Each of these components arises from different aspects of the structures used for the calculation. The signal for residues in the β_2 - β_3 loop originates from the wild-type T₂F structure, where displacement of H372 from its usual rotameric state leads to a split conformation of the β_2 - β_3 loop. This split conformation is then resolved with the G330T mutation. The same phenomenon applies in the case of the α_1 helix as well. Wild-type bound to both CRIPT and T₂F as well as G330T bound to CRIPT show no changes in the α_1 helix, while displacement is present in the G330T-CRIPT structure. Both of these cases suggest that the long-range structural effect of the G330T mutation is effectively decoupled from the rest of the protein in the context of binding to the T₂F peptide.

The non-additive displacement of the CBL is also notable. Wild-type bound to either peptide and G330T bound to CRIPT show the CBL in the “clamped” conformation (Figure 10 C), consistent with work by Doyle et al. showing that this loop packs against the side of the protein in the presence of ligand³¹. In the case of the G330T T₂F structure, however, residues 318–323 exist

in two states—clamped and unclamped, with 60% and 40% occupancy, respectively—suggesting that the β_2 - β_3 loop is also coupled through the ligand to the CBL (Figure 11 A). Physically, this region of the protein forms a direct interface with the C-terminus of the peptide, where V0 hydrogen bonds with the residues 322–325 (the GLGF motif, Figure 11 A). In the case of the G330T T₂F structure, this interface appears to be destabilized, with the average *B*-factor of the V0 backbone atoms being nearly double that of the same atoms in the other structures (Figure 11 B).

2.7 The pattern of structural coupling is consistent with allosteric and evolutionary models

Together, the pattern of coupling from this structural cycle reveals a physically contiguous network of residues which propagates anisotropically from the β_2 - β_3 loop to the α_1 helix and the CBL (Figure 11 C). Importantly, the majority of these residues are also found to be members of the sector for the PDZ family. Furthermore, the distribution of structurally coupled residues which are not found in the sector is found to be non-random; these positions cluster around the edges of the sector at the sites with the most dramatic degree of coupling. This pattern is robust to choice of cutoff for structural coupling. Relaxing the cutoff from the 80th percentile of the CDF to lower values results in the expansion of the pattern to residues adjacent to the sector.

This pattern is thus consistent with ideas related to allostery and the anisotropic propagation of energy throughout the protein structure. To test this model, we sought to disrupt this pattern through mutation. As such, we calculated the $\Delta\Delta r$ distribution for the structural cycle which includes the H372A mutation, as residue 372 lies directly along the observed path connecting the β_2 - β_3 loop to the α_1 helix and the CBL. Explicitly, this cycle includes four protein-ligand complexes—H372A CRIPT, H372A T₂F, G330T-H372A CRIPT, and G330T-H372A T₂F (Figure 4 B, C). While the pattern of structural coupling in the absence of the H372A mutation is

found to be heterogeneous and consistent an allosteric pathway through the protein sector, the cycle including the mutation is dramatically different, with structural coupling at key regions of the protein appearing quenched (Figure 12 A). That is, the effects of mutations at the ligand -2 position and in the β_2 - β_3 loop are now simply additive, suggesting that the H372A mutation removes coupling between these sites. This lack of physical coupling is explained by structural additivity in the region of the β_2 - β_3 loop, as the G330T mutation exerts similar influence in the background of either ligand, and by the lack of any conformational heterogeneity in the CBL or the α_1 helix in any of the structures in the cycle. We still do not entirely understand why structural double mutant cycles reveal this sort of pattern, however.

2.8 Saturation mutagenesis in the background of H372A supports conclusions from structural coupling

As an additional test, we measured the binding affinities of all possible single mutants in the background of the H372A mutation for the CRIPT peptide, and compared the mutational effects to those described previously for the wild-type protein. The data as well as details of the comparison are described in detail elsewhere³³, but several observations are relevant for our discussion here. First, the pattern of mutational sensitivity of H372A with respect to CRIPT binding is found to be similar as that for the wild-type protein at and around the α_1 helix (Figure 12 D). For the most part, mutations to positions 323 and 347 are found to have a dramatically negative effect on function in either genetic background. Positions 350–352 are largely neutral in either case. Position 353, however, shows a substantial gain-of-function effect in the H372A background, leading to a higher-affinity state. The mechanistic basis for this is unclear, as position 353 is several contact shells removed from the ligand binding cleft. Together, this suggests that— with the exception of position 353—mutation in and around the α_1 helix is largely unchanged in

the background of the H372A mutation, suggesting that the mechanism of coupling between the region and the ligand has not been substantially disrupted.

The β_2 - β_3 loop is a different story, however. Wild-type PDZ3 is highly sensitive to mutation at residues 329 and 330, with almost any mutation at these positions leading to a dramatic loss in CRIPT binding. Conversely, these mutations are essentially neutral in the background of the H372A mutation (Figure 12 C). This result reiterates the coupling between positions 330 and 372, consistent with the observation that the H372A decouples the β_2 - β_3 loop from α_1 .

3 Conclusions

3.1 A structural model for protein adaptation

The basic motivation for this work is the concept that evolutionary dynamics places non-trivial constraints on the “design” of natural proteins. An example of such a constraint is the existence of the class of epistatic mutations termed “conditionally neutral”—mutations that do not influence the existing functional activity, but that open up new activities that can be selected in the right environmental conditions^{9,11}. Such mutations can contribute to the standing genetic variation in populations, and can facilitate the acquisition of new phenotypes as selection conditions fluctuate^{11,12}. Thus the elucidation of general structural principles of conditional neutrality in protein molecules is a key next step in our understanding of their mechanisms and origin.

In this work, we demonstrate the existence, evolutionary relevance, and mechanism of conditionally neutral mutations in a member of the PDZ family of protein interaction modules. The G330T mutation in PSD-95 PDZ3 is a case study of conditional neutrality, providing binding phenotypes that act as bridges between physiologically distinct substrate specificities. We show that the mechanism is allosteric in nature, working from a distance through a network of amino acid interactions to open up conformational plasticity at the ligand binding pocket. In contrast, adaptive mutations such as H372A located at the active site have the property of direct switching of ligand class-specificity—new ligand binding is gained at the expense of binding to the existing ligand. A global study of all possible adaptive mutations reinforces the generality of these conclusions: direct-switching occurs directly at the site of adaptive challenge, and class-neutral phenotypes originate from a network of amino acids that is distributed anisotropically through the protein structure. Importantly, simulations of evolutionary dynamics confirm the notion that it is

the class-neutral mutations, not the direct switching ones, that are likely to serve as intermediates in adaptation. Thus, we conclude that in addition to its contributions to functional properties such as signal transmission and regulation, intramolecular allostery plays a key role in facilitating the evolutionary process.

These findings suggest an “outside-in” structural principle for protein adaptation. The idea is that the path of adaptation likely starts from the acquisition of mutations at positions distant from the active site, but that are wired up through a pre-existing network of cooperative amino acid interactions. Through action at a distance, these mutations have the capacity for opening up active site conformational states that can introduce new functional phenotypes without abrogating existing function—the property of conditional neutrality. From a point of view of evolutionary dynamics, the key benefit of such mutations is that they can temporally unlink the appearance of adaptive mutations from environmental fluctuations that alter selection pressures. The neutrality protects against purifying selection, and enables adaptive mutations to exist in populations as cryptic variations. In contrast, mutations that cause direct phenotypic switching (even if structurally more obvious) can only support adaptation with temporal coupling of mutation and selection pressures. More generally, we propose that the degree of neutrality of adaptive mutations towards existing function will set the time scale (relative to mutation rate and environmental switching rate) over which it can support adaptation. Thus, conditionally neutral mutations would seem to represent a pool of natural variation that is the engine for evolution of new phenotypes.

3.2 Implications for protein engineering

The “outside-in” concept for protein adaptation is interesting as it is essentially opposite to the current practice of structure-guided protein engineering. For example, classic work on switching

the primary (P1 site) substrate specificity of the serine protease trypsin to that of chymotrypsin³⁶ began with mutations at sites directly contacting the P1 side chain (the S1 pocket). The result was initial loss of enzyme function, a phenotype explained by collapse of the S1 pocket upon mutation³⁷. Successful transfer of specificity required the subsequent addition of mutations at positions distributed through the protein structure, which have the effect of restoring stability (and new functional specificity) to the S1 pocket. Similarly, attempts to switch activities of type II restriction enzymes^{38,39}, transcription factors⁴⁰ and beta-lactamases⁴¹ show that active site mutations tend to display loss of function, while combinations with structurally non-obvious peripheral mutations facilitate acquisition of new function. In light of the work presented here, a useful avenue for protein engineering might be to target mutations not by the principle of spatial proximity to the active site, but by the spatial pattern of adaptive mutations.

In this regard, it is interesting that the adaptive mutations, both direct-switching and class-neutral, occur within the network of coevolving positions (the sector) in the PDZ and serine protease families². This finding strongly argues that the pattern of adaptive mutations is not merely an idiosyncratic feature of each model system but is instead a deeply conserved aspect of the entire protein family that can be predicted through sequence analysis alone. It will be interesting to combine sector predictions and the principle of outside-in mutagenesis to explore new general strategies for the evolution and engineering of new protein functions.

3.3 Origins of allostery

The finding that the protein sector contains adaptive mutations offers an interesting hypothesis about the origin of this cooperative internal architecture within protein tertiary structures. Sectors are coevolving units of protein structures, and have been associated with various functional

properties of proteins—catalysis, binding, and allosteric signaling^{2,4,42}. A natural inference might be that the wire-like architecture of sectors, connecting active sites to distant surface sites through the protein core (Figure 7), emerged in evolution as a consequence of selection for the corresponding functional property. However, it is not obvious how such a network of cooperative amino acid interactions could be built through a process of stepwise variation and selection given that intermediate genotypes are not guaranteed to be functional.

The data presented here suggest another model: sectors are primarily a consequence of a history of adaptation to fluctuating conditions of fitness, with the wire-like distributed architecture evolving simply because conditional neutrality is enabled by non-local allosteric mechanisms. That is, we propose that the origins of allostery lie in evolvability, not in function. Per this model, functional properties that make use of allostery (e.g. long-range regulation and signal transmission) are derivatives that emerge easily at multiple surface sites through engagement of the pre-existing allosteric network. Indeed, experiments suggest that it is possible to naively engineer new allosteric control into proteins through engagement of sector connected surface sites^{25,43}. The recent development of techniques for fast continuous evolution of proteins⁴⁴ may help in designing experiments to test these ideas.

3.4 Structural couplings and the physical origins of the sector

Analysis of structural coupling provides a physical model which can explain the properties of these networks. While a computational study⁴⁵ explored the relationship between energy propagation in PDZ3 and the PDZ family sector, this is the first experimental study to link the physics of proteins to the sector. Here, we have revealed a pathway of physical coupling in PDZ3 which is revealed upon mutation within the sector. By exploring the physical coupling between

the G330T mutation to the domain and the T₂F mutation to the peptide ligand, we show that a pathway consistent with the protein sector links the β_2 - β_3 loop of the domain to a distant site previously shown to undergo changes in dynamics upon ligand binding⁴⁶. Furthermore, this pathway can be disrupted; by introducing an additional mutation, H372A, we demonstrate that correlations between the β_2 - β_3 loop and the α_1 helix can be quenched. The fact that second-order cycles are required to see this pattern of coupling suggests that the allosteric network being probed is perhaps consistent with the general concept of dynamic allostery as proposed by Cooper and Dryden²⁰, in which conformational change is not needed for allosteric signaling.

Furthermore, results described here provide the first experimental evidence for the presence of sectors in individual proteins, a non-trivial observation given that sectors are, in principle, a statistical description of the energetic composition of a protein. It is possible that the high degree of correlation between the statistical coevolution in the protein family and the structural coupling within an individual of the family is due to the emphasis on identifying conserved statistical couplings within an alignment. While idiosyncratic interactions specific to a particular member of the family may not be conserved, interactions under evolutionary selection would be constrained and therefore be present to a large degree in many individual family members. We therefore believe that the observed agreement between the structural couplings in PSD-95 PDZ3 and sector residues in the PDZ family can fundamentally be ascribed, at least in part, to our attempt of statistically identifying evolutionarily significant, and therefore conserved, couplings within a protein family.

4 Experimental procedures

4.1 Global analysis of PDZ ligand specificity

Comprehensive study of PDZ binding specificity is made possible by a modified version of a bacterial two-hybrid system³⁰ in which transcription of the chloramphenicol acetyl transferase (CAT) reporter gene (pZE1RM plasmid, pRM+ promoter, ampicillin resistant) is made quantitatively dependent on the binding between a PDZ domain (fused to the pRM+ promoter-binding bacteriophage λ -c1 DNA binding domain, pZS22 plasmid, IPTG inducible, trimethoprim (trm) resistant) and its ligand (fused to the N-terminal domain of *E. coli* RNA polymerase α subunit, pZA31 plasmid, anhydrotetracycline (aTC) inducible, kanamycin resistant) (Figure 8). Electrocompetent MC4100-Z1 cells containing pZE1RM-CAT and pZS22-PDZ3 variant plasmids were transformed with 1 ml of 20 ng μl^{-1} pZA31-RNA α -ligand library (see below), recovered for one hour in LB media, grown in 20 mg/ml trm, 50 mg/ml kan, 100 mg/ml amp to OD₅₅₀ of 0.04, and induced using 50 ng/ml doxycycline plus antibiotics for 3 hours to an OD₅₅₀ of 0.1. 10 ml of the induced culture was used to inoculate 100mL LB + antibiotics as above for selection; the remainder was reserved as the pre-selection population for deep sequencing. Selection was carried out with 150 mg/ml chloramphenicol for 6 hours (taking care that OD₅₅₀ \leq 0.1), washed in LB medium, and grown overnight at 37°C. Both pre- and post-selection cultures were subject to plasmid DNA isolation, PCR amplification of the ligand region of pZA31, and standard preparation for Illumina Hi-Seq 2500 sequencing (UT Southwestern genomics core).

4.2 Construction of the ligand library

The library of PDZ ligands (randomized in the C-terminal four amino acid positions, total theoretical library complexity $20^4 = 160,000$) was generated as C-terminal fusions with the N-terminal domain of *E. coli* RNA polymerase α subunit. The library was made using NNS oligonucleotide-directed mutagenesis with a pZA31-RNA α template containing a non-binding PDZ ligand (N-TKNYKQGGG-COOH) to eliminate background binding. Two oligonucleotides

(one sense, one antisense) were synthesized (IDT) with each sequence complementary to 15 base-pairs (bp) on either side but with one oligo containing four consecutive NNS codons at the target positions; N is a mixture of A, T, C, and G, and S is a mixture of G and C. This results in 32 codons at each position encoding all 20 amino acids. The oligos include a type II restriction site (BsaI), designed to optimize cloning efficiency by enabling a unimolecular ligation protocol. We carried out a single round of PCR, amplifying the entire plasmid while encoding the full library of ligand sequences. This product was subsequently restricted with BsaI, subject to a unimolecular ligation reaction (1 ml, incubated overnight at 16°C), and purified into a final volume of 10 ml. Ten individual transformations into MaxDH10B *E. coli* (Invitrogen) were made, grown overnight after recovery, and plasmid DNA prepped so as to minimize any possible bottlenecking effect. Transformation of the final library into MC4100-Z1 cells for selection yielded greater than 10^8 transformants, and a near complete representation of the theoretical complexity (Table S1).

4.3 Expression and purification of PSD-95 PDZ3 proteins

All rat PSD-95 PDZ3 constructs (amino acid range 297–415) were subcloned into pGEX-4T-1 (GE Healthcare) and verified by DNA sequencing. Proteins were expressed as glutathione S-transferase-tagged fusion proteins in *Escherichia coli* BL21(DE3) cells at 18°C overnight in either terrific broth (TB) supplemented with 1 mM isopropyl β -D-1-thiogalactopyranoside (IPTG) or ZYM-5052 auto-inducing medium⁴⁷. Cells were harvested by centrifugation, resuspended in lysis buffer composed of phosphate-buffered saline (PBS; 140 mM NaCl, 2.7 mM KCl, 10 mM Na₂HPO₄, 1.8 mM KH₂PO₄, pH 7.3) supplemented with 1% glycerol, 1 mg ml⁻¹ hen egg white lysozyme, 1 mM dithiothreitol (DTT), EDTA-free protease inhibitor cocktail (Roche). The cell suspension was subjected to sonication and centrifugation, and clarified lysate was then incubated with glutathione sepharose 4B resin (GE Healthcare). Bound protein was washed with PBS

supplemented with 1% glycerol and 1 mM DTT, and the GST tag was then cleaved overnight at room temperature with bovine thrombin protease (Calbiochem) in PBS supplemented with 10% glycerol and 1 mM DTT. Following cleavage, thrombin was removed from solution with benzamidine sepharose (GE Healthcare). The protein solution was diluted ~10-fold into salt-free anion exchange buffer (20 mM Tris HCl pH 7.5, 1% glycerol, 1.0 mM DTT), applied to Source 15Q anion exchange resin (GE Healthcare), and washed extensively prior to elution over a gradient of high-salt anion exchange buffer (20 mM Tris HCl pH 7.5, 1 M NaCl, 1% glycerol, 1 mM DTT). Peak fractions were dialyzed overnight against size exclusion buffer (10 mM HEPES pH 7.2, 10 mM NaCl), concentrated, and further purified by gel filtration on Superdex 75 (GE Healthcare). Peak fractions were pooled and concentrated to ~35 mg ml⁻¹ for crystallization.

CRIPT (Acetyl-TKNYKQTSV-COOH) and T₂F (Acetyl-TKNYKQFSV-COOH) peptides were synthesized using standard Fmoc chemistry (UTSW Proteomics Core Facility), HPLC purified, and lyophilized.

4.4 Protein crystallization and X-ray data collection

Crystallization was performed by the vapor diffusion hanging drop method. In all cases, purified protein was diluted to a final concentration of ~9 mg mL⁻¹ in protein buffer (10 mM HEPES pH 7.2, 10 mM NaCl). Where applicable, peptide was included in protein buffer to a final molar ratio of 2:1 relative to protein. Reservoir solutions typically contained ~1 M sodium citrate, pH 7.0; specific crystallization conditions for each mutant are shown elsewhere (Table 6). Equal amounts (1.5 μL) of protein and reservoir solution were mixed and equilibrated against 500 μL of crystallization buffer at 16°C. Diamond-shaped crystals appeared either spontaneously or with microseeding after 1–5 days and grew to 100–200 μm in length over several weeks. To prepare

microseeding solutions, wild-type crystals of the appropriate state were crushed and resuspended in crystallization buffer. Single crystals were cryoprotected by serial equilibration into crystallization buffer with increasing amounts of glycerol (up to 25%) and flash frozen in liquid N₂.

Data were collected at 100 K at either UT Southwestern Structural Biology Laboratory with a rotating-anode X-ray generator (Rigaku Americas Corp., The Woodlands, Texas, USA) or at beamline 19-ID of the Advanced Photon Source at Argonne National Laboratory (Argonne, IL, USA). Crystals diffracted to 2 Å or better and were found to have P4₁32 symmetry, with all unit cell lengths equal to ~90 Å and all angles equal to 90°.

4.5 X-ray data reduction, model building, and refinement

Diffraction data for all crystals were indexed, integrated, and scaled using HKL-2000 suite⁴⁸ (HKL Research, Inc). Resolution cutoffs were chosen based on I/σ and $CC_{1/2}$. Data reduction statistics are summarized in Table II. All automated model building was performed using the PHENIX software suite⁴⁹, and manual model building was performed using COOT⁵⁰. Riding hydrogens were used throughout, and X-ray/stereochemistry and X-ray/ADP weights were optimized during automated refinement. Molecular replacement with either apo wild-type PSD-95 PDZ3 (PDB ID 1BFE) or CRIPT-bound wild-type PSD-95 PDZ3 with ligand removed (PDB ID 1BE9) was used throughout. Initial models were subjected to 0.5 Å coordinate randomization using *phenix.pdb_tools* prior to refinement and then refined against the data using *phenix.refine*. During the first round, rigid body refinement ensured correct placement of the model and Cartesian simulated annealing was used to help reduce phase bias. The N-terminal region of the protein domain was then extended, and the ligand residues and waters were added where supported by the

electron density. Cartesian coordinates, individual *B*-factors, and occupancies were then further refined. In some cases, the N-terminus was found to bifurcate at L302, with the occupancies of residues 297–302 split between two states. Once crystallographic R-factors showed signs of diminishing returns, a translation-libration-screw (TLS) model was chosen. A series of models with different numbers of groups were calculated, both with *phenix.find_tls_groups* and the TLS Motion Determination web server^{51,52}. The TLS model which led to the greatest decrease in crystallographic R-factors was chosen. Further refinement was then performed until R-factors could not be substantially improved and validation statistics as reported by MolProbity⁵³ were acceptable. All model building statistics are summarized in Tables 2–5.

5 Figures

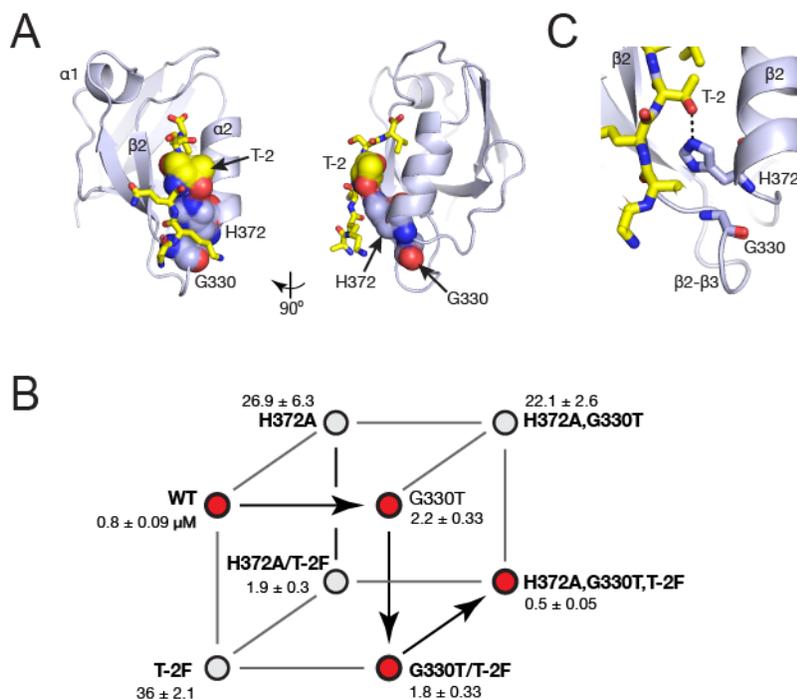


Figure 1. A two-mutation path to new functional specificity in a PDZ domain. **A.** The overall structure of the PDZ domain (PSD-95 PDZ3) bound to the CRIPT C-terminal peptide (yellow stick bonds). Positions G330 and H372 in the protein, and T₂ in the ligand peptide are shown as spheres with an overlaid van der Waals surface. **B.** A thermodynamic cube showing the effects of the G330T and H372A mutations in the context of the wild-type CRIPT ligand (top face) and the T₂F ligand (bottom face). Wild-type PSD-95 PDZ3 shows a 45-fold preference for the CRIPT ligand, while the G330T-H372A double mutant shows a 45-fold preference for the T₂F ligand. **C.** A close-up showing stereochemical details around ligand position -2; H372 makes a hydrogen bond with the class I-defining threonine side chain of ligand position -2, and G330 occurs on a surface loop (β_2 - β_3) that is packed against the region of position 372.

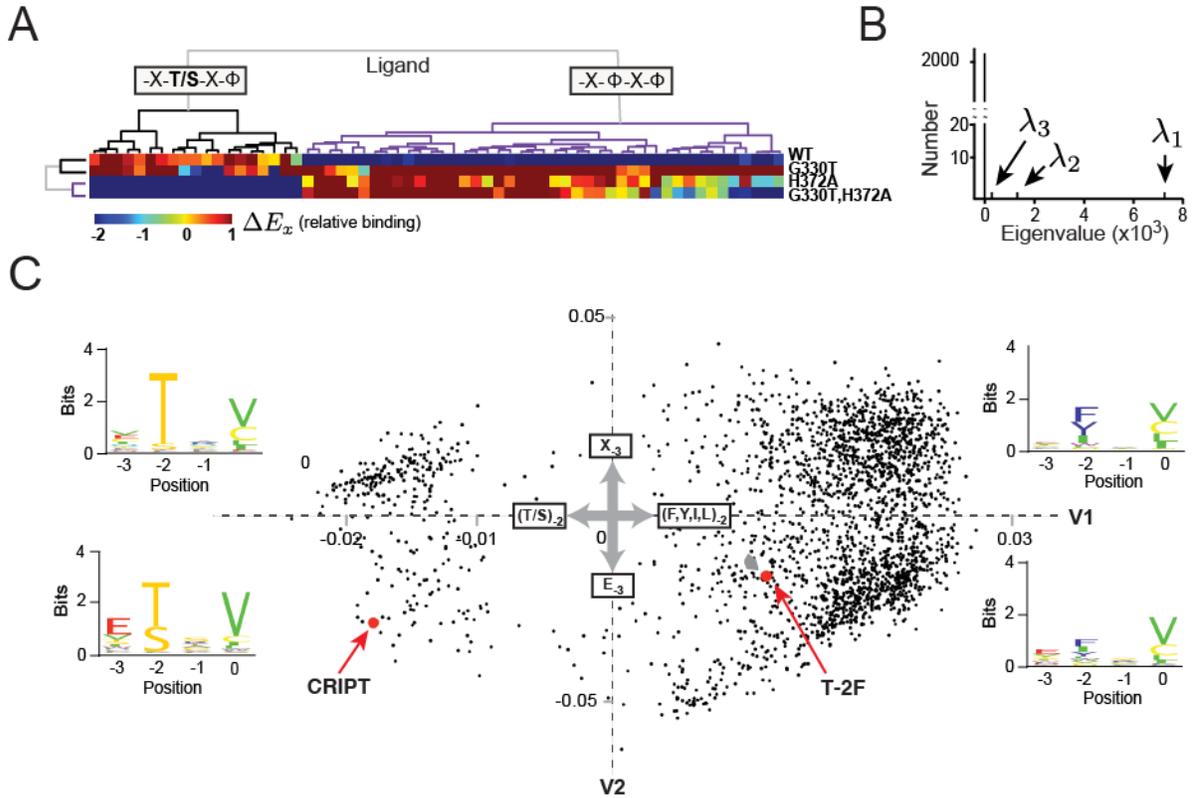


Figure 2. A global mapping of primary ligand specificity in the PDZ domain. The panels show the outcome of a quantitative bacterial two-hybrid assay in which we measure the binding of wild-type, G330T, H372A, and the double-mutant variants of PSD-95 PDZ3 to a library of C-terminal peptides randomized in the terminal four residues (154,521/160,000 ligands measured). Of these, 2,359 show better than 15 μM binding to at least one PSD-95 PDZ3 variant and are analyzed here. **A.** A clustered heat map showing a small representative sampling of data. Each pixel shows the binding of one ligand x (ΔE_x , log scale), normalized so that zero represents wild-type binding ($\sim 1 \mu\text{M}$). Ligands (columns) cluster by known class specificities, and proteins (rows) show profiles consistent with the study of the CRIPT and T₂F ligands (Figure 1 B): wild-type PSD-95 PDZ3 shows class I specificity, H372A and G330T-H372A show class II specificity, and G330T shows dual specificity. **B.** The eigenvalues of the correlation matrix of ligand profiles, showing that the top two eigenmodes account for essentially all the relationships between ligands. **C.** The ligand space defined by the top two eigenmodes. Dots correspond to the 2,359 physiologically relevant ligands and proximity of two dots indicates similarity in the binding profile over the four PDZ proteins assayed. The insets show amino acid motifs for the peptides in each quadrant of the map. The first eigenvector (V1) separates ligands by identity at the class-defining position -2, with class I ligands in the left half and class II ligands in the right half. The second eigenvector (V2) separates ligands by a motif involving both positions -2 and -3.

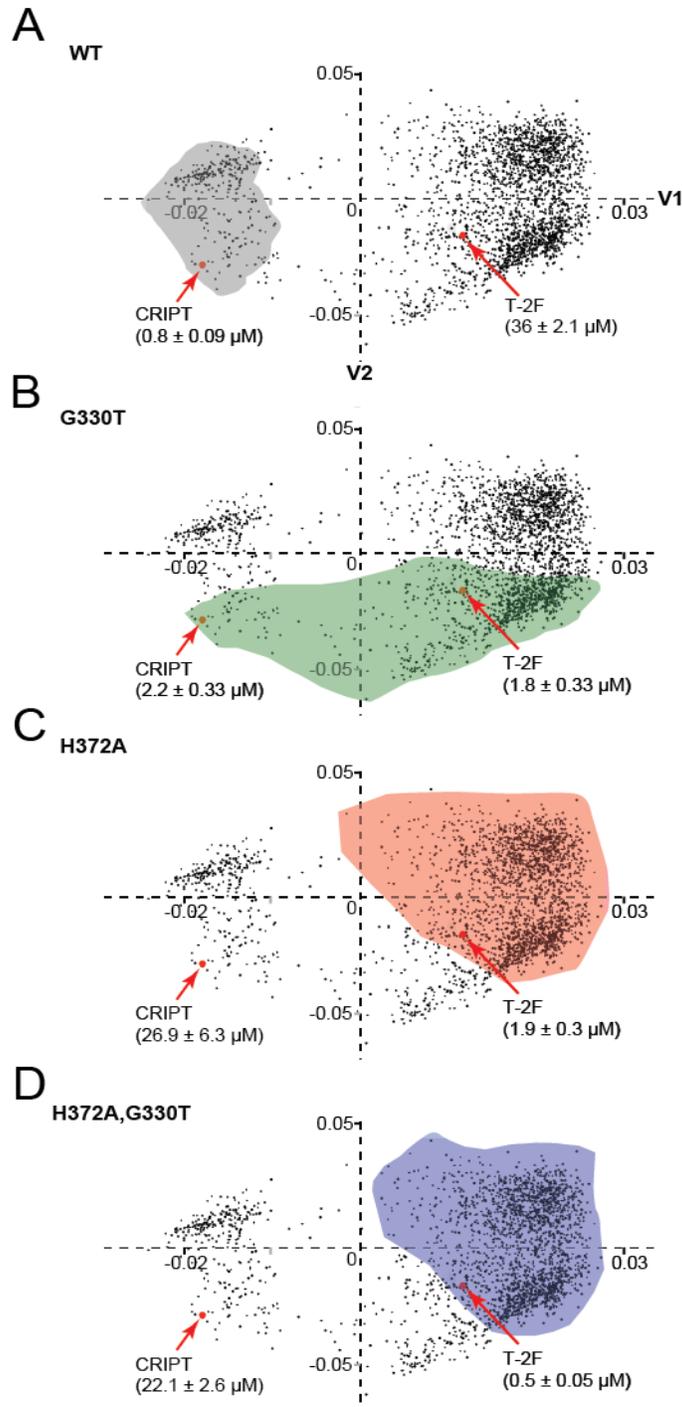


Figure 3. Binding phenotypes of PDZ variants along the adaptive path. **A–D.** The ligand space defined in Figure 2, with shaded regions indicating the peptides recognized by wild-type (grey), G330T (green), H372A (red), and the double-mutant (blue) variants of PSD-95 PDZ3, respectively. The boundary is defined by $K_D \leq 15 \mu\text{M}$. The data show that wild-type binds exclusively to class I ligands, H372A and the double mutant bind exclusively to class II ligands, and that G330T is a bridge between class specificities, binding a subset of ligands in both class I and class II regions.

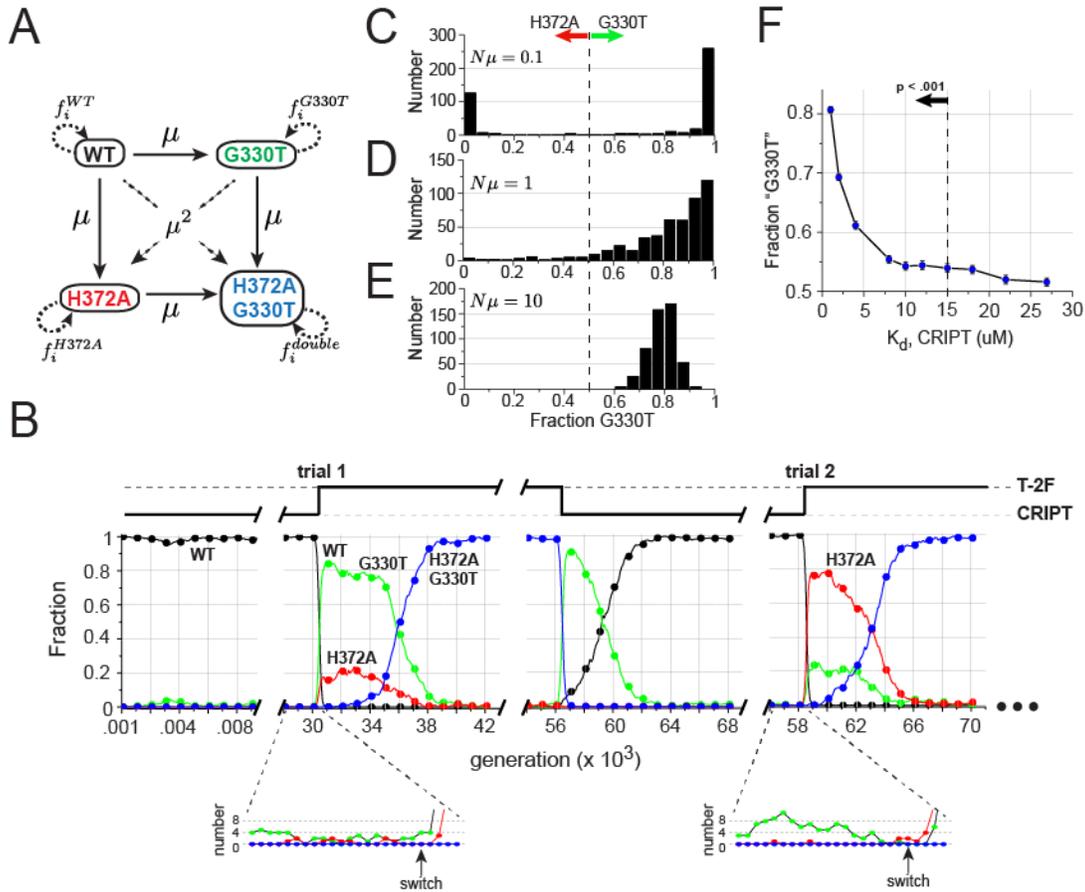


Figure 4. The preferred path of adaptation. **A.** A population dynamics model for the path of adaptation between the wild-type and G330T-H372A double-mutant genotypes. Simulations are initiated with a population of N wild-type individuals, and at each generation, single mutations are allowed with rate μ and double mutations with rate μ^2 , ligands switch between CRIPIT and T₂F with rate τ , and the fitness of each genotype is defined as the fraction bound of ligand. The simulation permits a quantitative analysis of the relative flux through G330T or H372A along the path to the double mutant state. **B.** A simulation trajectory at a particular mutation and ligand-switching rate ($\mu = 1$ and $\tau = 500$), showing two trials of adaptation in response to switching from the CRIPIT ligand (class I) to the T₂F ligand (class II). These two examples show that different proportions of the two single mutants can act as intermediates, depending on the pre-existing population of G330T and H372A variants at the moment of ligand switching (insets). **C–E.** Histograms of the fraction of G330T ($n^{G330T} / (n^{G330T} + n^{H372A})$), where n represents integrated counts over the period of switching to the double mutant state) for 500 trials of switching from the CRIPIT to T₂F ligands. The analysis is shown for three regimes of mutation rate: $N\mu = 0.1$, where mutations are rare (**C**), $N\mu = 1$ (**D**), and $N\mu = 10$, where mutations are abundant (**E**). The data show that G330T is always the preferred path of adaptation to the double mutant state upon ligand switching. **F.** The preference of a hypothetical “G330T” variant over H372A for adaptation to the T₂F ligand as a function of computationally varying the affinity for the CRIPIT ligand from 1 μM to 26.9 μM ,

the same affinity as H372A. The analysis shows that given model parameters, affinities as low as 15 μM can still provide a statistically significant advantage over H372A in facilitating adaptation (Wilcoxon rank sum test).

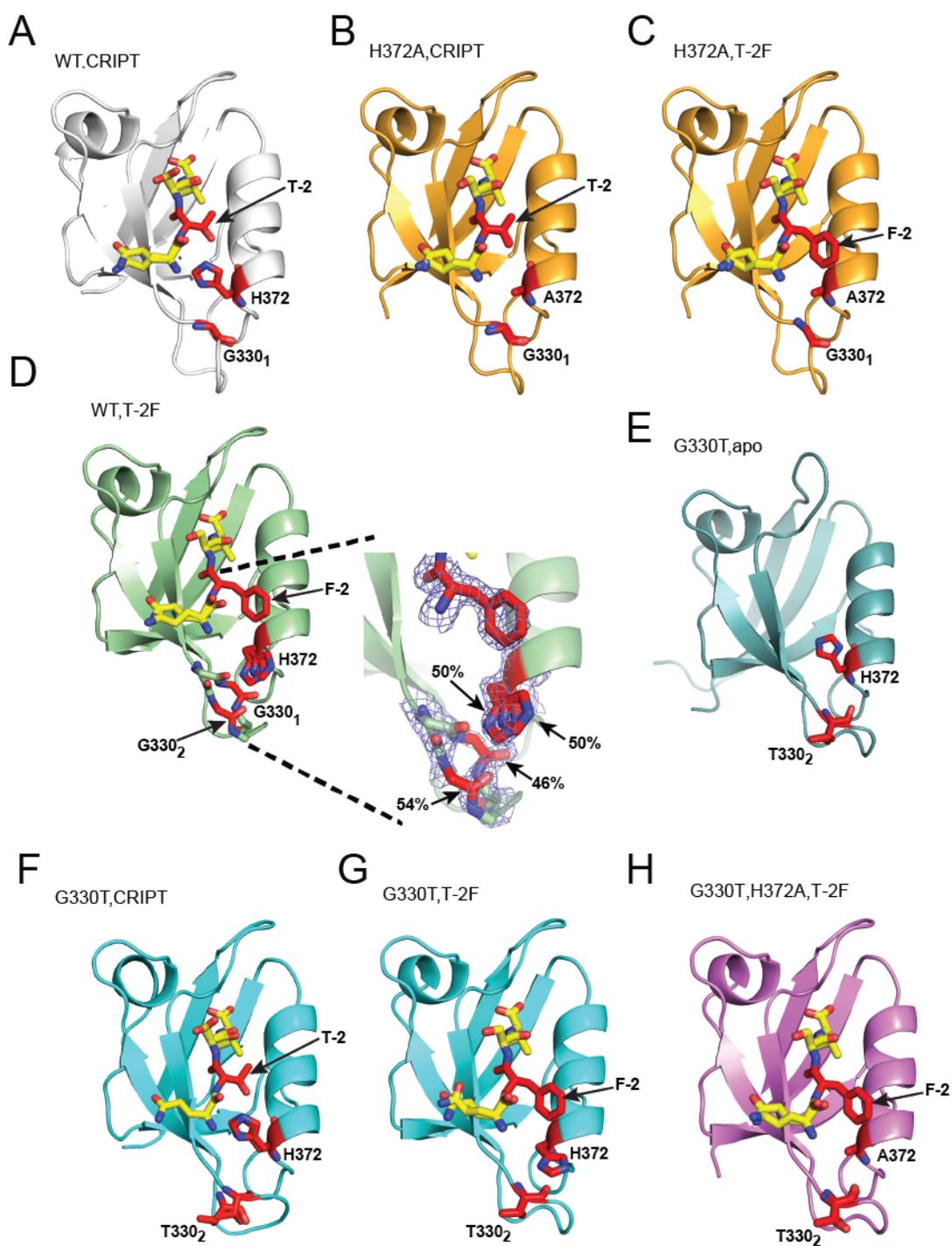


Figure 5. The structural basis for ligand specificity switching. The panels show high-resolution crystal structures of wild-type (WT), H372A, G330T, and double-mutant variants of PSD-95 PDZ3, either in the apo state or bound to CRIP T or T₂F ligands. **A.** The WT-CRIP T structure, recapitulating features of class I ligand recognition; the threonine hydroxyl of -2 is hydrogen bonded to histidine 372, and G330 is located on a well-ordered β_2 - β_3 loop packed against the region of 372 (G330₁, with the subscript indicating conformation 1). **B–C.** The H372A structures show

truncation of the 372 side chain and little other conformational change, a local perturbation permitting accommodation of the phenylalanine side chain at -2 without steric clash. The loss of both bulk and hydrogen bonding potential at position 372 are consistent with the partially class-switching phenotype of H372A. **D.** Binding of the T₂F ligand to wild-type PSD-95 PDZ3 causes rotation of H372 to a new, non-native rotamer state (to prevent steric clash), and induction of two partially occupied conformational states of the β_2 - β_3 loop (marked by G330₁ and G330₂). **E–G.** The G330T mutation (apo-state) stabilizes the β_2 - β_3 loop in the alternate conformation 2 (**E**), a state that can permit either rotamer of H372 without steric clash. Thus in G330T, the H372 side chain occupies the native rotameric state in binding CRIPT (**F**) as opposed to the alternative rotameric state in binding T₂F (**G**). **H.** The structure of the G330T-H372A double mutant bound to T₂F is similar to that of H372A alone (**C**), except that the β_2 - β_3 loop is in conformation 2, consistent with G330T.

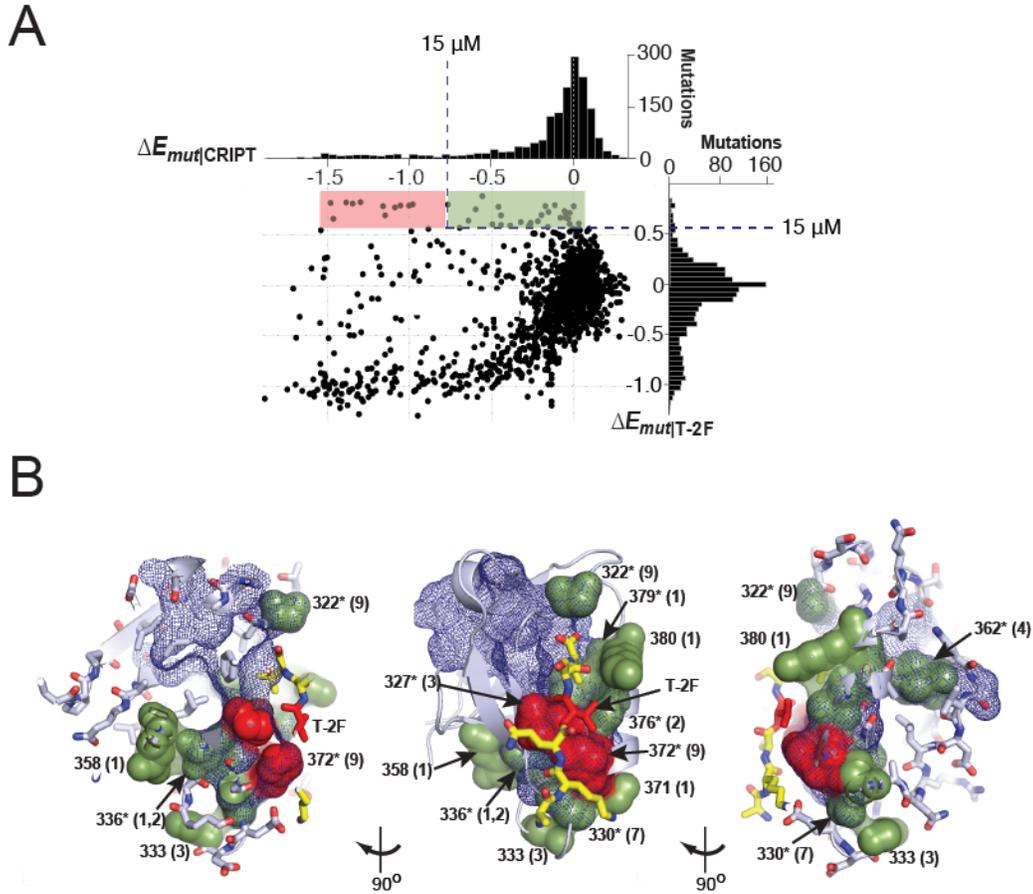


Figure 6. Spatial architecture of adaptive mutations in response to T₂F. **A.** A scatter plot showing the effect of all possible single mutations in PSD-95 PDZ3 on binding to either the class I CRIPT ligand (ordinate) or the class II T₂F variant (abscissa). The shaded regions describe physiologically significant ($\leq 15 \mu\text{M}$) binding to the T₂F ligand, with either simultaneous loss of function (red) or physiological neutrality (green) for the CRIPT ligand. **B.** Three rotations of PSD-95 PDZ3 with all positions containing adaptive mutations for T₂F in sphere representation with colors as in panel A. Thus, red spheres correspond to positions with direct switching mutational phenotypes, and green spheres indicate positions with class-neutral phenotypes; the number of mutations at each position with that phenotype are shown in parentheses. The blue mesh indicates the protein sector—the network of coevolving positions in the PDZ family. The data show (1) that nearly all adaptive mutations are contained in the protein sector, (2) that direct switching phenotypes localize to the site of adaptive challenge (ligand -2), and (3) that class-neutral phenotypes arise from a distributed, physically connected network of residues leading away from the binding pocket through the sector.

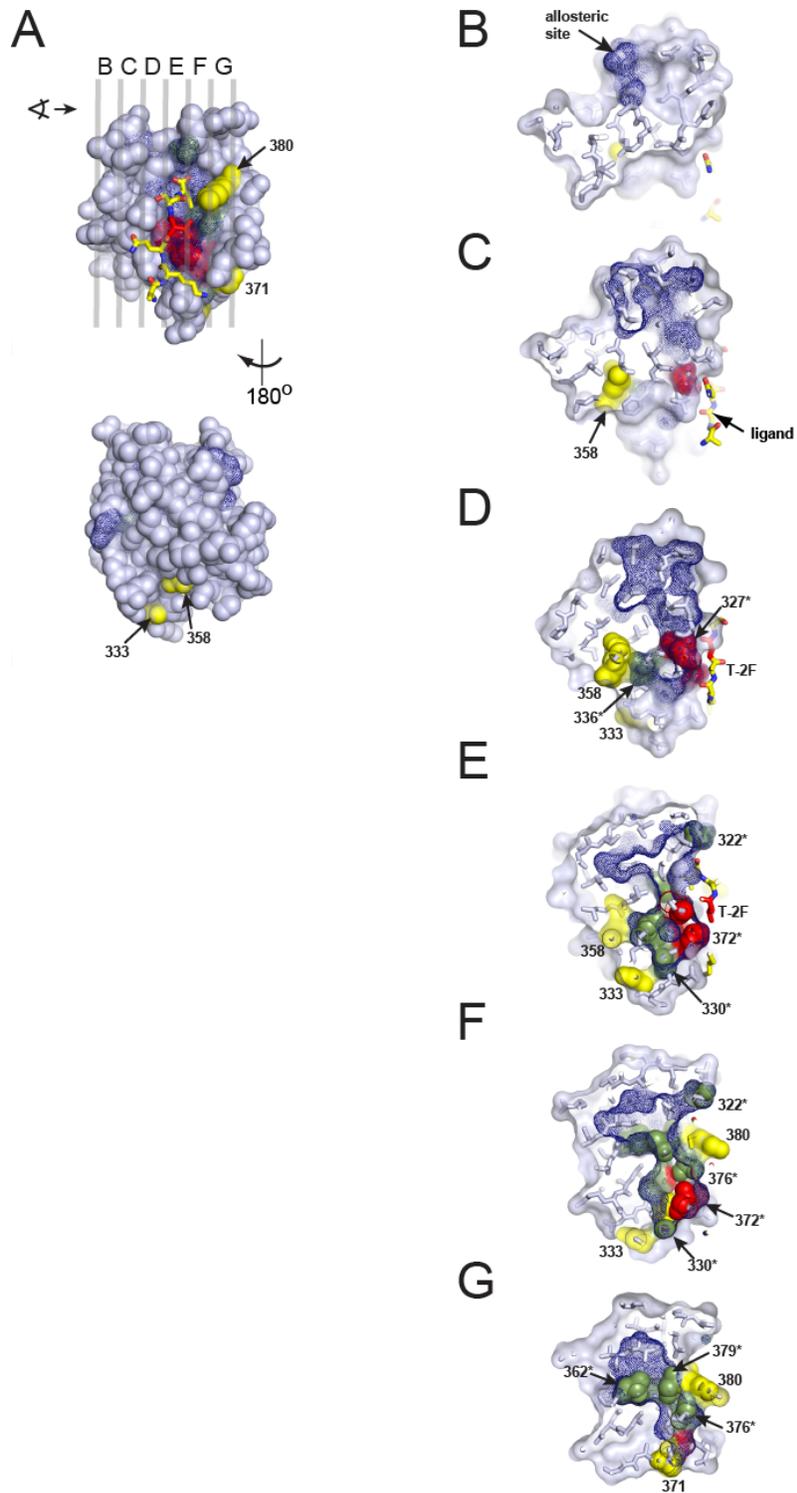


Figure 7. Relationship of the protein sector to adaptive positions. **A**, A space-filling representation of PSD-95 PDZ3, with the protein sector in blue mesh, and positions capable of adaptation to the T₂F ligand colored red (direct-switching, in sector), green (class-neutral, in sector), or yellow

(class-neutral, non-sector). The data show that the four non-sector adaptive mutations (labeled) occur at surface exposed sites distant from ligand position -2 (in red stick bonds). **B-G**, Serial slices through PSD95pdz3 at the planes indicated in **A**; the views are from the left as indicated. The data show that adaptive positions are nearly all contained within the protein sector (overlap of blue mesh with red and green spheres), and the four surface exposed positions with class-neutral mutations are connected to the peripheral regions of the protein sector. Overall, adaptive positions comprise a wire-like system of physically connected residues that connects the site of adaptive challenge (ligand -2) through the protein structure.

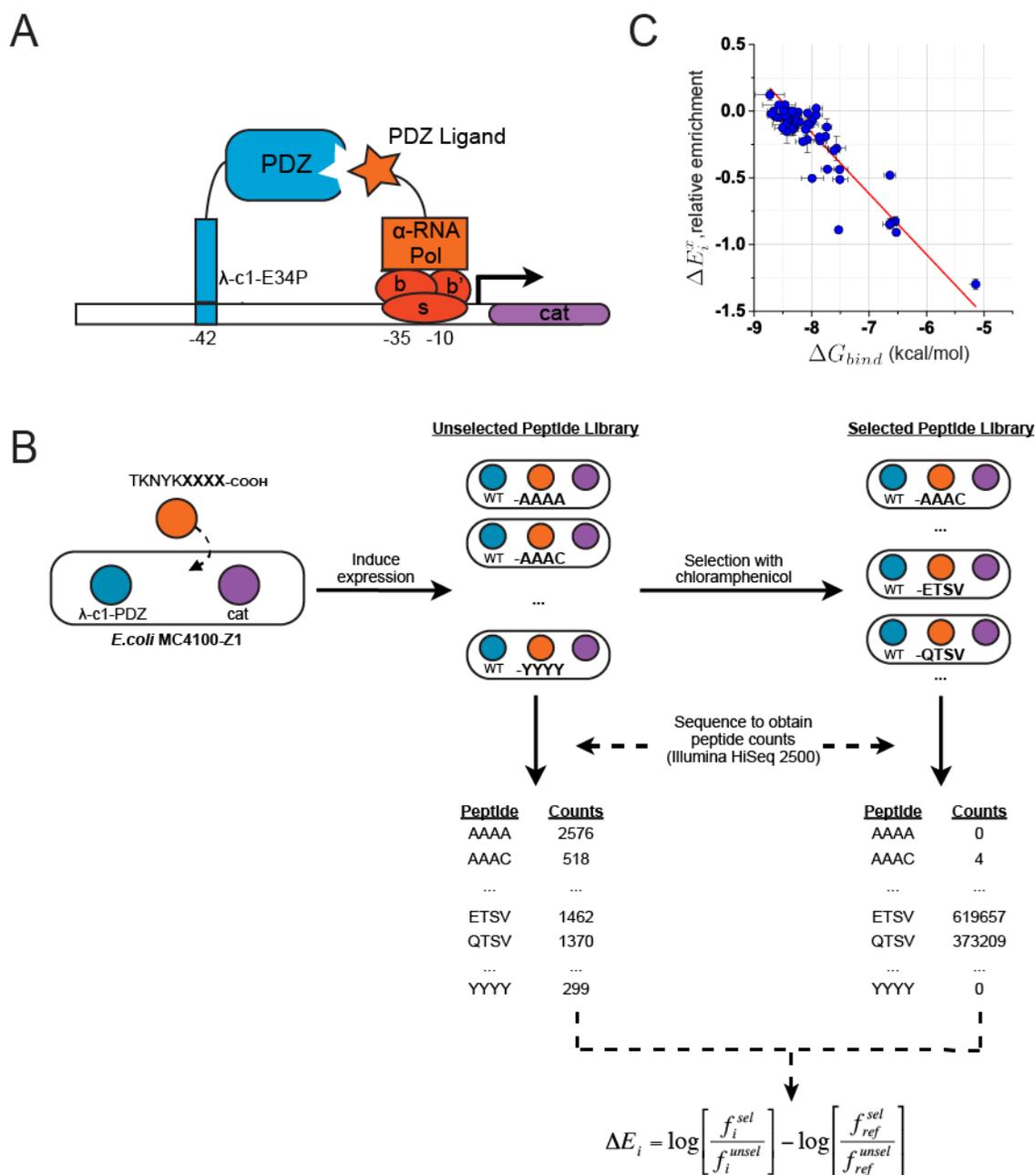


Figure 8. The bacterial two-hybrid (BTH) assay, and the PDZ ligand screen. **A.** The BTH assay is modified from previous work to give a quantitative readout of the affinity between a PDZ domain and its peptide ligand in the expression of a target gene. The PDZ domain is expressed as a C-terminal fusion with the DNA binding domain of the bacteriophage λ -cI protein (with the E34P mutation to optimize dynamic range). The peptide ligand is expressed as a C-terminal fusion to the N-terminal domain of the *E. coli* RNA polymerase α -subunit, and the target gene is chloramphenicol acetyl transferase (*cat*). **B.** A library of potential PDZ ligands (randomized in terminal four residues, $20^4 = 160,000$ total) is transformed into *E. coli* MC4100-Z1 cells expressing the other components of the BTH assay, and are selected on chloramphenicol for PDZ function (see methods for details). The ligand library is subject to Illumina MiSeq sequencing to count the

frequency of each C-terminal peptide in the library in both the unselected and selected populations. As shown for a few examples, legitimate ligands are highly enriched in the selected population while non-binding ligands are absent. Quantitatively, we compute the enrichment of ligands relative to reference ligand (“relative enrichment”) according to the equation shown. C. a standard curve showing the relationship of relative enrichment to the equilibrium free energy of binding for a set of 83 single point mutations in the PSD-95 PDZ3 domain to the CRIPT ligand. The experiment shows that the sequencing based assay provides a quantitative measure of ligand binding.

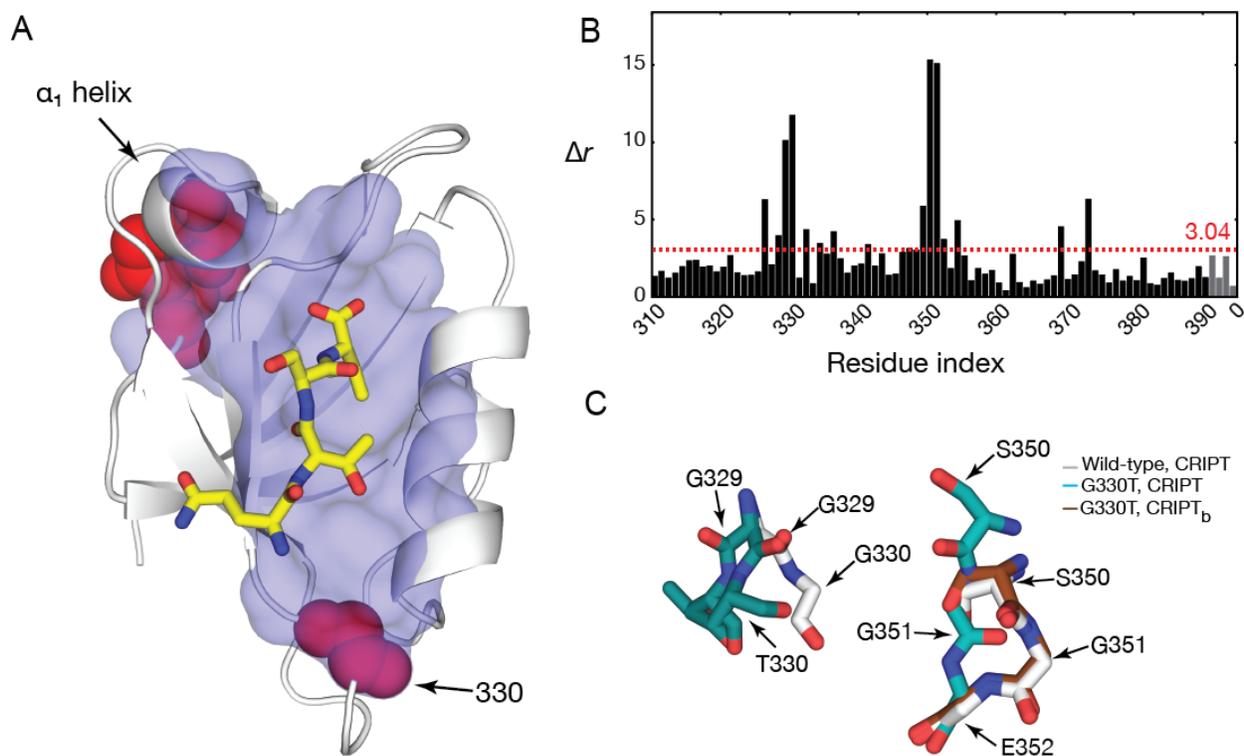


Figure 9. A. The G330T mutation to the β_2 - β_3 loop causes conformational rearrangement at a distant site, the α_1 helix. **B.** The error-normalized Δr reveals this pattern quantitatively. The four C-terminal residues of the ligand are depicted in grey. The red dashed line represents the 80th percentile cutoff on the ECDF, above which shifts are considered significant. The largest differences between the structures are located at the β_2 - β_3 loop and the α_1 helix. **C.** Stick representation of the residues which undergo the greatest displacements between the two structures. Positions 329 and 330 of the β_2 - β_3 loop are found in a single conformation in the structure of the wild-type protein bound to CRIPT, while they occupy a split conformation in the case of the G330T protein bound to CRIPT. Positions 350–352 are found in a single conformation in the case of wild-type bound to CRIPT, but they are split between the wild-type conformation and a second one in the case of G330T bound to CRIPT. This arises primarily through a shift in the position of G351.

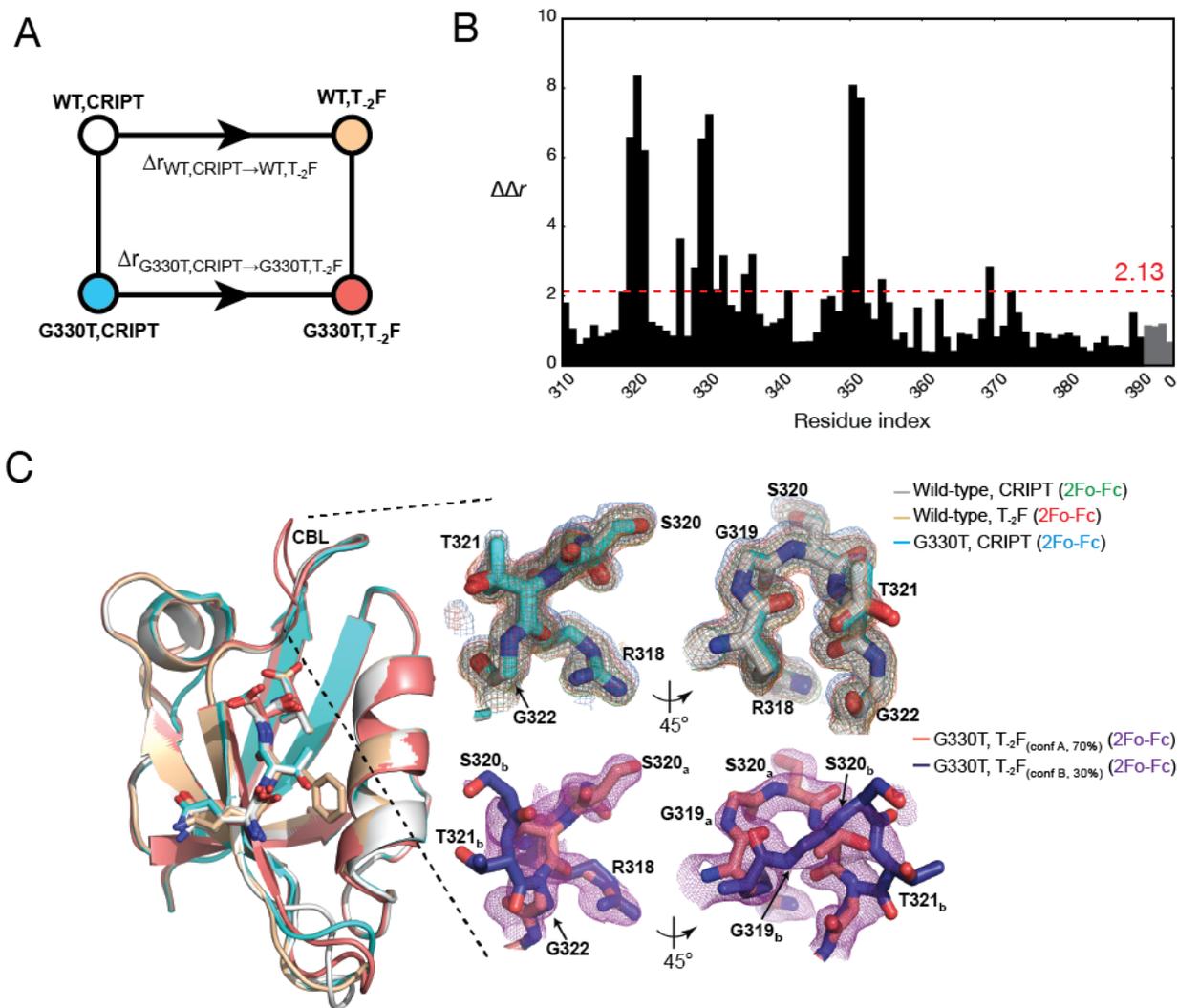


Figure 10. **A.** The second-order thermodynamic cycle corresponding to the two mutations, G330T and T₂F, chosen to probe structural coupling in PSD-95 PDZ3. **B.** The distribution of error-normalized structural coupling, $\Delta\Delta r$, over the cycle depicted in **A**. Significant heterogeneity is identified in three regions—the carboxylate binding loop (CBL), the β_2 - β_3 loop, and the α_1 helix. Ligand residues are depicted in grey, and the 80th percentile of the ECDF is drawn as a dashed red line. **C.** While signal corresponding to the the β_2 - β_3 loop and the α_1 helix arises from differences between the wild-type and G330T structures bound to the CRIPT ligand as discussed previously, the conformational heterogeneity arises from the inclusion of the structures bound to the mutant T₂F ligand. This arises in from the structure of the G330T mutant bound to the T₂F peptide, where residues 319–321 occupy a second low-occupancy “unclamped” conformation in addition to the standard “clamped” conformation.

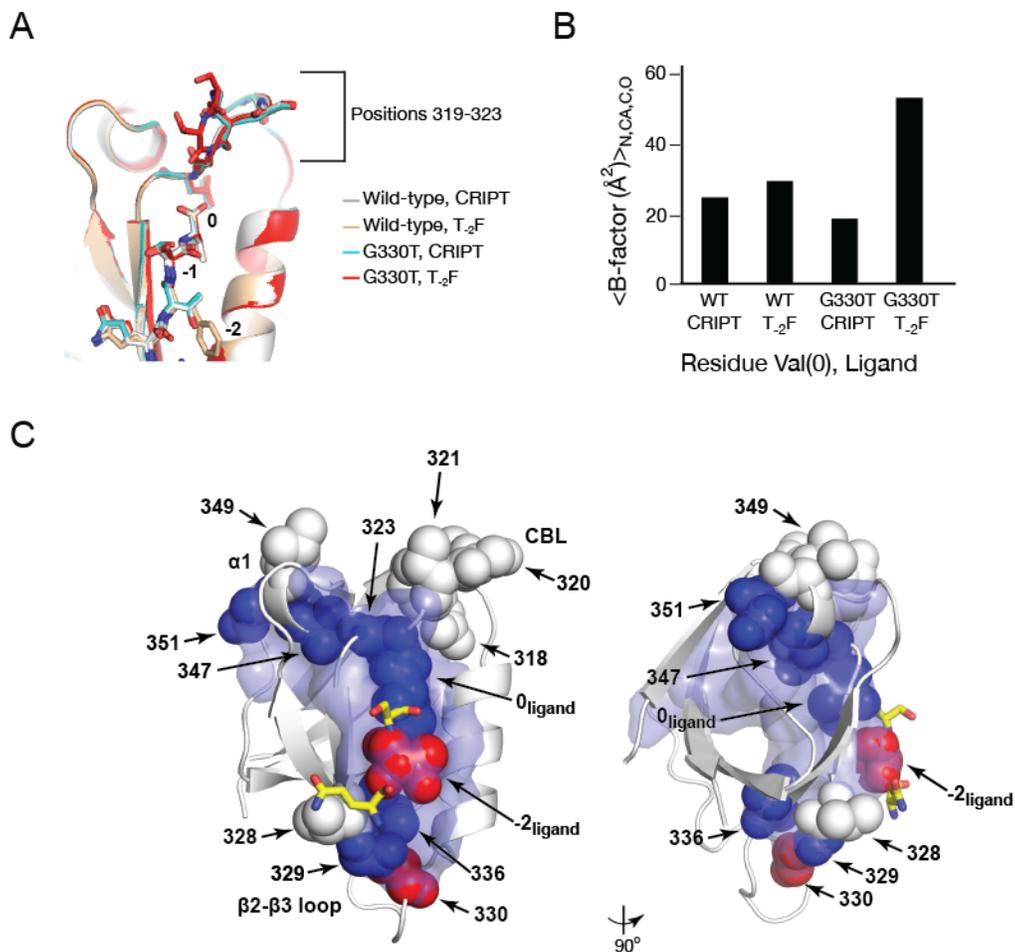


Figure 11. **A.** The ligand C-terminus interacts with the end of the carboxylate binding loop through a series of hydrogen bonding interactions with the conserved GLGF motif. While the conformations of the ligand C-terminus are the same in all four structures, the CBL partially occupies an “unclamped” conformation in the case of the structure of G330T bound to T₂F. **B.** This heterogeneity could be a consequence of increased disorder of V0 backbone atoms, as judged by crystallographic *B*-factors. **C.** Residues with $\Delta\Delta r \geq 2.13$ (80th percentile of the ECDF) are shown as spheres on the structure of PSD-95 PDZ3. Spheres are colored blue or white depending on whether a given residue is inside or outside the sector, respectively. The sites of mutation are depicted in red. This representation illustrates a pattern of distributed, anisotropic structural coupling running along the ligand binding cleft between the β_2 - β_3 loop, the α_1 helix, and the CBL.

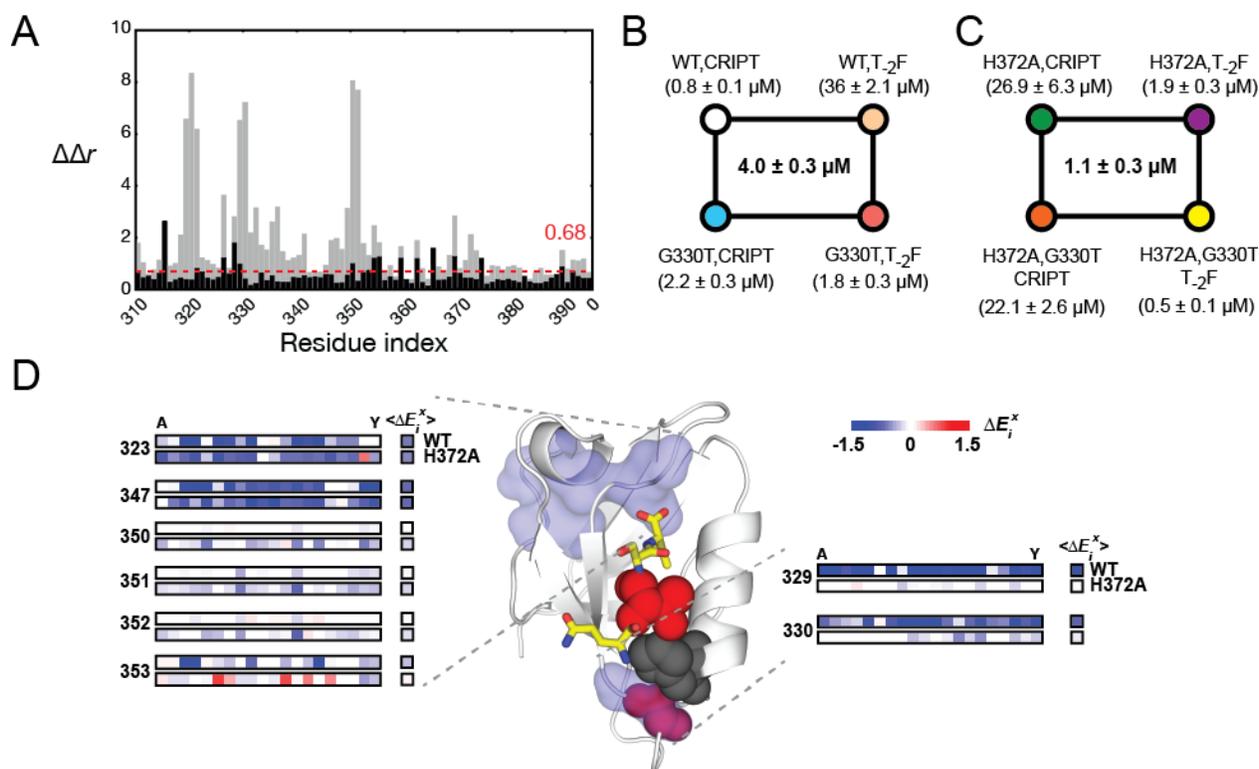


Figure 12. A. Inclusion of the H372A mutation in the second-order structural cycle (**B**, **C**) disrupts long-range coupling between the β_2 - β_3 loop, the α_1 helix, and the carboxylate binding loop (CBL). The $\Delta\Delta r$ distribution for the cycle including the H372A mutation is shown in black, while the $\Delta\Delta r$ distribution without the H372A mutation is shown in grey for reference. The 80th percentile of the ECDF for the distribution including the H372A mutation is shown as a red dashed line. In the background of H372A, structural displacements are largely additive and sum to near-zero values. **D.** The mutational sensitivity of the H372A mutant for CRIPT binding in two regions of the PDZ3 protein. Matrices show the relative enrichment measured using the bacterial two-hybrid assay for all possible amino acid substitutions at a series of positions. With the exception of position 353, mutations at and around the α_1 helix show similar patterns of sensitivity in either the wild-type or H372A background. The wild-type and H372A mutant proteins show differential sensitivity to mutations in the β_2 - β_3 loop, however. Mutations in the wild-type background are largely deleterious with respect to CRIPT binding, while mutations in the H372A background have little effect on CRIPT binding.

6 Tables

Unselected Library Statistics

	<u>Number of reads</u>	<u>Number of Ligands (> 50 counts)</u>	<u>% Library Coverage</u>
Total Input Library	1.07 x 10 ⁸	154,521	96.7

Selected Library Statistics

	<u>Number of reads</u>	<u>Number of Ligands</u>	<u>Number of Ligands (> 50 counts)</u>	<u>Number of Ligands Bound by Protein (> 15 μM)</u>
WT	46,598,840	56,640	55,278	188
G330T	51,195,397	90,735	83,056	854
H372A	29,419,042	59,935	43,488	1053
H372A/ G330T	48,989,769	123,295	86,255	1902

Table 1. Sequencing statistics from Illumina HiSeq2500 runs for the unselected and selected populations of peptide libraries for wild-type, G330T, H372A, and the double-mutant experiments. The unselected populations were combined over all experiments. The total number of reads, (1.07×10^8) represented approximately 97% coverage of all peptides in the library.

	WT-Apo			WT-CRIPT			WT-T7F											
PDB ID	5HDY			5HEB			5HED											
Source	APS 19-ID			UTSW SBL			UTSW SBL											
Wavelength (Å)	0.97918			1.54178			1.54178											
Resolution range (Å)	40.1–1.801 (1.866–1.801)			40.15–1.65 (1.709–1.65)			31.66–1.7 (1.761–1.7)											
Space group	P 41 3 2			P 41 3 2			P 41 3 2											
Unit cell (Å, °)	89.657	89.657	89.657	90	90	90	89.771	89.771	89.771	90	90	90	89.551	89.551	89.551	90	90	90
Total reflections	106607			209981			128220											
Unique reflections	11131 (693)			15083 (1156)			13972 (1299)											
Multiplicity	9.1 (8.1)			13.9 (2.5)			9.2 (3.0)											
Completeness	0.93			0.98			0.99											
Mean I/σ	52.621 (1.313)			61.571 (2.478)			60.017 (2.627)											
Wilson B-factor	19.75			14.71			15.35											
R-merge	0.035 (N/A)			0.034 (0.289)			0.031 (0.302)											
R-meas	0.036 (N/A)			0.034 (0.353)			0.033 (0.361)											
R-pim	0.012 (0.440)			0.008 (0.199)			0.010 (0.193)											
CC1/2	0.597			0.870			0.885											
Reflections used in refinement	11130 (693)			15082 (1156)			13971 (1299)											
Reflections used for R-free	1114 (70)			1509 (116)			1398 (130)											
R-work	0.1865 (0.2900)			0.1666 (0.2273)			0.1701 (0.2291)											
R-free	0.2199 (0.2895)			0.1945 (0.2745)			0.1973 (0.2694)											
Number of non-hydrogen atom	1223			1220			1262											
Macromolecules	1099			1086			1110											
Ligands	6			12			N/A											
Protein residues	117			126			126											
RMS (bonds)	0.010			0.007			0.014											
RMS (angles)	1.16			1.22			1.46											
Ramachandran favored (%)	98			97			99											
Ramachandran allowed (%)	0.71			2.1			0.72											
Ramachandran outliers (%)	1.4			0.71			0											
Rotamer outliers (%)	1.7			2.6			3.4											
Clashscore	5.81			3.19			7.29											
Average B-factor	27.34			19.78			20.48											
Macromolecules	26.41			18.51			19.21											
Ligands	54.54			62.39			N/A											
Solvent	34.65			26.84			29.79											
Number of TLS groups	7			3			3											

Table 2. Crystallographic statistics.

	G330T-Apo	G330T-CRIPT	G330T-T7F
PDB ID	5HET	5HEY	5HF1
Source	APS 19-ID	APS 19-ID	APS 19-ID
Wavelength (Å)	0.97937	0.97937	0.97918
Resolution range (Å)	36.52–2.001 (2.073–2.001)	27.21–1.5 (1.554–1.5)	40.13–1.747 (1.81–1.747)
Space group	P 41 3 2	P 41 3 2	P 41 3 2
Unit cell (Å, °)	89.445 89.445 89.445 90 90 90	90.23 90.23 90.23 90 90 90	89.724 89.724 89.724 90 90 90
Total reflections	87993	454547	358023
Unique reflections	8697 (805)	20265 (1645)	13007 (1217)
Multiplicity	10.0 (8.4)	21.9 (14.7)	27.3 (26.3)
Completeness	0.99	0.98	0.99
Mean I/σ	35.095 (2.077)	30.057 (1.448)	67.804 (1.844)
Wilson B-factor	25.36	14.99	19.04
R-merge	0.064 (N/A)	0.110 (N/A)	0.055 (N/A)
R-meas	0.068 (N/A)	0.112 (N/A)	0.056 (N/A)
R-pim	0.022 (0.397)	0.035 (0.665)	0.013 (0.581)
CC1/2	0.713	0.555	0.721
Reflections used in refinement	8696 (805)	20265 (1645)	13006 (1217)
Reflections used for R-free	871 (80)	2024 (165)	1301 (122)
R-work	0.1837 (0.2309)	0.1724 (0.2705)	0.1993 (0.2911)
R-free	0.2130 (0.2586)	0.2026 (0.2765)	0.2179 (0.3097)
Number of non-hydrogen atom	1054	1268	1182
Macromolecules	956	1121	1080
Ligands	N/A	N/A	N/A
Protein residues	118	127	124
RMS (bonds)	0.006	0.011	0.006
RMS (angles)	0.83	1.84	0.94
Ramachandran favored (%)	98	95	99
Ramachandran allowed (%)	1.6	4.2	0.73
Ramachandran outliers (%)	0	0.69	0
Rotamer outliers (%)	2.9	3.3	4.4
Clashscore	1.06	4.02	4.63
Average B-factor	32.04	22.33	28.71
Macromolecules	31.36	21.16	27.95
Ligands	N/A	N/A	N/A
Solvent	38.7	31.27	36.72
Number of TLS groups	1	16	5

Table 3. Crystallographic statistics, continued.

	H372A-Apo			H372A-CRIPT			H372A-T7F					
PDB ID	5HF4			5HFB			5HFC					
Source	APS 19-ID			UTSW SBL			UTSW SBL					
Wavelength (Å)	0.97918			1.54178			1.54178					
Resolution range (Å)	39.92–1.75 (1.813–1.75)			40.07–1.617 (1.675–1.617)			36.52–1.851 (1.918–1.851)					
Space group	P 41 3 2			P 41 3 2			P 41 3 2					
Unit cell (Å, °)	89.254	89.254	89.254	90	90	90	89.445	89.445	89.445	90	90	90
Total reflections	146556			203441			68030					
Unique reflections	12757 (1200)			15108 (556)			10759 (914)					
Multiplicity	11.4 (11.0)			13.3 (1.6)			6.3 (2.5)					
Completeness	1.00			0.93			0.98					
Mean I/σ	48.917 (2.286)			65.515 (0.936)			35.543 (1.000)					
Wilson B-factor	19.03			15.74			21.65					
R-merge	0.045 (N/A)			0.035 (0.338)			0.040 (0.448)					
R-meas	0.047 (N/A)			0.036 (0.470)			0.044 (0.547)					
R-pim	0.014 (0.336)			0.009 (0.325)			0.017 (0.304)					
CC1/2	0.736			0.695			0.689					
Reflections used in refinement	12757 (1200)			15109 (556)			10759 (914)					
Reflections used for R-free	1279 (123)			1513 (56)			1078 (92)					
R-work	0.1823 (0.2573)			0.1751 (0.3004)			0.1835 (0.2894)					
R-free	0.2207 (0.2900)			0.2033 (0.3281)			0.2051 (0.3146)					
Number of non-hydrogen atom	1158			1204			1161					
Macromolecules	1041			1055			1037					
Ligands	6			N/A			N/A					
Protein residues	118			123			124					
RMS (bonds)	0.005			0.006			0.012					
RMS (angles)	0.93			0.95			1.46					
Ramachandran favored (%)	98			99			95					
Ramachandran allowed (%)	2.2			0.73			4.5					
Ramachandran outliers (%)	0			0			0.76					
Rotamer outliers (%)	1.8			1.8			2.7					
Clashscore	1.42			1.91			10.18					
Average B-factor	28.65			22.31			30.76					
Macromolecules	28.09			20.99			30.12					
Ligands	58.33			N/A			N/A					
Solvent	32.32			31.72			36.13					
Number of TLS groups	4			5			8					

Table 4. Crystallographic statistics, continued.

	G330T-H372A-Apo						G330T-H372A-CRIPT						G330T-H372A-T7F					
PDB ID	5HFD						5HFE						5HFF					
Source	APS 19-ID						UTSW SBL						APS 19-ID					
Wavelength (Å)	0.97918						1.54178						0.97918					
Resolution range (Å)	26.92–1.6 (1.657–1.6)						40.14–1.8 (1.864–1.8)						36.57–1.749 (1.812–1.749)					
Space group	P 41 3 2						P 41 3 2						P 41 3 2					
Unit cell (Å, °)	89.281	89.281	89.281	90	90	90	89.751	89.751	89.751	90	90	90	89.575	89.575	89.575	90	90	90
Total reflections	257150						217694						264373					
Unique reflections	16607 (1608)						11944 (1113)						12879 (1179)					
Multiplicity	15.4 (15.7)						18.1 (3.9)						20.3 (20.1)					
Completeness	1.00						0.99						0.99					
Mean I/σ	63.192 (2.111)						50.539 (1.815)						50.222 (1.826)					
Wilson B-factor	13.82						20.84						18.64					
R-merge	0.043 (N/A)						0.061 (0.594)						0.063 (N/A)					
R-meas	0.043 (N/A)						0.063 (0.681)						0.065 (N/A)					
R-pim	0.011 (0.308)						0.014 (0.320)						0.016 (0.478)					
CC1/2	0.772						0.705						0.706					
Reflections used in refinement	16605 (1608)						11944 (1113)						12880 (1179)					
Reflections used for R-free	1660 (160)						1195 (112)						1285 (114)					
R-work	0.1810 (0.2423)						0.1725 (0.2786)						0.1740 (0.2314)					
R-free	0.1967 (0.2670)						0.2096 (0.3359)						0.2233 (0.2761)					
Number of non-hydrogen atom	1203						1190						1191					
Macromolecules	1075						1066						1080					
Ligands	N/A						N/A						12					
Protein residues	119						127						128					
RMS (bonds)	0.014						0.012						0.011					
RMS (angles)	1.26						1.31						1.27					
Ramachandran favored (%)	99						96						97					
Ramachandran allowed (%)	0.7						2.9						0.71					
Ramachandran outliers (%)	0						1.5						2.1					
Rotamer outliers (%)	1.7						5.3						6.2					
Clashscore	2.80						7.58						4.16					
Average B-factor	21.51						26.19						25.02					
Macromolecules	20.58						25.33						24.07					
Ligands	N/A						N/A						57.25					
Solvent	29.28						33.97						31.53					
Number of TLS groups	10						6						7					

Table 5. Crystallographic statistics, continued.

Protein	[sodium citrate] (M)	crystallization buffer pH	[protein] (mg/ml)
WTapo	1.0	6.9	9
WTCRIPT	1.0	7.0	9
WTT-2F	1.125	7.1	9
G330Tapo	1.05	7.0	7
G330TCRIPT	1.2	7.0	8
G330TT-2F	1.2	6.8	9
H372Aapo	0.95	7.0	9
H372ACRIPT	1.05	7.0	7
H372AT-2F	1.05	7.0	7
G330T-H372Aapo	1.0	7.0	13
G330T-H372ACRIPT	1.25	7.0	9
G330T-H372AT-2F	1.2	6.75	7

Table 6. Crystal growth conditions for each PSD-95 PDZ3 variant.

7 Works cited

Some text and many figures are reproduced here as presented in an as-of-yet unpublished manuscript entitled “Origins of allostery and evolvability in proteins: a case study” by Arjun Raman, myself, and Rama Ranganathan, and in the thesis of Arjun Raman. For more detailed exploration of some of the topics discussed in this chapter, Arjun’s thesis is an excellent resource.

1. Bowie, J. U., Clarke, N. D., Pabo, C. O. & Sauer, R. T. Identification of protein folds: matching hydrophobicity patterns of sequence sets with solvent accessibility patterns of known structures. *Proteins* **7**, 257–264 (1990).
2. Halabi, N., Rivoire, O., Leibler, S. & Ranganathan, R. Protein sectors: evolutionary units of three-dimensional structure. *Cell* **138**, 774–786 (2009).
3. Morcos, F. *et al.* Direct-coupling analysis of residue coevolution captures native contacts across many protein families. *Proc Natl Acad Sci USA* **108**, E1293–301 (2011).
4. Süel, G. M., Lockless, S. W., Wall, M. A. & Ranganathan, R. Evolutionarily conserved networks of residues mediate allosteric communication in proteins. *Nat Struct Biol* **10**, 59–69 (2003).
5. Anfinsen, C. B. Principles that govern the folding of protein chains. *Science* **181**, 223–230 (1973).
6. Bershtein, S., Segal, M., Bekerman, R., Tokuriki, N. & Tawfik, D. S. Robustness-epistasis link shapes the fitness landscape of a randomly drifting protein. *Nature* **444**, 929–932 (2006).
7. Smith, J. M. Natural selection and the concept of a protein space. *Nature* **225**, 563–564 (1970).
8. Tokuriki, N. & Tawfik, D. S. Protein Dynamism and Evolvability. *Science* **324**, 203–207 (2009).
9. Wagner, A. Robustness, evolvability, and neutrality. *FEBS Lett.* **579**, 1772–1778 (2005).
10. Aakre, C. D. *et al.* Evolving new protein-protein interaction specificity through promiscuous intermediates. *Cell* **163**, 594–606 (2015).
11. Hayden, E. J., Ferrada, E. & Wagner, A. Cryptic genetic variation promotes rapid evolutionary adaptation in an RNA enzyme. *Nature* **474**, 92–95 (2011).
12. Draghi, J. A., Parsons, T. L., Wagner, G. P. & Plotkin, J. B. Mutational robustness can facilitate adaptation. *Nature* **463**, 353–355 (2010).
13. Draghi, J. A. & Plotkin, J. B. Molecular evolution: Hidden diversity sparks adaptation. *Nature* **474**, 45–46 (2011).
14. Luria, S. E. & Delbrück, M. Mutations of Bacteria from Virus Sensitivity to Virus Resistance. *Genetics* **28**, 491–511 (1943).
15. Gould, S. J. & Vrba, E. S. Exaptation—a Missing Term in the Science of Form. *Paleobiology* **8**, 4–15 (2016).
16. Monod, J., Wyman, J. & Changeux, J. On Nature of Allosteric Transitions — a Plausible Model. *J Mol Biol* **12**, 88–118 (1965).

17. KOSHLAND, D. E., Némethy, G. & Filmer, D. Comparison of experimental binding data and theoretical models in proteins containing subunits. *Biochemistry* **5**, 365–385 (1966).
18. Smock, R. G. & Gierasch, L. M. Sending signals dynamically. *Science* **324**, 198–203 (2009).
19. Fraser, J. S. *et al.* Hidden alternative structures of proline isomerase essential for catalysis. *Nature* **462**, 669–U149 (2009).
20. Cooper, A. & Dryden, D. T. F. Allostery without conformational change. *Eur Biophys J* **11**, 103–109 (1984).
21. Frederick, K. K., Marlow, M. S., Valentine, K. G. & Wand, A. J. Conformational entropy in molecular recognition by proteins. *Nature* **448**, 325–329 (2007).
22. Lockless, S. W. & Ranganathan, R. Evolutionarily conserved pathways of energetic connectivity in protein families. *Science* **286**, 295–299 (1999).
23. Shulman, A. I., Larson, C., Mangelsdorf, D. J. & Ranganathan, R. Structural determinants of allosteric ligand activation in RXR heterodimers. *Cell* **116**, 417–429 (2004).
24. Ferguson, A. D. *et al.* Signal transduction pathway of TonB-dependent transporters. *Proc Natl Acad Sci USA* **104**, 513–518 (2007).
25. Reynolds, K. A., McLaughlin, R. N. & Ranganathan, R. Hot Spots for Allosteric Regulation on Protein Surfaces. *Cell* **147**, 1564–1575 (2011).
26. Whitney, D. S. *et al.* Crumbs binding to the Par-6 CRIB-PDZ module is regulated by Cdc42. *Biochemistry* [acs.biochem.5b01342](https://doi.org/10.1021/acs.biochem.5b01342) (2016). doi:10.1021/acs.biochem.5b01342
27. Songyang, Z. *et al.* Recognition of unique carboxyl-terminal motifs by distinct PDZ domains. *Science* **275**, 73–77 (1997).
28. Stiffler, M. A. *et al.* PDZ domain binding selectivity is optimized across the mouse proteome. *Science* **317**, 364–369 (2007).
29. Niethammer, M. *et al.* CRIPT, a novel postsynaptic protein that binds to the third PDZ domain of PSD-95/SAP90. *Neuron* **20**, 693–707 (1998).
30. McLaughlin, R. N., Poelwijk, F. J., Raman, A., Gosal, W. S. & Ranganathan, R. The spatial architecture of protein function and adaptation. *Nature* **491**, 138–142 (2012).
31. Doyle, D. A. *et al.* Crystal structures of a complexed and peptide-free membrane protein-binding domain: molecular basis of peptide recognition by PDZ. *Cell* **85**, 1067–1076 (1996).
32. Sayou, C. *et al.* A promiscuous intermediate underlies the evolution of LEAFY DNA binding specificity. *Science* **343**, 645–648 (2014).
33. Raman, A. Origins of allostery and evolvability in proteins: a case study. 1–129 (2015).
34. Jain, R. K. & Ranganathan, R. Local complexity of amino acid interactions in a protein core. *Proc Natl Acad Sci USA* **101**, 111–116 (2004).
35. Stroud, R. M. & Fauman, E. B. Significance of structural changes in proteins: expected errors in refined protein structures. *Protein Sci* **4**, 2392–2404 (1995).
36. Hedstrom, L., Szilagyi, L. & Rutter, W. J. Converting trypsin to chymotrypsin: the role of surface loops. *Science* **255**, 1249–1253 (1992).
37. Perona, J. J., Hedstrom, L., Rutter, W. J. & Fletterick, R. J. Structural origins of substrate discrimination in trypsin and chymotrypsin. *Biochemistry* **34**, 1489–1499 (1995).
38. Lukacs, C. M., Kucera, R., Schildkraut, I. & Aggarwal, A. K. Understanding the immutability of restriction enzymes: crystal structure of BglII and its DNA substrate at 1.5 Å resolution. *Nat Struct Biol* **7**, 134–140 (2000).
39. Morgan, R. D. & Luyten, Y. A. Rational engineering of type II restriction endonuclease

- DNA binding and cleavage specificity. *Nucleic Acids Res.* **37**, 5222–5233 (2009).
40. Poelwijk, F. J., de Vos, M. G. J. & Tans, S. J. Tradeoffs and optimality in the evolution of gene regulation. *Cell* **146**, 462–470 (2011).
 41. Stiffler, M. A., Hekstra, D. R. & Ranganathan, R. Evolvability as a function of purifying selection in TEM-1 β -lactamase. *Cell* **160**, 882–892 (2015).
 42. Smock, R. G. *et al.* An interdomain sector mediating allostery in Hsp70 molecular chaperones. *Mol Syst Biol* **6**, 414 (2010).
 43. Lee, J. *et al.* Surface sites for engineering allosteric control in proteins. *Science* **322**, 438–442 (2008).
 44. Esvelt, K. M., Carlson, J. C. & Liu, D. R. A system for the continuous directed evolution of biomolecules. *Nature* **472**, 499–503 (2011).
 45. Ota, N. & Agard, D. A. Intramolecular signaling pathways revealed by modeling anisotropic thermal diffusion. *J Mol Biol* **351**, 345–354 (2005).
 46. Petit, C. M., Zhang, J., Sapienza, P. J., Fuentes, E. J. & Lee, A. L. Hidden dynamic allostery in a PDZ domain. *Proc Natl Acad Sci USA* **106**, 18249–18254 (2009).
 47. Studier, F. W. Protein production by auto-induction in high density shaking cultures. *Protein Expr. Purif.* **41**, 207–234 (2005).
 48. Otwinowski, Z. & Minor, W. in *Macromolecular Crystallography Part A* **276**, 307–326 (Elsevier, 1997).
 49. Adams, P. D. *et al.* PHENIX: a comprehensive Python-based system for macromolecular structure solution. *Acta Crystallogr D Biol Crystallogr* **66**, 213–221 (2010).
 50. Emsley, P., Lohkamp, B., Scott, W. G. & Cowtan, K. Features and development of Coot. *Acta Crystallogr D Biol Crystallogr* **66**, 486–501 (2010).
 51. Painter, J. & Merritt, E. A. Optimal description of a protein structure in terms of multiple groups undergoing TLS motion. *Acta Crystallogr D Biol Crystallogr* **62**, 439–450 (2006).
 52. Painter, J. & Merritt, E. A. TLSMDweb server for the generation of multi-group TLS models. *J Appl Crystallogr* **39**, 109–111 (2006).
 53. Davis, I. W. *et al.* MolProbity: all-atom contacts and structure validation for proteins and nucleic acids. *Nucleic Acids Res.* **35**, W375–83 (2007).

Chapter III

Perturbing with temperature: Time-averaged dynamics of a protein family

1 Introduction

The motions of proteins have been implicated in a variety of essential biological processes¹. Large-scale motions of the nascent polypeptide chain amount to a conformational search in folding space², while less dramatic motions are often associated with protein function, either directly through substrate binding³ or the coordination of catalysis^{4,5} or indirectly as in the case of allostery⁶⁻⁹. Despite the abundance of such work, however, general mechanical models describing the motions of proteins remain simplistic and mostly anecdotal.

What features would characterize such a model? Ideally, it would provide a framework from which to understand—in atomic detail—the conformational states of proteins, the energetics of those states, and any motions linking them. Unfortunately, however, no current experimental approach can provide an experimental foundation from which to address these questions. Nuclear magnetic resonance (NMR) methods¹⁰ and even single-molecule FRET¹¹ can provide information about the conformational exchange of proteins, but only over specific timescales and with limited spatial resolution. In principle, time-resolved X-ray crystallography (TRX) can directly report both the timescales of motion and the energies associated with them, but it has only been fully developed to date for the study of a very specific subset of proteins—primarily those with photoactivatable chromophores¹²⁻¹⁴. A recent generalization of TRX in which strong, pulsed electric fields are used to directly drive protein motions has been described (EFX; see Chapter 4)

and shows great promise, but substantial research and development remains before it can be used as a turnkey method.

A growing body of work suggests that the foundations of a more general model for protein mechanics be accessible—at least in part—through careful analysis of disorder in conventional X-ray diffraction data. While a crystal lattice certainly places restraints on the motions that a protein molecule can undergo, both continuous and discrete conformational heterogeneity has long been observed in crystal structures¹⁵. The observed disorder is the convolution of atomic positions in time (dynamic disorder), which arises from atomic vibrations and collective motions over the course of data collection, and in space (static disorder) which results from averaging conformations over millions of molecules and crystal lattice defects.

In practice, both forms of disorder are conventionally parameterized by the so-called *B*-factor, which describes the spatial extent of decay in electron density about the position of a given atom. Ideally, the *B*-factor reports solely on the degree of harmonic disorder for an atom and is related to the probability of observing it at a given location, but in practice it often includes a variety of additional contributions related to anharmonic and/or anisotropic disorder and model error while potentially underestimating true disorder^{16,17}.

Recent work provides several avenues for detecting and modeling conformational heterogeneity more directly, including qFit^{18,19}, Ringer²⁰, and ensemble refinement²¹. For the purposes of inferring correlated conformational states, ensemble refinement represents a promising approach. Rather than building a single model with multiple minor conformations, ensemble refinement relies on a data-constrained molecular dynamics (MD) simulation with time-averaged restraints to generate a large population, or ensemble, of models in accordance with a Boltzmann-

distribution^{21,22}. An important consequence of this approach is that (assuming adequate sampling) the resulting ensemble of structures is essentially drawn from a maximum entropy distribution—a distribution of models approximately consistent with data and force field but with no additional statistical structure introduced²³. As a result, probability distributions can be calculated for a variety of model features—for example, the positions of atoms in space or the dihedral angles characteristic of bonds—and conformational heterogeneity can be quantitatively assessed.

In the limit that these probability distributions are Gaussian-distributed (i.e., the motions of atoms are purely harmonic), correlations between atoms or residues can be calculated directly, and the resulting statistical structure compared to any number of first-principle models describing protein dynamics, such as elastic network models^{24,25}. Numerous studies have demonstrated, however, that functionally relevant motions are often anharmonic in nature and involve multiple potential wells separated by substantial energy barriers. As a result, two atoms might move together consistently, albeit non-linearly, and the correlation coefficient relating the motion would be underestimated. An alternative approach based on the calculation of mutual information (MI) between atoms, formalized by Lange and Grubmüller²⁶ and employed in the search for allosteric pathways over molecular dynamics trajectories by McClendon et. al.²⁷, provides a more general method for identifying collective atomic fluctuations even in cases where motions are highly non-linear.

The analysis of mutual information between residues over ensembles thus represents a promising approach for extracting collective fluctuations consistent with experimental data. However, as with any other method for studying protein dynamics, the relevance of observed coupling with respect to function can only be inferred. For a crystalline protein, collective fluctuations may arise from any number of sources—the topology of the fold, the functional niche

occupied by the protein and any idiosyncratic features associated with it, the crystal lattice and the constraints on motion it might impose, experimental error, etc. *A priori*, these contributions cannot be easily separated. The principle of evolutionary conservation provides a powerful solution to this problem. For an observed collective feature to be of fundamental “importance,” that is, relevant to the function of the protein and thus to the fitness of the organism from which it originates, it must appear repeatedly over evolutionarily related proteins with similar folds carrying out similar functions—that is, over homologous proteins.

To implement this notion, we obtained high-resolution X-ray diffraction data sets for nine different PDZ domains. PDZ (Post-synaptic density, Discs large, Zona occludens) domains are small (~90 amino acids), modular protein-protein interaction domains which typically bind the C-termini of other proteins^{28,29}. The typical PDZ domain is composed of a six-strand β -sandwich decorated with a pair of α helices, and the C-termini of binding partners typically associate by β -sheet augmentation in a groove between the β_2 strand and the α_2 helix. Ligand binding is often accompanied by conformational changes^{30,31}. In addition, allosteric modulation of ligand affinity has been demonstrated for a number of PDZ domains^{32,33}. These observations suggest an important role for collective motions in PDZ function.

To extract correlated fluctuations from the PDZ domain data sets, we performed ensemble refinement, followed by a newly developed statistical analysis of the resulting conformational ensembles. First, we show that ensemble refinement is robust and generates consistent patterns of MI between amino acids over multiple diffraction datasets for one of the PDZ domains, PSD-95 PDZ3—a model system for understanding protein dynamics and allostery^{34,35}. We then show that conformational correlations reflect both local contacts in tertiary structure and a few global modes. Although these correlated motions vary from homolog to homolog, we demonstrate a clear,

substantial conservation of correlated motions across homologs, providing evidence that ensemble refinement can indeed reveal correlated motions of fundamental importance to protein function.

2 Results and discussion

2.1 Consistent patterns of conformational heterogeneity can be extracted from independent X-ray datasets collected from PSD-95 PDZ3 protein crystals.

We first sought to assess the robustness of the ensemble refinement algorithm over independent datasets. To this end, we collected four high-resolution room-temperature datasets from independent crystals of PSD-95 PDZ3 bound to its cognate peptide ligand, derived from cysteine-rich interactor of PDZ (CRIPT)^{30,36} (Figure 1 A) in accordance with current best practices. The datasets were highly isomorphous ($a = b = c = 90.86 \pm 0.14 \text{ \AA}$, $\alpha = \beta = \gamma = 90^\circ$), and of sufficiently high resolution to model individual atoms ($r_{\text{max}} = 1.338 \pm 0.007 \text{ \AA}$). Additional data collection statistics are summarized in Table I (sets 1-1, 2-1, 3-1, 4-1).

Following data reduction, we built a total of six PSD-95 PDZ3-CRIPT models using conventional methods, starting from model 1BE9 (Table I, refinement statistics). Two additional models were built from dataset 4, but with different test sets excluded for the calculation of R_{free} (sets 4-2, 4-3). Almost all protein and ligand atoms could be resolved, and numerous alternative conformations were placed in accordance with the difference electron density. Global radiation damage was limited (Table I), and no evidence of specific radiation damage (e.g., decarboxylation of aspartates or glutamates), was observed in the electron density.

These models were used as input for two rounds of ensemble refinement. Briefly, a grid search was performed to identify optimal values of two key ensemble refinement parameters, T_{offset} and p_{TLS} , as assessed by crystallographic R -factors; optimal values of T_{offset} ranged from 0.75–12, while p_{TLS} ranged from 0.3–0.8 (Table I, ensemble refinement statistics). For each dataset, 124 single-

conformer models were generated, with alternative conformations chosen at random with probability proportional to occupancy in the original model. Separate ensemble refinements were then performed on the models to yield 124 ensembles with 550 models each (Figure 1 B). In general, refinement statistics for these models are good, although crystallographic R -factors are typically a few percent larger for ensembles than for starting models (Table I, ensemble refinement statistics). Finally, we excluded as many as 20% of ensembles based on deviation of R_{work} distributions over ensembles from normality in order to exclude outlier conformations not in accordance with the data (Figure 2).

To calculate all pairwise coupling between residues in each super-ensemble, we combined all retained models for each dataset to yield six PSD-95 PDZ3-CRIPT “super-ensembles” composed of 107 ± 7 ensembles with a total of $58,942 \pm 3,803$ superposed models each—well in excess of necessity as assessed by random sampling (Figure 3)—and calculated the mutual information (MI) between all pairs of atoms. Naïve inspection of atomic MI leads to several observations. First, the global pattern of coupling is heterogeneous, and distributed in a non-trivial way throughout the structure, with strong signal at and near the diagonal corresponding to coupling between covalently linked atoms or atoms in adjacent residues (Figure 4). Coupling is generally dependent on the distance between atoms (Figure 4), and striking chains of MI can be observed to propagate throughout the protein structure, often through hydrogen-bonded backbone atoms (Figure 5 A). This observation is consistent with expectation—long-range coupling must propagate in a physics-dependent manner—and illustrates the value of our approach in discovering such patterns. Additionally, the use of MI is further motivated by examples where the distributions of atomic coordinates are complex and non-Gaussian (Figure 5 B, C); in some cases, a more traditional

approach based on analysis of correlation coefficients would substantially underestimate the degree of coupling (Figure 5 B).

To gain further intuition about the types of interactions contributing to these conformational correlations, consider the identities of the atoms involved. Certain types of atomic interactions appear to carry more MI than others. Analysis of the distribution of MI as a function of interacting pair identity (Figure 4 A) leads to several interesting conclusions. First, we observe a reasonably strong relationship between the distance between any two atoms in the protein and the MI that characterizes their interaction (Figure 4 B). This relationship appears exponential, but does not vanish at large distances, illustrating the long-range nature of some atomic couplings in the protein. Second, coupling between backbone atoms (C, C_α, N, O) contributes a greater portion of MI than expected given the number of backbone atoms in PSD-95 PDZ3-CRIPT; specifically, 28% of total pairs are between backbone atoms and contribute 35% of total MI. Interactions between backbone and sidechain atoms follow a similar but weak trend, as they comprise 48% of the total pairs and contribute 50% of the total MI. Analysis of pairs of interacting sidechain atoms reveals the opposite trend; despite accounting for 22% of pairs, sidechain-sidechain interactions contribute only 17% of total MI. While these differences are small, they suggest that, on average, backbone atoms are more likely to serve as conduits in the propagation of physical coupling than sidechain atoms, at least when considering the contributions of core and surface side-chains to be equivalent. What types of atom pairs are the most coupled? Analysis of the average MI as a function of pair identity reiterates the observation that a subset of atom pairs contributes disproportionately to the the propagation of coupled conformational change (Figure 4 C); of the top ten types of atomic interactions, nine are interactions between two backbone atoms (Figure 4 D). This is not a consequence of covalent linkage between atoms, as such pairs were not included in this analysis.

We then turned our attention to the overall pattern of coupling between residues in the protein. The MI for every pair of residues across all models in the super-ensemble was calculated, yielding a square mutual information matrix, \mathbf{I} , for each super-ensemble (Figure 5). These matrices share a number of features. First, the signal is dominated by the diagonal, which essentially reports on the entropy characteristic of a given residue in the protein. Off-diagonal elements typically have values around 10-fold lower in signal than diagonal elements and report on the degree of coupling between residues, with substantial coupling evident between many pairs of both contacting and non-contacting residues. Furthermore, minimum values of the MI matrices approach zero for some pairs of residues, which suggests that superposition has largely eliminated trivial signal associated with rigid body motions between models in the super-ensemble.

Importantly, MI matrices calculated from different super-ensembles are remarkably similar (Figure 6), with the mean pairwise correlation coefficient between elements equal to 0.93 ± 0.06 . This result strongly suggests that the coupling pattern observed is robust over independent datasets and ensemble refinement parameter choice.

2.2 Collective fluctuations can be extracted from the PDZ3-CRIPT coupling data

Encouraged by the consistency of the results, we proceeded to investigate patterns of coupling between residues in the core PSD-95 PDZ3 domain (residues 308–390) and the four most C-terminal residues of the CRIPT peptide (residues 6–9). As we are not interested in the strong but trivial correlations arising from interactions in primary structure, we rescaled the diagonal and near-diagonals of the average MI matrix over super-ensembles $\bar{\mathbf{I}}$ to the off-diagonal element average, yielding a new matrix, \mathbf{J} (Figure 7). In order to assess the number of significant modes in the data, we performed eigendecomposition on \mathbf{J} and examined the eigenvalue spectrum relative

to a random matrix model, which suggests that two collective features dominate (Figure 8 B). This represents a conservative lower bound on the estimation of the dimensionality of the data. Reconstruction of \mathbf{J} with the top two eigenvectors and eigenvalues from the decomposition yields a pattern which visibly captures the global pattern present in the raw matrix (Figure 8 C).

To assess the contributions of the different amino acids in the protein to the collective features of \mathbf{J} , we performed symmetric non-negative matrix factorization (SNMF)^{37,38} of rank two. SNMF is an ideal approach for decomposing pairwise mutual information matrices, as per-residue contributions to the factors used to reproduce the data are constrained to be positive (in contrast to the elements of eigenvectors). The resulting two factors clearly reproduce the data in a manner similar to the top modes from the eigendecomposition, but they can be mapped unambiguously to the structure of PSD-95 PDZ3-CRIPT in order to gain an intuitive sense of the contributions of different residues to collective features of the data (Figure 9). Furthermore, decomposition with increasing rank results in what appears to be a physically meaningful hierarchical breakdown of the collective features (Figure 10).

Strikingly, the contributions of the remaining (“bottom”) modes of \mathbf{J} also contain substantial information about the coupling of amino acids in the protein. Reconstruction of the data with the bottom eigenmodes yields an MI matrix (Figure 8 E) which is visibly similar to the contact probability matrix averaged over super-ensembles (Figure 8 F). This suggests that trivial local correlations which do not contribute dramatically to the global coupling pattern are described by the weakest modes of the data, an argument which is supported statistically as well (Figure 11 B). This result not only supports our analytical approach as a whole—features which must be present due to the physics of proteins do indeed emerge—but also validates the conceptual idea that amino acids in the protein can be engaged in both local and more distributed networks of motion.

2.3 Collective fluctuations and simple models for protein physics

Given the observation of collective fluctuations arising over super-ensembles for PSD-95 PDZ3-CRIPT, we wondered whether they might be predicted by a simple, first-principles-based approaches to modeling protein dynamics. As such, we calculated two types of elastic network models from the core atoms of PSD-95 PDZ3 domain together with the C-terminal region of the CRIPT ligand. The first model, a Gaussian-network model (GNM), idealizes the protein as an elastic network in which atoms undergo Gaussian distributed fluctuations about their mean positions^{24,39,40}. The GNM describes atoms and their interactions throughout the protein as a system of nodes and springs, where interaction potentials between atoms within some distance of one another are energetically defined as linear springs. Atoms thus undergo coupled motion in accordance with a simple harmonic potential where the only free parameters are the interaction cutoff in Ångstroms, r_c , and a force constant, γ , which effectively scales the energetics of the interactions relative to $1 kT$ and is typically set to unity. The second model, the anharmonic network model (ANM), simply generalizes this representation such that fluctuations can be anisotropic in nature, yielding both a fluctuation magnitude and fluctuation vector for each atom²⁵.

Both the GNM and ANM directly yield predictions for a correlation coefficient matrix, \mathbf{R}_{NM} , that describes the correlated fluctuations of atoms in PSD-95 PDZ3-CRIPT, and an equivalent correlation coefficient matrix, \mathbf{R}_{data} , can be calculated directly from the positions of each atom throughout the corresponding super-ensemble (Figure 12 A–C). While some patterns within the matrices appear similar, \mathbf{R}_{data} appears to describe a much different pattern of correlation than either network model. The GNM clearly captures primarily local information that is dominated by atomic contacts, as \mathbf{R}_{GNM} largely lacks the bands of correlation that span \mathbf{R}_{data} . The pattern of correlation present in \mathbf{R}_{ANM} also appears to favor contacts, but the pattern of coupling is much more pervasive

than that of \mathbf{R}_{GNM} . Importantly, neither appears to appropriately capture both the dimensionality and the distributed nature of correlations in \mathbf{R}_{data} , suggesting that, in general, neither adequately describes the fluctuations observed over the super-ensemble.

That being said, the principle of superposition allows a more nuanced approach to the information content of these matrices. Indeed, a powerful feature of our analysis and of the network models described here is that the correlation matrix is the linear sum of many contributions, so we can statistically isolate and compare individual observed or predicted modes corresponding to collective fluctuations. To perform this comparison, we performed eigendecomposition on each of the correlation coefficient matrices and analyzed the top ten eigenvectors with the largest corresponding eigenvalues. We then calculated the pairwise correlation coefficients between each of the eigenvectors of \mathbf{R}_{data} with the equivalent eigenvectors of either \mathbf{R}_{GNM} or \mathbf{R}_{ANM} . While the overall correlation between eigenvectors is low, several modes showed higher degrees of correlation than average.

In the case of the GNM, the magnitudes of the second eigenvector were found to be quite correlated to those from second mode calculated from the data ($r = -0.71$; Figure 12 D). Reconstruction of the mode with data (Figure 12 E) or with the model (Figure 12 F) yields actual and predicted contributions of the atoms to this mode. Remarkably, the reconstructions correspond reasonably well for this mode. While certainly not perfect—the model seems to frequently overestimate the coupling between different regions of the protein—this observation does suggest PSD-95 PDZ3-CRIP1 undergoes at least some isotropic fluctuations which are coupled throughout the molecule, with interactions between the α_1 helix and the beginning of the β_2 strand dominating and influencing motions in other regions of the protein. For the ANM, the results are less dramatic but interesting nonetheless. The magnitudes of the first eigenvector correlated

reasonably well with the equivalent eigenvector from the data ($r = -0.59$; Figure S15, G). Reconstruction of atomic correlations with the first eigenmodes of the data (Figure S15, G) or the model (Figure 12 I) yields disparate but related patterns. As in the case of correlation with the GNM, the relationship between these matrices suggests that the model typically overestimates the degree of coupling between regions of the protein, but generally produces correlations of the appropriate sign with proper boundaries. Together, these results suggest that the fluctuations observed over the PSD-95 PDZ3-CRIPT super-ensemble are considerably more complex than expected from theoretical considerations. While certainly not surprising, this lack of correspondence may provide clues for future development of more accurate elastic network models.

2.4 Averaging of mutual information across homologous PDZ domains reveals patterns of conserved collective fluctuations

We have shown that correlated motions can be reliably inferred from high-resolution crystallographic data. While we have taken care to eliminate factors such as data incompleteness, non-isomorphism, and radiation damage as causes of correlated motion, motions may still arise for a number of less interesting reasons. First, protein motions may be constrained or distorted by lattice contacts. Second, in the absence of corroborating information, there is no inherent reason to believe that a collective feature is relevant for protein function, and, even in the limit where an observed motion is biologically relevant, the relationship between it and the general function of the protein family may not be clear.

In light of these concerns, how can these collective features be validated and related to the broader features of the PDZ domain family? As the principle of conservation provides an essential filter for assessing the relevance of features observed in biological systems, we sought to identify

collective features which appear consistently across different PDZ domains. This approach is critical both technically and conceptually. First, it provides a consistency check for the approach; some patterns of dynamics are likely to be shared between domains simply by virtue of conservation of overall fold⁴¹. Second, it represents a novel opportunity to establish the relevant dynamical features of PDZ domains in general—to allow us to separate the idiosyncratic dynamical features of any given domain from the conserved features which must be related to the general function of the PDZ domain family.

To identify correlated motions of basic importance to PDZ domain function, we repeated our analysis for a selection of PDZ domains. Specifically, we solved room-temperature crystal structures of ten different PDZ domain complexes (including PSD-95 PDZ3 CRIPT; Figure 13) and repeated our ensemble refinement protocol for each, yielding a diverse set of super-ensembles (Figure 14). We calculated pairwise MI matrices for each (Figure 15); qualitatively, they are quite diverse, but share the same characteristic “blockiness” observed in the case of PSD-95 PDZ3-CRIPT. To quantitatively compare the MI matrices of different homologues, we used a multiple sequence alignment (MSA) to assess simple correlation between them. Overall, the average degree of correlation between any pair of matrices is modest ($r = 0.40 \pm 0.15$), and hierarchical clustering reveals substantial structure in this pattern; two groups emerge upon clustering (Figure 16 A). Furthermore, this pattern can not be explained by simple sequence identity ($r = 0.25$; Figure 16 B).

These data allow us to separate the idiosyncratic features of individual proteins from the conserved features conserved over the PDZ family. To identify patterns of coupling which persist across PDZ domains, we used the MSA to average MI matrices across PDZ domains, yielding a consensus MI matrix, **K** (Figure 17 A). Much like the matrices from which it was calculated, **K**

shows a heterogeneous pattern of off-diagonal coupling, although it is visibly more complex and less dominated by bands of coupling. This observation is further supported upon eigendecomposition of \mathbf{K} ; analysis of the eigenvalue spectrum reveals five significant modes as assessed by comparison to a random matrix model (Figure 17 B), and reconstruction of the data with the top five eigenmodes or using SNMF largely recapitulates the pattern (Figure 17 C, D). As in the case of PSD-95 PDZ3-CRIPT, the remaining modes carry meaningful information as well. Visual comparison of the bottom reconstruction of \mathbf{K} with the average of all contact probability matrices reveals a similar pattern of off diagonal peaks (Figure 17 E, F) and quantitative comparison demonstrates a statistically significant difference between the magnitude of MI at contacting positions versus non-contacting positions (Figure 11 D).

How do the top five modes of \mathbf{K} map to the structure of a PDZ domain? To answer this, we mapped the elements of each SNMF factor to the structure of PSD-95 PDZ3-CRIPT (Figure 18; all residue indices discussed correspond to this structure). Intriguingly, each factor is dominated by a different subset of residues in the alignment which localize to a distinct region of the protein in three-dimensional space. The first factor is comprised largely of both core and surface residues comprising the two β -sheets of the PDZ domain in the region at which they converge, near the termini of the domain. The second factor is dominated by a set of residues adjacent to the first; coupling spans the specificity determining region of the ligand binding cleft, from the α_2 helix to the second and third β -strands. Importantly, one of the residues contributing the most to this factor is residue 372, which forms a key interaction with peptide ligand and has been shown to play an essential role in mediating the class specificity of PDZ domains (see Chapter 2). Residues in the β_1 - β_2 loop—the so-called carboxylate binding loop—are at the core of the third factor, and coupling spreads to nearby secondary structure elements, including the α_1 helix and the α_2 - β_6 loop.

This factor is notable in that motion of the carboxylate binding loop is often directly coupled with binding state in X-ray crystal structures, where the loop is frequently packed more tightly against the domain upon ligand binding. The fourth factor is primarily composed of the β_1 - β_4 - β_6 sheet, with correlation emanating radially about position 387. The fifth and final factor is perhaps the most centrally located, and is notable as the positions most enriched in MI contribute directly to interactions with much of the C-terminal residue of the ligand. The pattern of coupling extends from this site, and includes elements of the very end of the carboxylate binding loop, the α_1 helix, the β_2 - β_3 sheet, and the beginning of β_4 .

2.5 Conclusions and future directions

Here, we have established a framework with which to extract correlated motions from crystallographic data. Our approach reveals the conformational fluctuations of a series of homologous PDZ domains, and explores the degree to which such fluctuations are conserved. Generally, the patterns of pairwise coupling between residues suggests that a limited number of collective fluctuations are present in any given super-ensemble. In the case of PSD-95 PDZ3-CRIPT, mapping this pattern to the structure reveals two features which dominate either half of the β -sandwich. The first mode is dominated by residues with the strongest coupling, and expands outwards from the C-terminus of the ligand to several distant sites—up to the α_1 helix and down along the surface where the two halves of the sandwich converge near the surface that contacts the α_3 helix. Previous work investigating the correlation between backbone chemical shift upon peptide binding in PSD-95 PDZ3 with mutagenesis suggests that dynamics of many of these sites may be important as well^{34,42}. NMR studies in other domains undergoing changes in chemical shift upon ligand binding also highlight these regions of the domain to varying degrees, including PTP-1E PDZ^{31,43}, AF-6 PDZ⁴⁴, Tiam-1⁴⁵, and PAR6 PDZ^{33,46}.

Upon averaging over the coupling patterns of all PDZ domains in our set, more collective features emerge. Clearly, some aspects of fluctuations in PSD-95 PDZ3 are preserved—the first mode of the consensus matrix links the ligand binding site to the termini of the domain, and the fifth connects it to the α_1 helix—but the relative contributions of the residues involved are clearly modulated. New features come into focus as well. The second mode is dominated by residues which play important roles in establishing the specificity of the domain—both directly at the 372 position, which directly interacts with the -2 position of the bound ligand²⁸, and indirectly by positions 336 and 375, the identities of which can dramatically alter ligand binding cleft geometry⁴⁷. The third mode appears to be associated with fluctuation of the carboxylate binding loop and surrounding elements—a fluctuation observed in several cases to correlate with ligand binding, including PSD-95 PDZ3^{30,42}. Finally, the fourth mode appears similar to the second in the case of PSD-95 PDZ3, albeit more focused.

Together, these results suggest that our method meets the three criteria we initially posed. Multiple ensemble refinements over independent datasets collected for the same protein yield consistent patterns of correlation, and these patterns can be statistically decomposed to yield collective fluctuations in regions previously noted to undergo functionally relevant motion. Finally, and most importantly, we demonstrate that averaging patterns of coupling calculated from independent domains yields a consensus picture of inter-residue conformational coupling—a picture of the conserved dynamics of the PDZ family. The pattern is sparse, with peaks on the order of the individual matrices and a background which is quantitatively similar to the consensus contact graph. Together, this supports the validity of our method as an approach to infer functionally important motions from crystallographic data.

Several practical avenues of research remain. First, while diffraction data were collected at room temperature for the purposes of this study, preliminary results suggest that cryogenic data may in fact be perfectly sufficient for the application of our method. Furthermore, questions regarding the resolution range over which the method can be applied are now of interest; while we focused on only the most well-diffracting crystals for the present study, most structural biologists aren't so lucky. Finally, while we show that conformational space is sampled sufficiently by the ensemble refinement algorithm, we note that this is effectively an internal test which assumes that our starting model generation procedure and the ensemble refinement protocol are fully sampling conformational space accessible in reality. Further analysis of the role of input model heterogeneity in influencing the ensemble refinement trajectories as well as the implementation of features designed to give users greater control over ensemble refinement simulation behavior will be required for more general application.

From a conceptual perspective, this approach lays the foundations for understanding the types of motions that proteins exhibit as a function of thermal agitation. Just as averaging correlations over members of the PDZ family reveals the conserved dynamical features that they share, it will be interesting to extend this approach to the analysis of multiple protein families in order to understand the dynamical properties of proteins as general materials. Such studies may reveal information that is needed to understand the role of conformational exploration and collective dynamics in essential processes, from protein folding to catalysis.

3 Materials and methods

3.1 Construct acquisition

15 members of the PDZ domain family were selected for study based on the presence of a crystal structure in the PDB with resolution of at least 1.5 Å; of these, data from 9 are presented. Constructs were acquired from a variety of sources. Erbin was a gift from Megan McLaughlin, Sachdev Sidhu lab, University of Toronto. PDLIM7 PDZ (SGC1), PICK1 PDZ1 (SGC3), LNX2 PDZ2 (SGC5), MPDZ PDZ3 (SGC7), and SJ-BP PDZ (SGC8) were acquired from the Structural Genomics Consortium. Syntenin-1 PDZ2 was a gift from Natalya Olekhnovich, Zygmunt Derewneda lab, University of Virginia. Tiam-1 was a gift from Xu Liu, Ernesto Fuentes lab, the University of Iowa Carver College of Medicine.

3.2 Protein purification and crystal growth

Syntenin-1 with and without C-terminal ligand and Erbin were subcloned into pNIC28 for expression. The final 11 amino acids corresponding to a linker and putative ligand were removed from the Erbin construct. Expression, purification, and crystallization protocols were adapted from those described elsewhere⁴⁸. Unless otherwise noted, proteins were expressed recombinantly by growing *Escherichia coli* strain BL21(DE3) harboring the expression plasmid in Terrific Broth (TB) with selection antibiotic at 37°C with shaking until an OD_{600} of 0.8–1. Culture temperature was then reduced to 18°C and expression was induced by adding isopropyl β -D-1-thiogalactopyranoside (IPTG) to a final concentration of 1.0 mM. After 18 hr, cells were harvested by centrifugation, flash frozen, and stored at -20°C until purification. For 6×His-tagged proteins (Erbin, PDLIM7 PDZ, PICK1 PDZ1, LNX2 PDZ2, MPDZ PDZ3, SJ-BP PDZ), cells were resuspended in binding buffer (25 mM Tris pH 8, 500 mM NaCl, and 10 mM imidazole) in the

presence of protease inhibitors (either 1 mM PMSF, 2 $\mu\text{g}/\text{mL}$ pepstatin, 10 $\mu\text{g}/\text{mL}$ leupeptin or Roche cOmplete protease inhibitor tablets), lysed by sonication, and centrifuged. 6 \times His-tagged proteins were then bound with Ni-NTA agarose resin (Qiagen) and eluted in elution buffer (100 mM Tris pH 8.0, 1 M sodium chloride, and 400 mM imidazole). For GST-tagged proteins (Syntenin-1 PDZ2 with and without attached C-terminal ligand, Tiam-1), cells were resuspended in phosphate-buffered saline (PBS; 137 mM sodium chloride, 2.7 mM potassium chloride, 10 mM disodium hydrogen phosphate, 1.8 mM potassium dihydrogen phosphate) in the presence of protease inhibitors (either 1 mM PMSF, 2 $\mu\text{g}/\text{mL}$ pepstatin, 10 $\mu\text{g}/\text{mL}$ leupeptin or Roche cOmplete protease inhibitor tablets), lysed by sonication, and centrifuged. GST-tagged proteins were then bound with glutathione sepharose 4B resin (GE Healthcare Life Sciences) and eluted in PBS with 20 mM reduced glutathione pH 8. 6 \times His and GST tags were then removed by proteolysis with ProTEV protease (Promega). Proteolysis was performed concurrently with overnight dialysis into gel filtration buffer, typically at room temperature. The contents of the gel filtration buffer differed by protein. Erbin was dialyzed into 50 mM Tris pH 8.3 and 500 mM sodium chloride. PDLIM7 PDZ was dialyzed into 50 mM HEPES pH 7.4, 500 mM sodium chloride, and 0.5 mM TCEP. PICK1 PDZ1 was dialyzed into 20 mM HEPES pH 7.4, 150 mM sodium chloride, and 0.5 mM TCEP. LNX2 PDZ2 was dialyzed into 50 mM HEPES pH 7.5, 500 mM sodium chloride, 5% glycerol, and 0.5 mM TCEP. MPDZ PDZ3 was dialyzed into 50 mM Tris pH 7.5, 500 mM sodium chloride, and 0.5 mM TCEP. SJ-BP PDZ was dialyzed into 50 mM HEPES pH 7.4, 500 mM sodium chloride, and 0.5 mM TCEP. PSD-95 PDZ3 was dialyzed into 20 mM HEPES pH 7.2 and 150 mM sodium chloride. Syntenin-1 PDZ2 proteins were dialyzed into 50 mM Tris pH 7.5 and 150 mM sodium chloride. Tiam-1 was dialyzed into 10 mM HEPES pH 8.0 and 50 mM sodium chloride. After dialysis, all proteins were further purified by gel filtration (HiLoad 26/60 Superdex

75, GE Healthcare Life Sciences) and concentrated to 1–2 mM using a 10 kDa cutoff Amicon Ultra filtration device (EMD Millipore).

3.3 Peptide synthesis

The CRIPT (Acetyl-TKNYKQTSV-COOH) and syndecan-1 (Acetyl-ENEQVSAV-COOH) peptides were synthesized using standard Fmoc chemistry (UTSW Proteomics Core Facility), HPLC purified, and lyophilized. CRIPT peptide was resuspended in water, while Syndecan-1 peptide was resuspended in 100 mM ammonium hydroxide.

3.4 Protein crystallization

All crystals were grown by the hanging drop vapor diffusion method at 20°C. Microseeding was used throughout to optimize crystal nucleation and morphology. Erbin crystals were grown under conditions similar to those reported previously⁴⁹. Protein was diluted to 8.3 mg/mL (750 µM) in PBS and then mixed 1:1 with crystallization buffer containing 100 mM Tris-HCl pH 7.5 and 29% polyethylene glycol (PEG) 4000. PDLIM7 PDZ crystals were grown as described previously^{gileadi:2007:2q3g}. 28.0 mg/mL (3.0 mM) protein was mixed 1:1 with crystallization buffer containing 100 mM HEPES pH 7.5, 20% PEG 4000, and 10% isopropanol. PICK1 PDZ1 crystals were grown as described previously⁵⁰. 3.4 mg/mL (340 µM) protein solution was mixed 1:2 with crystallization buffer containing 100 mM M Bis-Tris pH 5.5, 23% PEG 3350, and 0.2 M ammonium acetate. LNX2 PDZ2 crystals were grown as described previously^{roos:2008:2vwr}. 8.7 mg/mL (840 µM) protein solution was mixed 1:1 with crystallization buffer containing 43 mM disodium hydrogen phosphate and 57 mM citric acid pH 4.2 and 27% PEG 300. MPDZ PDZ3 crystals were grown as described previously⁵⁰. 10.2 mg/mL (980 µM) protein was mixed 1:1 with crystallization buffer containing 100 mM sodium acetate pH 4.5, 27% PEG 10000, and 200 mM

lithium sulfate. SJ-BP PDZ crystals were grown as described previously^{tickle:2007:2jin}. 13.2 mg/mL (1.2 mM) protein was mixed 1:1 with crystallization buffer containing 100 mM sodium citrate pH 4.0, 1 M lithium nitrate, and 22% PEG 6000. Crystals of PSD-95 PDZ3 bound to CRIPT ligand were grown under conditions similar to those reported previously³⁰. CRIPT ligand was diluted to 10 mM in protein buffer. Protein was mixed with CRIPT ligand 1:2 for a final protein and peptide concentration of 9.0 mg/mL (700 μ M) and 1.6 mg/mL (1.4 mM), respectively; the mixture was then incubated on ice for 1 hr. The protein-ligand solution was then mixed 1:1 with crystallization buffer containing 1.0 M sodium citrate pH 7.0. Crystals of Syntenin-1 PDZ2 were grown under conditions similar to those reported previously^{51,52}. Pseudo-liganded crystals were grown by mixing 9.7 mg/mL (1.1 mM) of Syntenin-1 PDZ 1:1 with crystallization buffer containing 100 mM HEPES pH 7.0 and 34% PEG 4000. To grow crystals of Syntenin-1 PDZ2 bound to Syndecan-1 peptide, protein and ligand were first mixed to around 1:3 for a final protein concentration of 7.9 mg/mL (940 μ M) and ligand concentration of 2.3 mg/mL (2.5 mM) and then incubated on ice for 1 hr. The protein ligand solution was then mixed 0.65:1 with crystallization buffer containing 100 mM HEPES pH 7.3, 1.8 M ammonium sulfate, and 200 mM magnesium sulfate. Crystals of Tiam-1 were grown as described previously⁴⁵. 20.6 mg/mL (2.0 mM) protein was mixed 1:1 with crystallization buffer containing 100 mM Tris-HCl pH 7.2, 200 mM sodium thiocyanate, and 22% PEG 3350.

3.5 X-ray diffraction data collection

Data were collected at Stanford Synchrotron Radiation Lightsource (SSRL) Beamline 11-1. All data described here were collected at 277 K, and crystals were protected from humidity fluctuations with either MicroRT X-ray capillaries (MiTeGen, Ithaca, NY)⁵³ or, in the case of

PDLIM7 PDZ and SJ2BP PDZ, a viscous oil, Paratone-N (Parabar 10312). Data were collected using a PILATUS 6M PAD operating in continuous data collection mode at 12.5 Hz.

3.6 Data reduction

X-ray data were processed using HKL2000⁵⁴ with autocorrections enabled in the final scaling steps. Dataset quality was primarily assessed by four metrics, that is, (1) maximum resolution less than 1.5 Å, (2) precision in merging R-factor (R_{pim}) less than 5%, (3) an radiation damage coefficient less than 0.2, and (4) mosaicity less than 2°. Resolution was chosen based on merging statistics and $CC_{1/2} = 0.5$.

An important consideration with respect to room temperature data collection is the accrual of specific radiation damage, as chemical changes to crystalline proteins take place at higher rates and can introduce undesirable artifacts⁵⁵⁻⁵⁷. For each dataset, data reduction statistics indicate limited radiation damage as assessed by changes in the scaling B -factor and mosaicity over data collection as well as the decay R factor, R_d ⁵⁸.

3.7 Model building

Model building was performed using various programs of the PHENIX 1.9 software suite⁵⁹. Friedel pairs were merged, test sets were assigned by setting aside 8% of total reflections (with a cap of 1500 reflections), and French-Wilson corrections were applied. For each dataset, initial phases were calculated by molecular replacement using with the best available structure in the PDB. In some cases, where possible, the search model was improved with *PDB_REDO*⁶⁰. Next, coordinates of the initial model were randomized by 0.5 Å, subjected to a round of Cartesian simulated annealing, and refined with riding hydrogens using *phenix.refine* with an isotropic atomic displacement parameter (ADP) model and coordinate and ADP weight optimization

enabled. The model was then manually improved with *Coot*⁶¹ and subjected to further rounds of automated refinement. Throughout this process, waters and alternative conformations were added conservatively. Once a local minimum in R_{free} had been reached, the data were further refined against an anisotropic ADP model. Composite omit maps used to guide placement of alternative conformations. Model building was completed for 16 datasets over 10 proteins. Refinement statistics were typically well below average for structures of equivalent resolution and are summarized in Table 1.

3.8 Ensemble refinement and construction of super-ensembles

Super-ensembles were constructed from final models using a series of Python scripts and *phenix.ensemble_refinement*²¹. First, two ensemble refinement grid searches were performed on a maximum occupancy single conformer model in order to determine optimal values of the ensemble refinement parameters p_{TLS} , T_{offset} , and τ_x as determined by crystallographic R -factors.

Next, 124 single-conformer models were generated from the multi-conformer starting model. These models differ at positions with multiple conformations; this approach improves conformational sampling, as does not always run long enough to fully equilibrate. Two subsets comprise the full set. The first subset of models was created by choosing a conformation at each position of the model with no regard for the conformations of adjacent residues, while the second subset of models was created by choosing a conformation for a series of contiguous residues with alternative conformations. In both modes, the probability of choosing a conformation was equal to its occupancy.

Ensemble refinement was then performed on each model with determined parameter values, yielding a set of ensembles (i.e., a "super-ensemble"). As many as 20% of super-ensembles for

each dataset were discarded based on deviation of R_{work} distribution from normality as determined by the Shapiro-Wilk test⁶² as implemented in `scipy.stats.shapiro`⁶³.

3.9 Calculation of correlation coefficient matrices and PCA

The calculation of all-atom covariance over atomic coordinates in a given super-ensemble was calculated as implemented in ProDy, which provides a unified API for performing dynamics analysis⁶⁴. First, all models in the super-ensemble were iteratively superposed using the *interpose* method with all non-hydrogen atoms as input. A three-dimensional coordinates matrix \mathbf{C} with $n_{\text{models}} \times 3 \times n_{\text{atoms}}$ elements was then derived from the super-ensemble, where n_{models} is the total number of models in the super-ensemble, n_{atoms} is the number of non-hydrogen atoms in the protein, and 3 denotes Cartesian space. A two-dimensional atomic covariance matrix \mathbf{Q} with $n_{\text{atoms}} \times 3$ elements was then calculated from this matrix using the ProDy *getCovariance* method. Each element of \mathbf{Q} is then calculated over observations of two atoms in a given dimension. For a pair of atoms $i, j \in \{1, 2, 3, \dots, n_{\text{atoms}}\}$, vectors \mathbf{i}^a and \mathbf{j}^b having n_{models} elements corresponding to all observations in dimensions $a, b \in \{x, y, z\}$ are constructed from \mathbf{C} . For a pair of atoms in two matched dimensions, a corresponding element of \mathbf{Q} is given by

$$\mathbf{Q}_{ij}^{ab} = \frac{1}{n_{\text{models}}^2} \sum_{p=1}^{n_{\text{models}}} \sum_{q=1}^{n_{\text{models}}} (\mathbf{i}_p^a - \mathbf{i}_q^a) \cdot (\mathbf{j}_p^b - \mathbf{j}_q^b)^{\text{T}}.$$

For comparing linear correlations to MI, we calculated the correlation coefficient matrix \mathbf{R} . As correlation coefficients can be considered as a form of covariance normalized to the variance, we then calculated the atomic correlation coefficient matrix \mathbf{R} , where

$$\mathbf{R}_{ij}^{ab} = \frac{\mathbf{Q}_{ij}^{ab}}{\sqrt{\mathbf{Q}_{ii}^{ab}} \sqrt{\mathbf{Q}_{jj}^{ab}}}.$$

To assess covariance or correlation at the level of residues without regard for Cartesian dimension, we operated on blocks of \mathbf{Q} or \mathbf{R} . For example, to calculate the residue-level correlation coefficient matrix \mathbf{S} from \mathbf{R} , submatrices of correlation coefficients \mathbf{R}_{uv} between all atoms in a pair of residues $u, v \in \{1, 2, 3, \dots, n_{\text{residues}}\}$ in all dimensions d were reduced. Explicitly,

$$\mathbf{S}_{uv} = \text{mean} \left(\max_d (\text{abs}(\mathbf{R}_{uv})) \right)$$

The matrix \mathbf{S} thus has dimensions $n_{\text{residues}} \times n_{\text{residues}}$ and is directly comparable to pairwise residue MI.

3.10 Calculation of elastic network models

Elastic network models were calculated from single crystal structures using the ProDy package⁶⁴. Calculation of Gaussian network models was performed using the GNM class and related methods, while anharmonic network model calculations were performed using the ANM class and related methods. The theory underlying these calculations is thoroughly presented elsewhere^{25,39,40}. In both cases, ProDy yields a covariance matrix \mathbf{Q} ; in the case of the GNM, the dimensions are $n_{\text{atoms}} \times n_{\text{atoms}}$, while the dimensions in the case of the ANM are $n_{\text{atoms}} \times n_{\text{atoms}} \times 3$. Correlation coefficient matrices \mathbf{R} were calculated from \mathbf{Q} as described in §3.9. These matrices are directly comparable to correlation coefficient matrices calculated from super-ensembles.

Several considerations related to the calculation of either elastic network model are relevant. First, atom selections were chosen to minimize so-called “tip-effects”, in which regions of the protein extending away from the body of the protein undergo abnormally large displacements due to a lack of steric constraint in the asymmetric unit⁶⁵. Although Mingyang et al. have proposed a systematic approach for automatically dealing with these phenomena, we simply trimmed either

terminus of a given protein chain back to the core domain until correlations appeared balanced throughout the molecule. Second, we chose to build all-atom elastic network models, as opposed to models describing only the motion of C_α atoms. This decision was initially an arbitrary one, but subsequent comparison of all-atom models with C_α -only models showed limited differences in the overall pattern of coupling or correlation. Furthermore, super-ensembles provide an all-atom description of the correlation structure, so an adequate model should reproduce all of the data anyways. Third, with respect to parameter choice for either network model, we chose to keep the force constant γ fixed at 1.0. We then chose the distance cutoff r_c by calculating correlation matrices for r_c from 4.5–15 Å in 0.5 Å steps, and picked the value that for which eigenvectors were most correlated with those from the correlation coefficient matrix derived from the super-ensemble. In all cases, the best value for r_c was substantially lower than the default of 10 Å, typically ~ 5 Å.

3.11 Calculation of residue-by-residue mutual information from super-ensembles

To characterize correlation between residue motion for each protein, residue-by-residue mutual information was computed between either distributions of dihedral (torsion) angles (ϕ , ψ , ω , χ_1 , ...) or atomic coordinates (x , y , z) in Cartesian space over each super-ensemble.

First, all models in a given super-ensemble were aligned to single reference structure (C_α atoms only) using the *Bio.PDB.Superimposer* method from the Biopython package⁶⁶ or the ProDy *iterpose* method (with all non-hydrogen protein atoms)⁶⁴. Then, for a given model $m \in \{1, 2, 3, \dots, M\}$ and amino acid residue $r \in \{1, 2, 3, \dots, R\}$, where M is the total number of models in the super-ensemble and R is the total number of residues, each unique dihedral class h with data in units of degrees or atom class i with coordinates j in units of Ångstrom were extracted

and stored in \mathbf{D}_h^{mr} or \mathbf{C}_{ij}^{mr} , respectively. These matrices were then used to calculate the pairwise mutual information in bits between all residues in the protein across the super-ensemble using a k -nearest-neighbor estimator as implemented by Stögbauer et al. as *MIxynyn*⁶⁷. Explicitly, the mutual information in dihedrals between two residues x and y is then

$$\mathbf{I}_{xy} = I(\mathbf{D}_h^{mx}, \mathbf{D}_h^{my}) = \sum_{x \in \mathbf{D}_h^{mx}} \sum_{y \in \mathbf{D}_h^{my}} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right)$$

This calculation was repeated for all dihedrals and coordinates for residues in range R , yielding \mathbf{I} , a symmetric, non-negative matrix with residue-by-residue mutual information in bits. For coordinates, a pairwise atomic mutual information matrix was also calculated in a similar manner.

To compare \mathbf{I} between S datasets for the same protein, the mean mutual information, $\bar{\mathbf{I}}$, was calculated:

$$\bar{\mathbf{I}} = \frac{\sum_{s \in S} \mathbf{I}_s}{S}$$

Typically, the magnitude of the first few diagonals of $\bar{\mathbf{I}}$ are around an order of magnitude greater than most of the remaining matrix. This signal arises from local interactions that occur between amino acids adjacent in primary sequence space. As the focus of this work is on uncovering collective features which span the protein in ways which do not necessarily arise directly from such interactions, the diagonal of $\bar{\mathbf{I}}$ is scaled such that the mean of each diagonal approaches the average value of off-diagonal elements. Operationally, this is accomplished in several steps. First, for an $n \times n$ MI matrix, a vector $\vec{\mu}$ is calculated, where

$$\vec{\mu} = \frac{\sum \text{diag}(\bar{\mathbf{I}}, i)}{n - i} \quad \text{with } \{i \mid i \in \mathbb{Z}, 0 \leq i < n\}$$

Values of $\vec{\mu}$ will thus be maximal at zero while falling off exponentially as i approaches n .

Next, a simple exponential decay is fit to $\vec{\mu}$ by linear regression as implemented in *scipy.optimize.curvefit*⁶³:

$$\vec{\mu} : \mapsto ae^{bi} + c$$

The parameter c thus represents the “background MI” for a given matrix. This parameter is used to rescale $\bar{\mathbf{I}}$:

$$\mathbf{J} = \begin{cases} \bar{\mathbf{I}}_i \times \frac{c}{\sum \bar{\mathbf{I}}_i / (n - i)}, & \text{if } \frac{c}{\sum \bar{\mathbf{I}}_i / (n - i)} \leq 0.95 \\ \bar{\mathbf{I}}_i, & \text{otherwise} \end{cases} \quad \text{with } \{i \mid i \in \mathbb{Z}, -n < i < n\}$$

This procedure is also presented graphically (Figure S2).

To compare \mathbf{J} across H homologues, the average, diagonal suppressed MI matrices were expanded to an indexing scheme consistent with the multiple sequence alignment. Gapped positions were incorporated with non-numerical values. Matrices were then averaged over homologues to yield “consensus” matrices for dihedral and coordinate MI:

$$\mathbf{K} = \frac{\sum_{h \in H} \mathbf{J}_h}{H}$$

For averaging, any position with one or more gaps over the space of homologues was omitted from \mathbf{K} . As such, \mathbf{K} is has dimensions less than or equal to the shortest sequence in the alignment.

A variation on this averaging procedure was performed in the case of coordinate data, as different homologues showed different degrees of background MI. Naïvely averaging such

matrices can in principle over- or underweight features, so a conservative strategy was employed in order to transform and scale the data in some matrix \mathbf{J}_h relative to some reference \mathbf{J}_{ref} .

Explicitly, this is performed by solving for parameters m_h and b_h given the 25th and 75th percentiles of coordinate MI matrices \mathbf{J}_h and \mathbf{J}_{ref} :

$$\begin{aligned} m_h p_h^{25} + b_h &= p_{\text{ref}}^{25} \\ m_h p_h^{75} + b_h &= p_{\text{ref}}^{75} \end{aligned}$$

Then, transform \mathbf{J}_h using the parameters from the previous step:

$$\mathbf{J}_h^{\text{trans}} = m_h \mathbf{I}_h + b_h$$

A consensus matrix for coordinate MI can then be calculated by averaging as described previously.

3.12 Identification of collective modes from individual super-ensembles and from consensus data for the PDZ family

The MI matrices calculated have two key properties which inform subsequent analytical strategies. First, they are symmetric; the mutual information undirected and simply reports on the degree of surprise given the distributions of observables for two amino acids. Second, they are positive-valued; the mutual information of two independent distributions approaches zero by definition. While the first property implies that a natural analytical tool for the data would be eigendecomposition to reveal independent features in the data, the second parameter constrains the approach, as negative eigenvector magnitudes are challenging to interpret. As such, decomposition is performed in two steps.

First, the eigenvalue spectrum of a given matrix is compared to the eigenvalue spectra of many randomized versions of that matrix. Given the eigendecomposition of MI matrix \mathbf{J} , where $\mathbf{J} =$

$\mathbf{V}\mathbf{A}\mathbf{V}^T$, a probability distribution can be calculated for the diagonal elements of \mathbf{A} —that is, the eigenvalues of \mathbf{J} . Similarly, a randomized version of the MI data, \mathbf{J}_{rand} , can be created as well, where the primary diagonal is maintained but off diagonal elements are permuted in such a way as to maintain the symmetry of the matrix. This procedure is expected to remove long-range coupling in the matrix and provide a lower limit on the number of significant eigenmodes comprising collective features in the data. As such, an empirical cumulative distribution function (ECDF) was calculated from the diagonal elements of 100 different \mathbf{A}_{rand} , and the rank of the collective feature, k , was taken as the number of eigenvalues greater than the 99th percentile of the ECDF.

The value of k was then used for the second part of the decomposition protocol. A matrix decomposition method, non-negative matrix factorization (NMF), was originally developed for use in the decomposition of positive, real matrices³⁷. Generally, NMF is a non-convex optimization problem which seeks to reproduce a nonnegative matrix as the product of two low-rank matrices which are not necessarily identical. Recent work by Da Kuang et al. provides an algorithm, *SymNMF*, with added constraints to the method for operating on symmetric matrices³⁸. This procedure is a natural approach for decomposing the MI matrices. Given a positive, symmetric matrix $\mathbf{J} \in \mathbb{Z}_+^{n \times n}$, *SymNMF* employs Newton's method to iteratively find the non-negative matrix $\mathbf{W} \in \mathbb{Z}_+^{k \times n}$ that minimizes the functional

$$f(\mathbf{W}) = \frac{1}{2} \|\mathbf{J} - \mathbf{W}\mathbf{W}^T\|_F^2$$

In other words, the algorithm converges when the difference in Frobenius norms between the input matrix and its reconstruction is below some threshold. Here, the algorithm operates as described by Da Kuang et al., but was implemented in Python using elements from the NumPy package⁶⁸.

In practice, the performance and results of any NMF-based decomposition depend strongly on the initialization of \mathbf{W} . While many initialization strategies exist, here \mathbf{W} is initialized such that it is equal to the positive component of its rank k singular vector decomposition (SVD); this approach was first described by Boutsidis and Gallopoulos as *NNDSVD*⁶⁹ and is implemented here as in the Scikit-learn package `sklearn.decomposition.nmf._initialize_nmf`⁷⁰. While NMF solutions arising from random initializations of \mathbf{W} are not unique, the *NNDSVD*-based approach yields unique solutions due to the properties of the SVD.

In summary, the net result of this two-part approach to the decomposition of \mathbf{J} is a $n \times k$ matrix, \mathbf{W} . Each column k of \mathbf{W} represents a part of the overall pattern of correlations throughout a protein or over the consensus, analogous to an eigenmode. A given column can thus be mapped to a representative protein structure in order to facilitate the interpretation of the mode in the context of biophysical properties such as residue proximity in Cartesian space. The relationship between factors and eigenvalues is explored in Figure 19.

3.13 Contact probability matrix calculation

As a subset of residues in a protein may be in various states of contact over a given super-ensemble, contact matrices were calculated for each model in a given super-ensemble by calculating all pairwise atomic distances between each pair of residues to yield a distance matrix \mathbf{D} with dimensions $n_{\text{atoms}} \times n_{\text{atoms}} \times n_{\text{models}}$. Distance matrices were then binarized, yielding contact matrix \mathbf{C} with the same dimensions; for any two atoms $i, j \in \{1, 2, 3, \dots, n_{\text{atoms}}\}$ in model $m \in \{1, 2, 3, \dots, n_{\text{models}}\}$ and their corresponding van der Waals radii r_i and r_j ,

$$\mathbf{C}_{ij}^m = \begin{cases} 1, & \text{if } \mathbf{D}_{ij}^m \leq r_i + r_j + 0.2 \times (r_i + r_j) \\ 0, & \text{if } \mathbf{D}_{ij}^m \geq r_i + r_j + 0.2 \times (r_i + r_j) \end{cases}$$

Van der Waals radii for each atom are taken from Tsai et al.⁷¹. The pairwise residue contact matrix $\hat{\mathbf{C}}$ was calculated in a similar fashion. For a given pair of residues $p, q \in \{1, 2, 3, \dots, n_{\text{residues}}\}$ and their respective atoms $i \in \{1, 2, 3, \dots, n_p\}$ and $j \in \{1, 2, 3, \dots, n_q\}$,

$$\hat{\mathbf{C}}_{pq}^m = \begin{cases} 1, & \text{if } \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \mathbf{C}_{ij}^m > 0 \\ 0, & \text{if } \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} \mathbf{C}_{ij}^m = 0 \end{cases}$$

A contact probability matrix, \mathbf{P} or $\hat{\mathbf{P}}$, can then be calculated from \mathbf{C} or $\hat{\mathbf{C}}$ simply by averaging over models, yielding a two dimensional matrix with $n_{\text{atoms}} \times n_{\text{atoms}}$ or $n_{\text{residues}} \times n_{\text{residues}}$ where each element reflects the probability of observing an atom-atom or residue-residue contact over the super-ensemble, respectively.

4 Figures

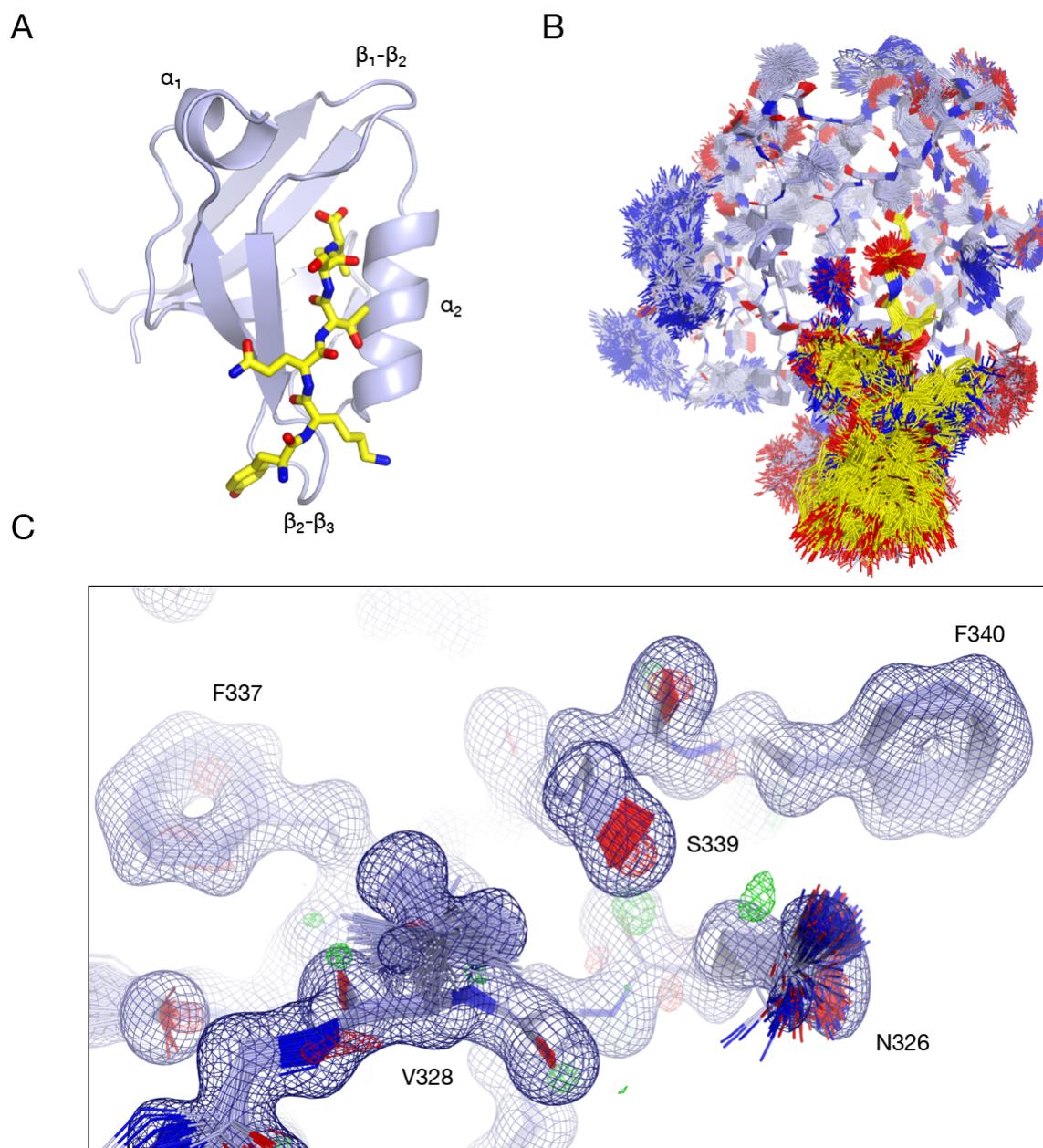


Figure 1. Ensemble refinement of crystallographic data reveals a diverse set of conformational fluctuations. **A.** Cartoon representation of the PSD-95 PDZ3 domain (blue) bound to the last nine residues of the CRIPT ligand (yellow). **B.** A single structural ensemble as refined into the X-ray

data. Nitrogen atoms are indicated in dark blue, and oxygen atoms are colored red. Conformational heterogeneity largely follows expectation, with large-amplitude displacements observed for surface-exposed side chains and loop regions and smaller distributions for atoms in the core of the protein. Some core residues do show rotamer switching as well, however. **C.** A region of density from near the start of the β_2 strand showing different types of fluctuations. $2mF_o-DF_c$ density is drawn in dark blue and contoured at 1σ ; mF_o-DF_c density is shown in green and red and contoured at 3 and -3σ , respectively. F339, and F340 all show harmonic fluctuations throughout the ensemble, while V328 and N326 show anharmonic, rotameric transitions.

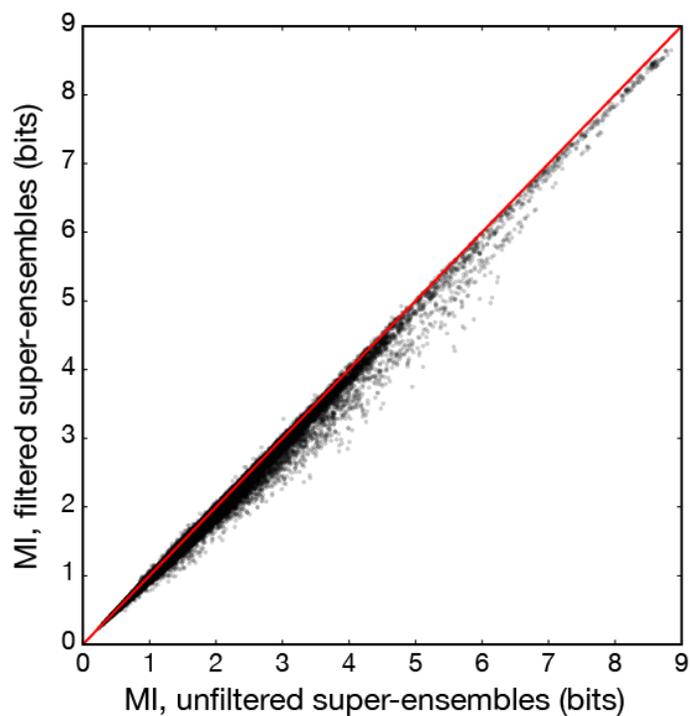


Figure 2. Filtering super-ensembles by crystallographic R -factors reduces spurious MI. Comparison of off-diagonal MI calculated over all PDZ3-CRIPT super-ensembles with and without filtering by R_{work} ; the worst $13 \pm 6\%$ of ensembles were discarded. Many points fall below the red line, indicating that MI calculated from filtered ensembles is on average lower than MI calculated from unfiltered ensembles.

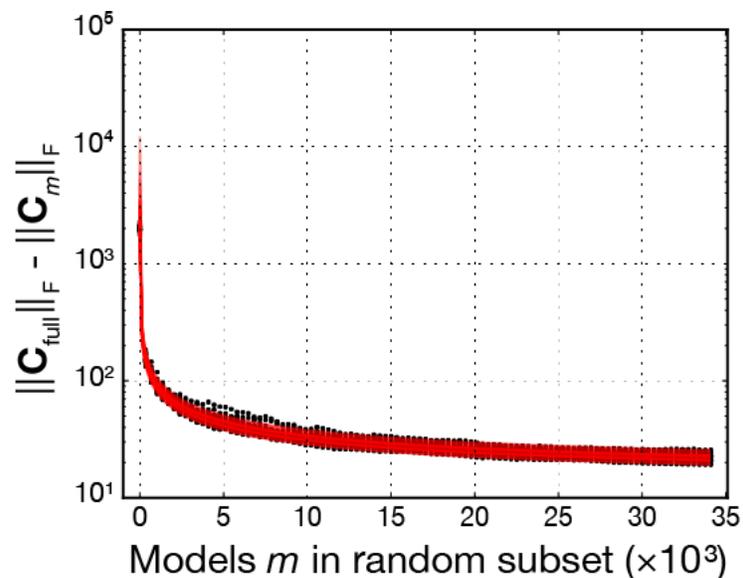


Figure 3. Conformational sampling converges over the PSD-95 PDZ3 super-ensemble. The extent to which conformational space is explored over the PSD-95 PDZ3 super-ensemble (68,200 models) for data set 1 was tested by comparing the Frobenius matrix norms of a correlation matrix, \mathbf{C}_{full} , calculated over coordinates of a randomly chosen half of the super-ensemble (34,100 models) to correlation matrices, \mathbf{C}_m , calculated over coordinates from m models, with $\{m \mid 0 < m < 34,100\}$. This process was performed 32 times, yielding an exponential decay in the difference between the correlation matrices (black). This trend was modeled (red) based on the Fisher z -transform, taking the number of observations per model (i.e., the number of atoms per model, 973 in three dimensions for a total of 2,919 observations per model) and the total number of models with a single free parameter, τ , as a measure of the statistical independence of \mathbf{C}_m relative to \mathbf{C}_{full} over the dataset. Over 32 replicates, $\langle \tau \rangle = 0.98$; this suggests that knowledge of the number of models in a super-ensemble and the number of atoms in each model is sufficient to predict the extent of conformational sampling.

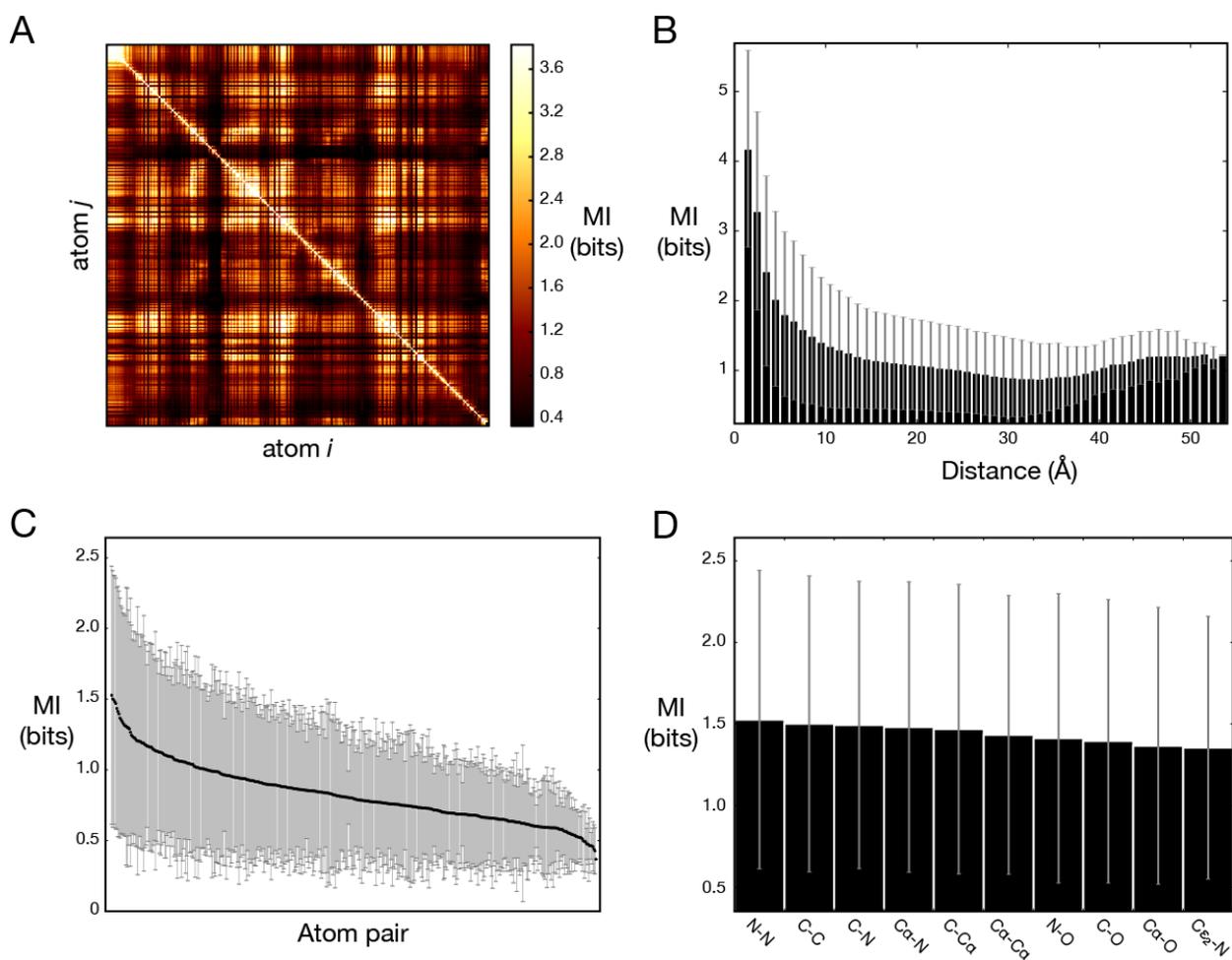


Figure 4. Pairwise atomic MI reveals key interactions for propagating coupled motion. **A.** The average pairwise atomic MI matrix calculated over all PDZ3-CRIPT datasets, with the color map scaled to the first and 99th percentiles of off-diagonal elements. Each pixel is effectively the sum of MI along each Cartesian coordinate. **B.** The distribution of MI as a function of distance between atoms shows that average MI decreases exponentially with increasing distance between pairs, but does not converge to zero. Error bars represent the standard deviation about the mean. **C.** The average MI for each observed pair of atom types in PDZ3-CRIPT with covalently linked pairs excluded. Pairs are sorted from greatest to least average MI, with error bars representing the standard deviation about the mean for all observations. **D.** The top ten pairs of atom types from C with the greatest average MI. Despite the fact that covalent interactions have been removed from the analysis, backbone-backbone interactions dominate in nine out of ten cases.

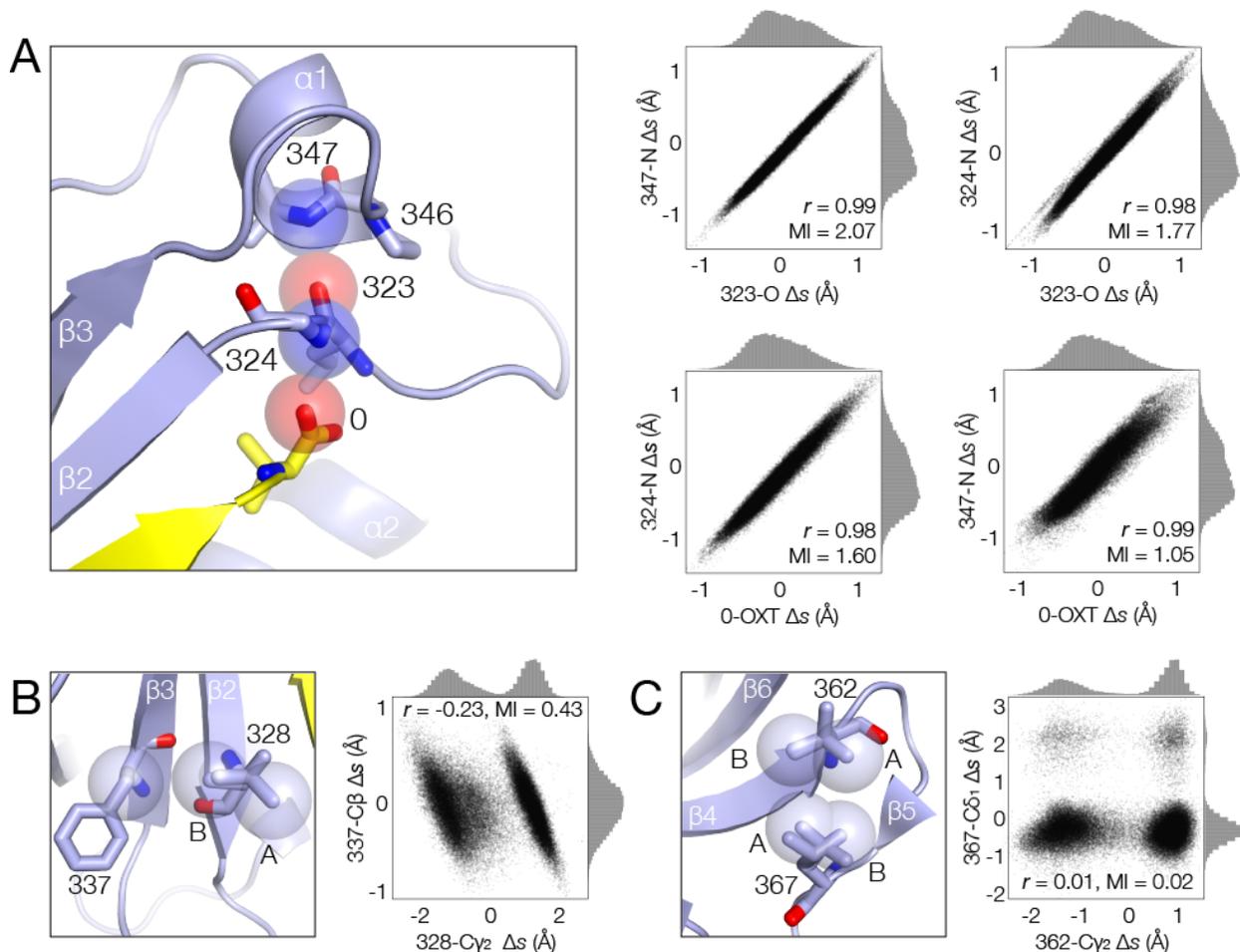


Figure 5. Correlated conformational fluctuations between atoms in the super-ensemble link sites spanning multiple elements of secondary structure. **A.** Strongly-correlated atomic fluctuations link the position of the ligand (yellow) carboxy-terminus (V0) to the α_1 helix of the PSD-95 PDZ3 domain (light blue) through a series of backbone atoms (shown as spheres; color denotes atomic identity) from residues L323, G324, and A347. The displacement about the mean of each atom over its first principal component (Δs) relative to the next is plotted on the right. The first three plots illustrate large correlation between atoms in close proximity, and the final plot (bottom right) shows relationship between either end of the chain of interactions. While distributions of Δs are non-Gaussian along these projections, the response is strikingly linear. Pearson’s correlation coefficient, r , and MI for each pair of atoms in the sequence are shown; here, the MI provides a more interpretable measure of coupling as it spans a greater range than r . **B.** A rotameric transition of the V328 side-chain does not abolish coupling with nearby F337 along the first principal component of either. The lack of relative decoupling associated with $C_{\gamma 2}$ in the conformation denoted “A” relative to that denoted “B” likely arises due to the fact that $C_{\gamma 1}$ will replace it in the interaction with C_{β} of F337. Additional coupling arises through backbone interactions between the two residues. Regarding statistics, r likely underestimates the correlation between the two atoms depicted as the distribution of Δs as $C_{\gamma 2}$ is bimodal; MI is thus more representative of the coupling

between these atoms. **C.** Despite close structural proximity, the positions of V362 C_{γ2} and L367 C_{δ2} are largely uncorrelated, regardless of rotameric state.

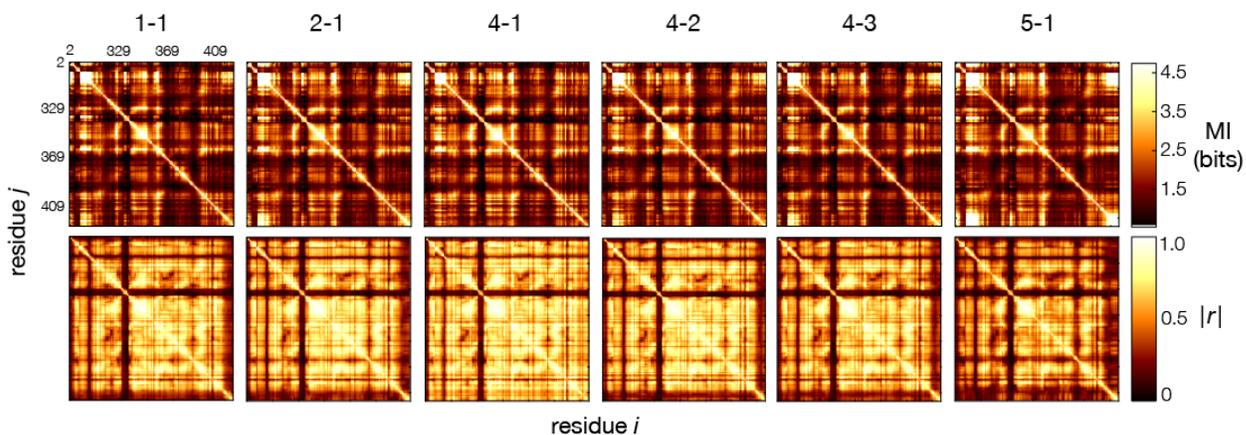


Figure 6. Coordinate mutual information matrices are highly reproducible over independent datasets and sparser than correlation coefficient matrices calculated from the same datasets. Mutual information matrices (top row) and correlation coefficient matrices (bottom row) calculated over PSD-95 PDZ3-CRIPT super-ensembles from independent datasets (1-1, 2-1, 4-1, 5-1) and from super-ensembles refined against different test sets (4-1, 4-2, 4-3). The color map for each MI matrix is scaled to the 1st and 99th percentiles of the upper triangle of the the matrix for 1-1 in order to emphasize off diagonal elements, while the color map for correlation coefficient matrices is unscaled. The average pairwise correlation coefficient between upper triangles of the MI matrices was 0.935 ± 0.060 , while the average pairwise correlation coefficient between upper triangles of the correlation coefficient matrix was 0.930 ± 0.049 .

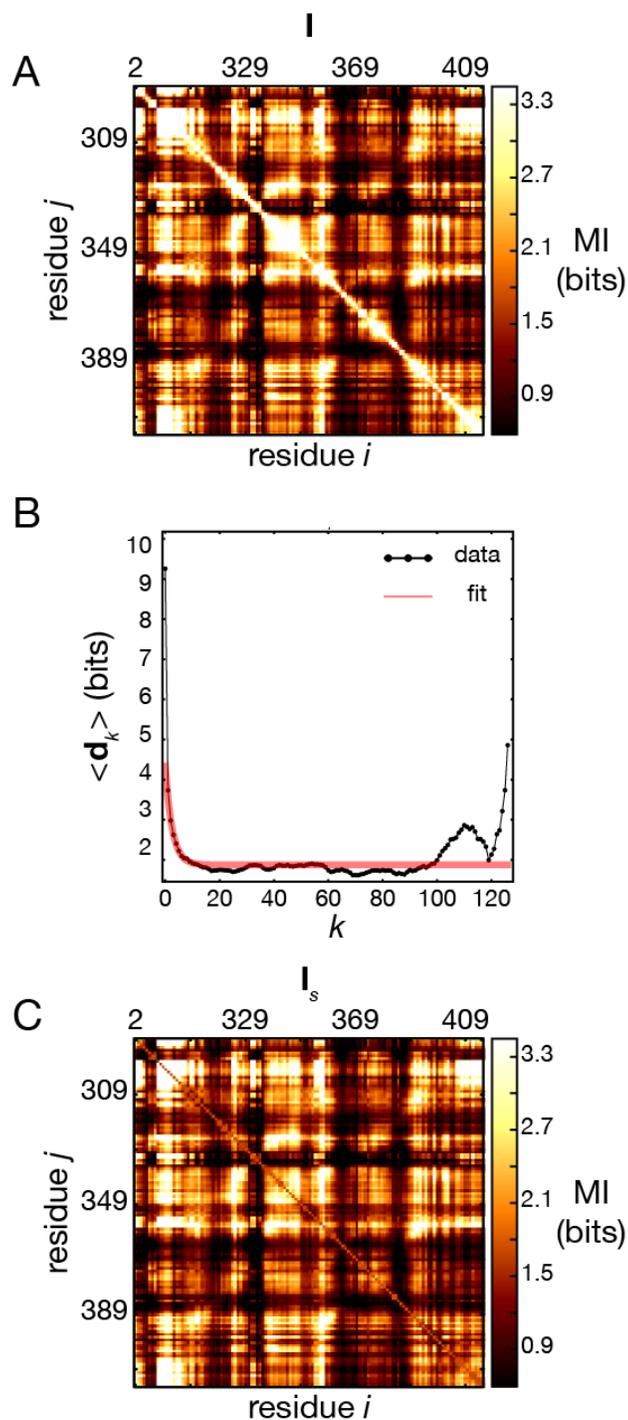


Figure 7. Diagonal scaling of MI matrices suppresses trivial signal associated with primary structure. **A.** The coordinate MI matrix, \mathbf{I} , for PSD-95 PDZ3, averaged over all super-ensembles. The mean of the diagonal of \mathbf{I} is an order of magnitude greater than the mean of the rest of the matrix. **B.** For each diagonal vector \mathbf{d} of the mean coordinate MI matrix \mathbf{I} with k for k in $\{0, 1, 2, \dots, n\}$, where n is the size of \mathbf{I} , a scaling term was applied such that $\langle \mathbf{d}_k \rangle = c$ as determined by a

simple exponential fit $y = ae^{bx} + c$, where c approximates the mean of the matrix in the absence of diagonal and near-diagonal elements. This scaling procedure is performed for increasing k until $\langle \mathbf{d}_k \rangle - c < 0.05$, typically on the order of a few diagonals. **C.** The scaled, mean coordinate MI matrix, \mathbf{I}_s , for PSD-95 PDZ3. Trivial signal near the diagonal has been suppressed, and the value of the 0th diagonal is equal to c . In both **A** and **C**, the color map is scaled to the 5th and 95th percentiles of \mathbf{I}_s .

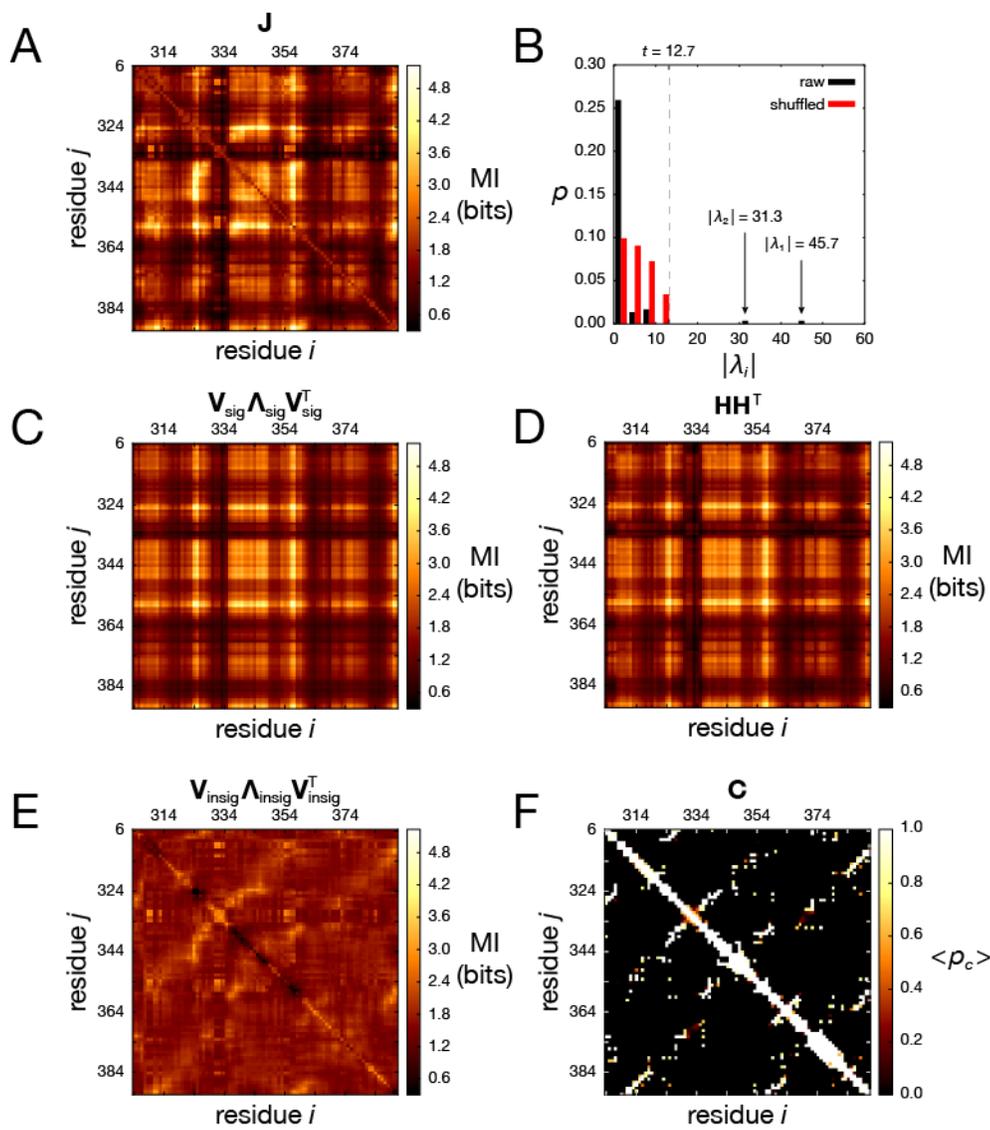


Figure 8. Mutual information between amino acids is pervasive, hierarchical, and dominated by a low-dimensional pattern of coupling. **A.** Average, scaled coordinate MI matrix, \mathbf{I}_s , for the last four residues of the CRIPT ligand and the core domain of PSD-95 PDZ3. The median MI is 1.68 ± 0.67 bits, with minimum and maximum values of 0.29 and 5.23 bits, respectively. A number of positions appear highly correlated with residues throughout the domain (e.g., F325 and D357 on the β_2 and β_4 strands, respectively) while others are largely uncorrelated with other amino acids (e.g., residue G333 in the β_2 - β_3 loop and G364 in the β_4 - β_5 loop). **B.** Eigenspectra for \mathbf{I}_s (black) and 1,000 shuffled versions of it (red). Using the maximum eigenvalue of the shuffled distribution as a cutoff, t , two significant eigenvalues, λ_1 and λ_2 are identified which explain 39% of the variance in the data. **C.** The top two eigenmodes of \mathbf{I}_s were used to calculate another MI matrix, \mathbf{I}_{top} , which accounts for the global pattern of coupling found in \mathbf{I}_s . **D.** The MI matrix \mathbf{I}_{top} as computed using SNMF with factorization rank $k = 2$. **E.** The remaining 86 eigenmodes of \mathbf{I}_s were used to calculate another MI matrix, $\mathbf{I}_{\text{bottom}}$, which accounts for many of the local features \mathbf{I}_s . **F.** The average contact probability matrix over the super-ensemble, \mathbf{C} , for data set 1-1. Each pixel

represents the probability of observing a contact of observing a contact between any of the atoms in two residues, where a contact is defined as any distance less than the sum of the van der Waals radii of the atoms plus 20%.

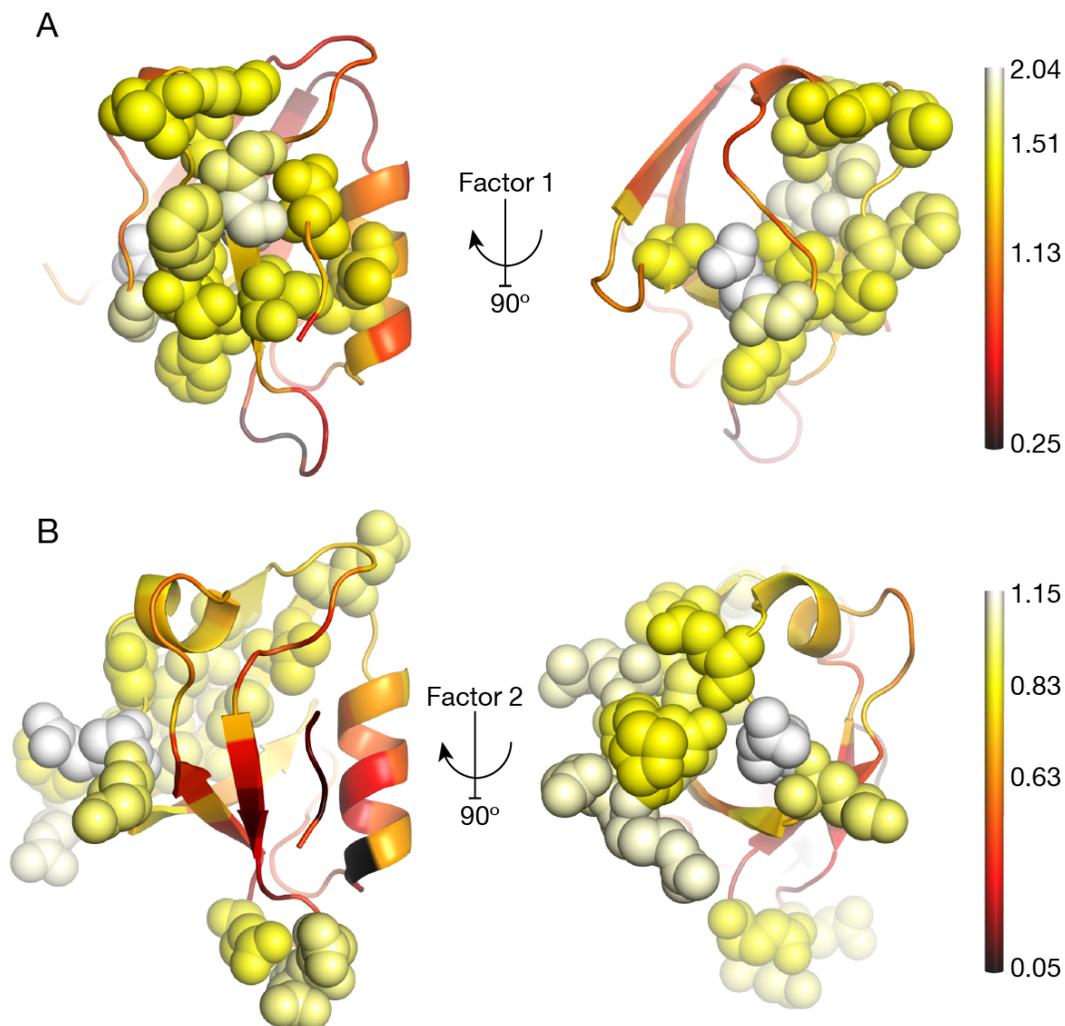


Figure 9. Coordinate MI matrix factors for a rank two decomposition map to mostly contiguous regions of the protein structure, span multiple elements of secondary structure, and are dominated by different sets of residues. Mapping of SNMF factor magnitudes for a rank-2 decomposition of the scaled, mean PSD-95 PDZ3 coordinate MI matrix to the protein structure. Residues are colored by factor magnitude, with spheres drawn for positions in the 80th percentile. **A.** Factor 1 is dominated by contributions from residues running from the C-terminal position of the ligand to both the termini of the domain and the α_1 helix. **B.** Factor 2 is comprised primarily of positions running along the “backside” of the domain. Several positions on the β_2 - β_3 also emerge in this factor; upon decomposition with a higher rank, these positions separate into another factor.

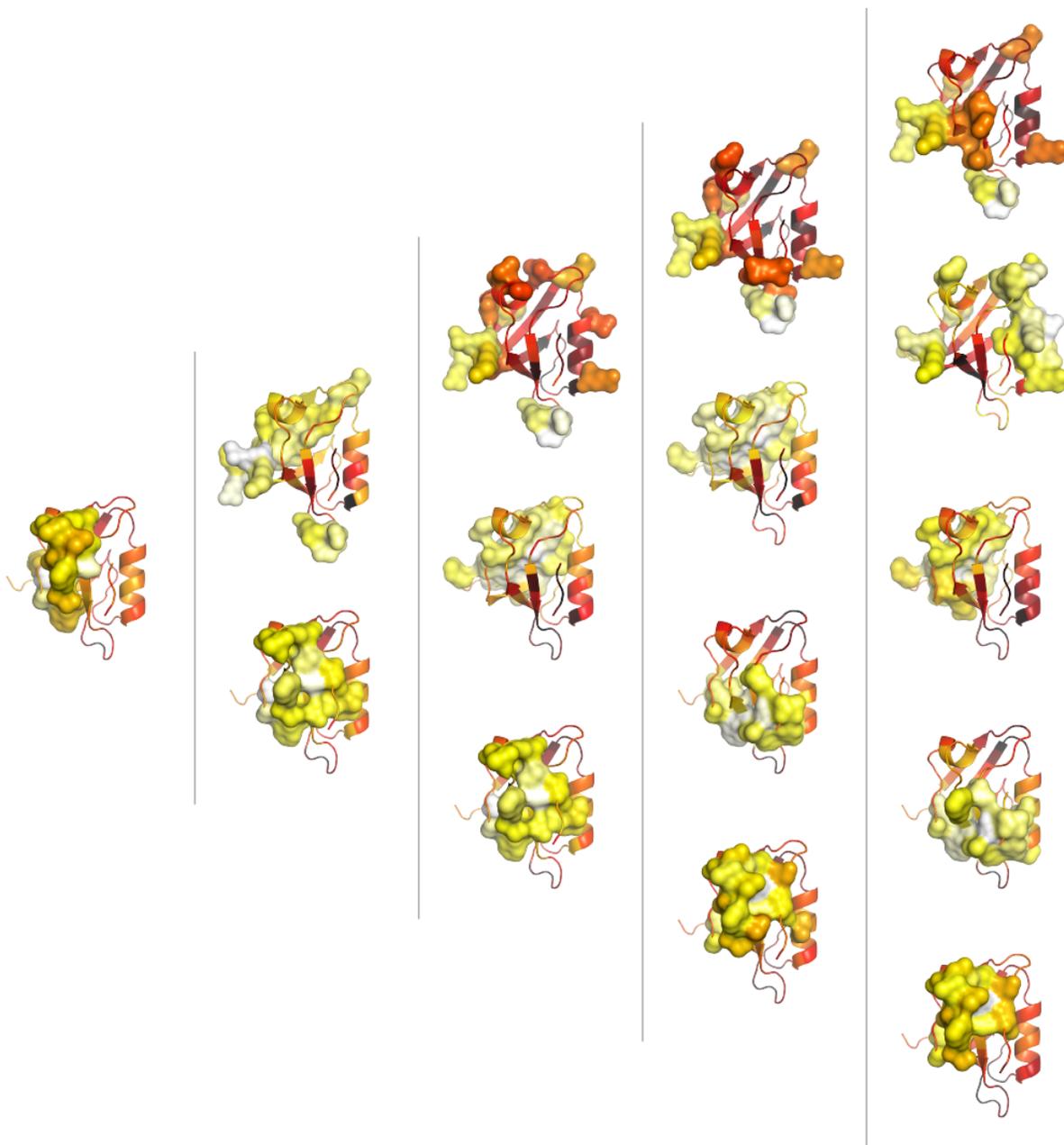


Figure 10. Factorization of PSD-95 PDZ3 coordinate MI matrix with increasing rank reveals hierarchical nature of the data. Mapping of SNMF factor magnitudes for a rank- k decomposition of the scaled, mean PSD-95 PDZ3 coordinate MI matrix to the protein structure for k from 1 to 6. Residues are colored by factor magnitude, with spheres drawn for positions in the 80th percentile. Factorization beyond $k = 5$ yields further breakdown of residue groups, although they become increasingly overlapping.

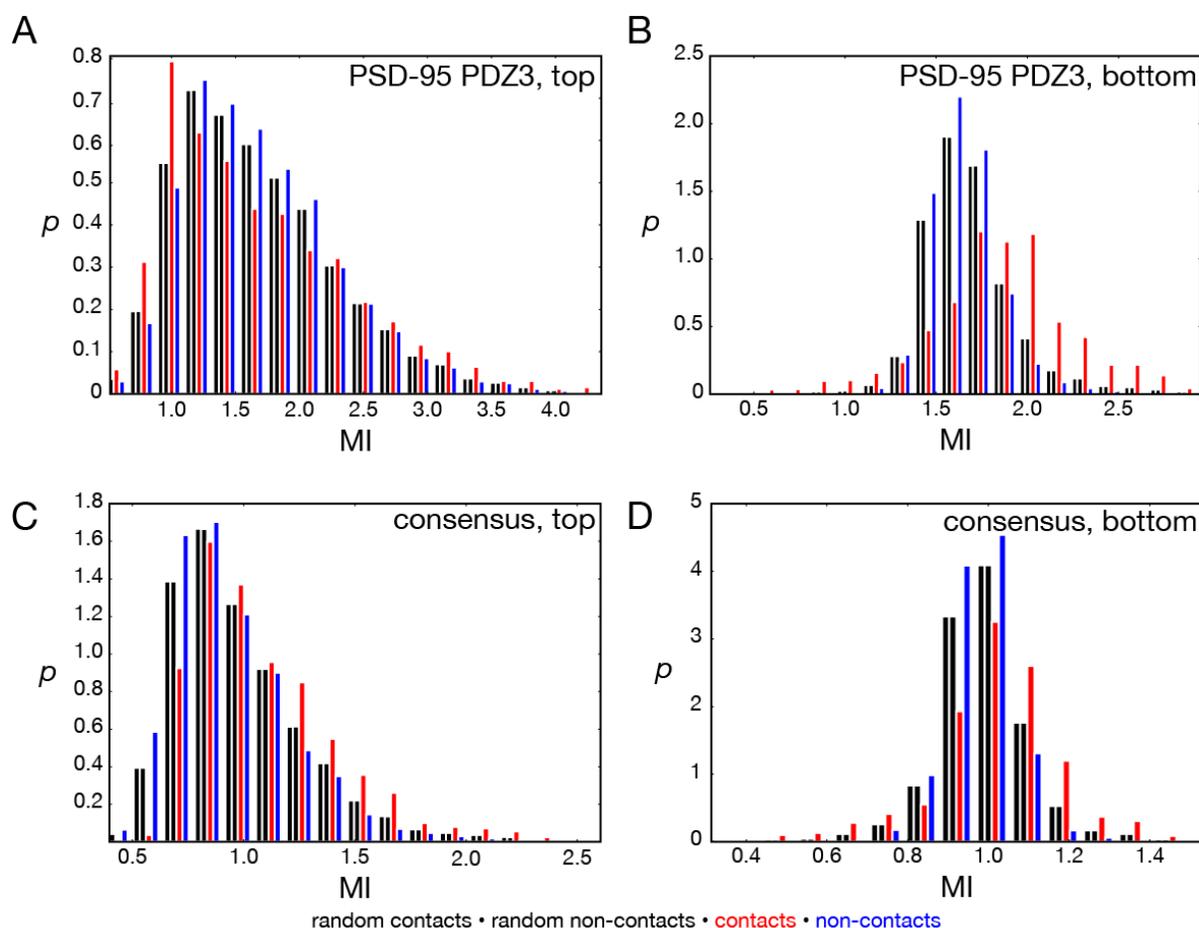


Figure 11. Contacting residues in PSD-95 PDZ3 show enriched MI in weak modes of the coordinate MI matrix. Distribution of MI differs between top and bottom reconstructions of the scaled, mean PSD-95 PDZ3 coordinate mutual information matrix with mean and standard deviation of 1.68 ± 0.27 bits (top row) as well as the top and bottom reconstructions of the consensus coordinate mutual information with mean and standard deviation of 0.99 ± 0.13 bits (bottom row). Distributions of MI for contacting residues (blue) and non-contacting residues (red) are shown. Distributions for a random sampling (10,000 random subsets) of the matrix equal to the number of contacts or noncontacts are also shown (black). **A.** Comparison of distributions of MI for contacting (1.68 ± 0.73 bits) and non-contacting positions (1.68 ± 0.60 bits) in the MI matrix constructed from the two modes with the largest contribution to the variance of the MI matrix. Neither distribution is significantly different from expectation for random sampling by Welch's unequal variances *t*-test based on average *p*-value ($\langle p \rangle$) over all samplings (contacts, $\langle p \rangle = 0.66$; non-contacts, $\langle p \rangle = 0.80$). **B.** Comparison of distributions of MI for contacting (1.85 ± 0.43 bits) and non-contacting (1.64 ± 0.20 bits) positions in the MI matrix constructed from the 86 remaining modes of the MI matrix. MI distributions for both contacting and non-contacting residues are significantly different from expectation based on random sampling by Welch's unequal variances *t*-test (both, $\langle p \rangle < 0.01$). **C.** As **A**, but for the top five modes of the consensus coordinates matrix. MI distributions for both contacting (1.09 ± 0.34 bits) and non-contacting residues (0.93 ± 0.28 bits) are significantly different from expectation based on random sampling by Welch's unequal variances *t*-test (both, $\langle p \rangle < 0.01$). **D.** As **B**, but for the bottom 67 modes of

the consensus coordinates matrix. MI distributions for both contacting (1.02 ± 0.17 bits) and non-contacting residues (0.97 ± 0.08 bits) are significantly different from expectation based on random sampling by Welch's unequal variances t -test (both, $\langle p \rangle < 0.01$).

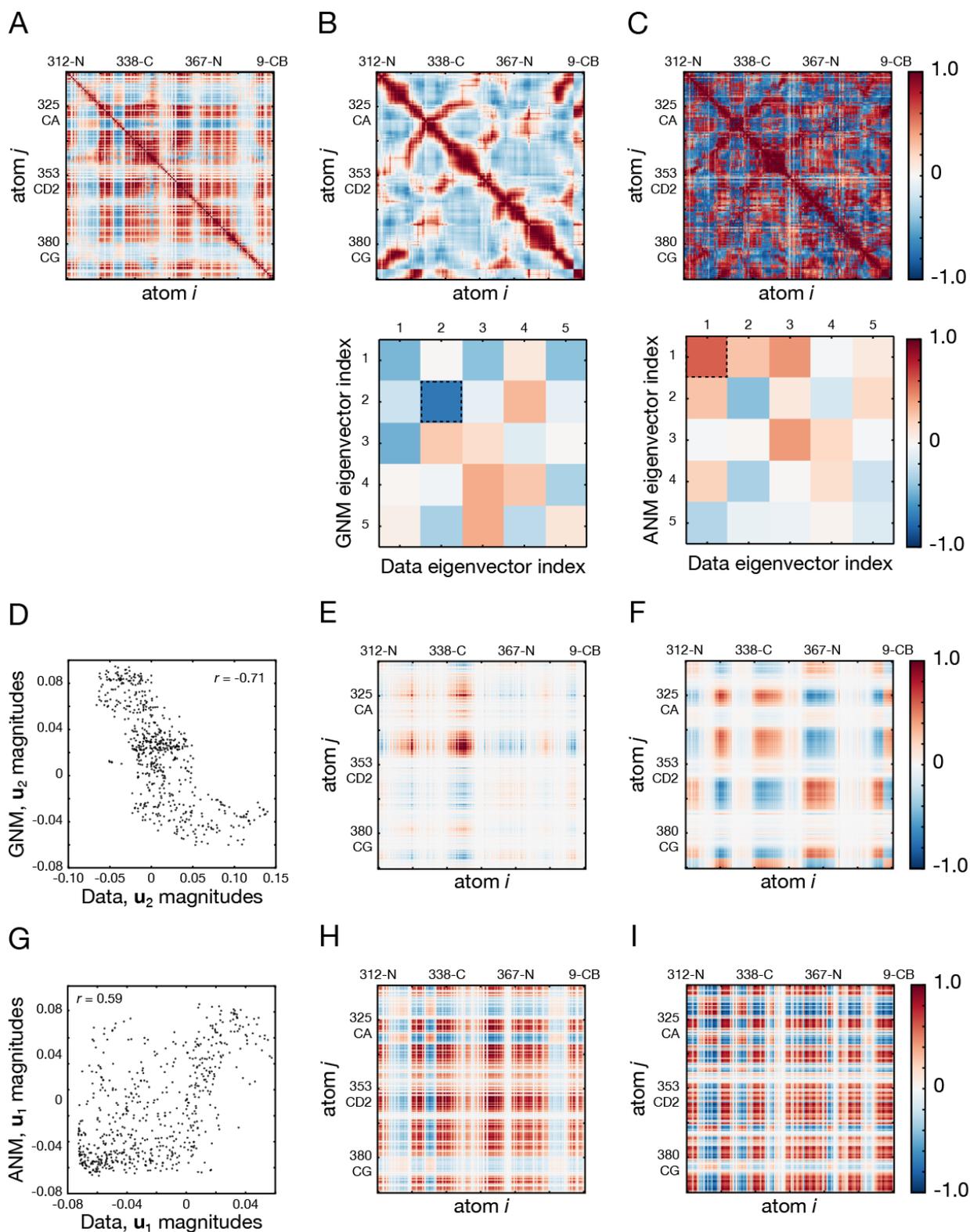
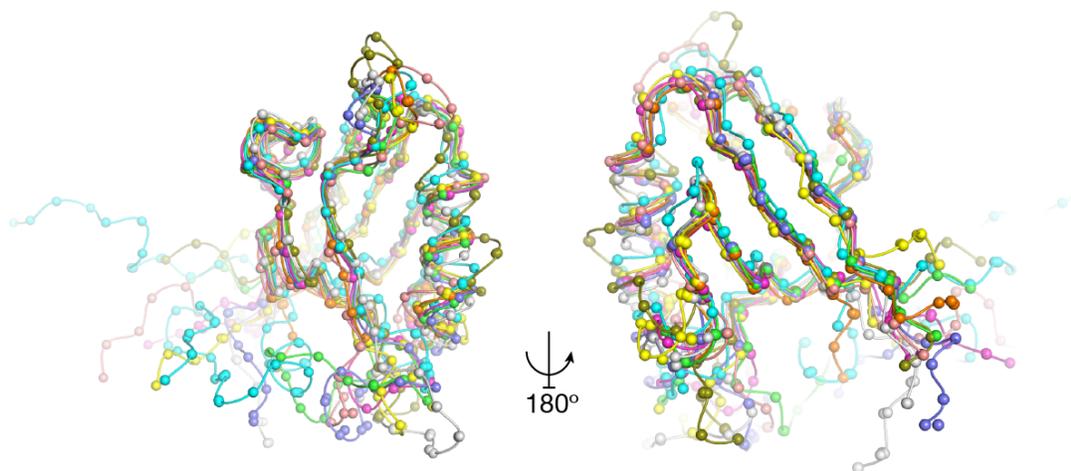


Figure 12. Pairwise atomic correlation coefficients are incompletely described by elastic network models. **A–C.** A series of correlation coefficient matrices were calculated from the PDZ3-CRIPT dataset 1-1 super-ensemble or from a single-conformer starting model; only residues 312–390 of

the domain and residues 6–9 of the ligand were included. Hydrogens were excluded. **A.** A pairwise atomic correlation coefficient matrix, \mathbf{R}_{data} , calculated from the super-ensemble. Correlation coefficients for each Cartesian axis were reduced to a single value by taking the maximum of the absolute values of the correlation coefficients. **B.** A pairwise atomic correlation coefficient matrix, \mathbf{R}_{GNM} , calculated from an all-heavy atom Gaussian network model with force constant $\gamma = 1.0$ and an interaction cutoff of 4.5 Å, determined empirically. The correlation coefficient between \mathbf{R}_{data} and \mathbf{R}_{GNM} is 0.35 (diagonal excluded). Eigenvector overlaps are shown in the bottom sub-panel; the most correlated eigenvector pair (2, 2) is indicated by a dashed box. **C.** A pairwise atomic correlation coefficient matrix, \mathbf{R}_{ANM} , calculated from a heavy atom anisotropic network model with force constant $\gamma = 1.0$ and an interaction cutoff of 6.0 Å, determined empirically. The correlation coefficient between \mathbf{R}_{data} and \mathbf{R}_{ANM} is 0.32 (diagonal excluded). Eigenvector overlaps are shown in the bottom sub-panel; the most correlated eigenvector pair (1, 1) is indicated by a dashed box. **D–F.** Analysis of the most highly correlated mode from the GNM with the experimental data. The average correlation between the top three eigenvectors of each matrix is 0.45 ± 0.22 . **D.** The second eigenvectors of \mathbf{R}_{data} and \mathbf{R}_{GNM} are the most highly correlated pair ($r = -0.71$). The ANM is far less accurate in reproducing the simple pattern of coupling that dominates this mode ($r = -0.42$; data not shown). **E.** The contribution of the second mode to \mathbf{R}_{data} is dominated by coupled motion between atoms at between the beginning of the β_2 strand and the α_1 helix; positive and negative coupling with these regions extends throughout the protein in a relatively weak fashion. **F.** The contribution of the second mode of \mathbf{R}_{GNM} mirrors that of the second mode of \mathbf{R}_{data} . While the boundaries and signs of coupled regions are remarkably consistent with correlations derived from the data, the GNM does not capture subtle differences in coupling between regions well. **G–I.** Analysis of the most highly correlated mode from the ANM with the experimental data. The average correlation between the top three eigenvectors of each matrix is 0.48 ± 0.08 . **G.** The first eigenvectors of \mathbf{R}_{data} and \mathbf{R}_{ANM} are the most correlated ($r = -0.59$). The GNM is far less accurate in reproducing the complex pattern of coupling that dominates this mode ($r = -0.46$; data not shown). **H.** The contribution of the first mode to \mathbf{R}_{data} is complex and reflects coupling between many atoms throughout the protein. **I.** The contribution of the first mode of \mathbf{R}_{ANM} mirrors that of the first mode of \mathbf{R}_{data} . The boundaries and signs of coupled regions are reasonably consistent.

A



B

```

>psd95pdz3_cript
KNYKQTSVGSPEFLGEEIDPREPRRIIVHRGS---TGLGFNIVGGE-----DGEGIFISFILAGGPADLSGELRKGQDILSVNGVDLARNASHEQAAIALKNAGQTVTIIAQYKPEEYSRFEANSRVNSSGRIVTD
>pdlim7pdz_pseudo_sgc1_2q3g
---ITSL-----SMDSFKVVLEGP---APWGFRLQGGKDF-----NVPLSISRLLTPGGKAAQA-GVAVGDWVLSIDGENAGSLTHIEAQNKIRACGERLSLGLSRA-----
>syntenin1pdz2_pseudo_1r6j_chainA
---F-----GAMPRTIIMHKD---STGHVGFIFKN-----GKITSIVKDSSAARN-GLLTEHNICEINGQNVIGLKDSQIADILSTSGTVVTITIMPA-----
>tiam1pdz_apo_3kzd_chainA
-----AGKVTHSIHIEKSDTAADTYGFSLSVVEE-----DGIRRLYVNSVKETGLASKK-GLKAGDEILEINNRAADALNSSMLKDFLS---QPSLGLLVRTYPEL-----
>erbinshortpdz_apo_2h3l_chainA
-----GSMEIRVRVEKD-----PELGFESISGGVGGRNPFPRDDDGIFVTRVQPEGPASK---LLQPGDKIIQANGYSFINIEHGQAVSLKTFQNTVELIIVREVSS-----
>pick1pdz1_pseudo_sgc3_2gzv_chainA
---YYKV---SMVPGKVTLQKD---AQNLIIGISIGGA-----QYCPCLYIVQVFNTPAALDGTVAAGDEITGVNGRSIKGKTKVEAKMIQEVKGEVTIHYNKLQ-----
>sj2bppdz_pseudo_sgc8_2jik_chainA
---ESSI---SMDYLVTEEEINLTRGP---SGLGFNIVGGTDQ---QYVSNDSGIYVSRIKENGAAALDGRLLQEGDKILSVNGQDLKNLLHQDAVDLFRNAGYAVSLRVQHRL-----
>lnx2pdz2_pseudo_sgc5_2vwr_chainA
---EIEL---SMEILQVALHK-RDSGEQLGIKLVRR-----DEPGVFILDLEGGAAQDGRLLSSNDRVLAINGHDLKYGTPELAAQIIQASGERVNLTIARPGKP-----
>mpdzpdz3_pseudo_sgc7_rt_chainA
-----ETSV-----SMSETFDVELTKNV---QGLGITIAGYIG---DKKLEPSGIFVKISITKSSAVEHDGRIQIGDQIIAVDGTNLQGFITNQAVEVLRHTGQTVLLTLMRRG-----

```

Figure 13. Structural alignment and multiple sequence alignment (MSA). **A.** Superposition of PDZ domains from this study, with C_{α} atoms depicted as spheres. **B.** Multiple sequence alignment of all proteins from this study, with residues engaged in ligand-like interactions in the first nine positions. The MSA was created with ProMALS3D and manually adjusted.

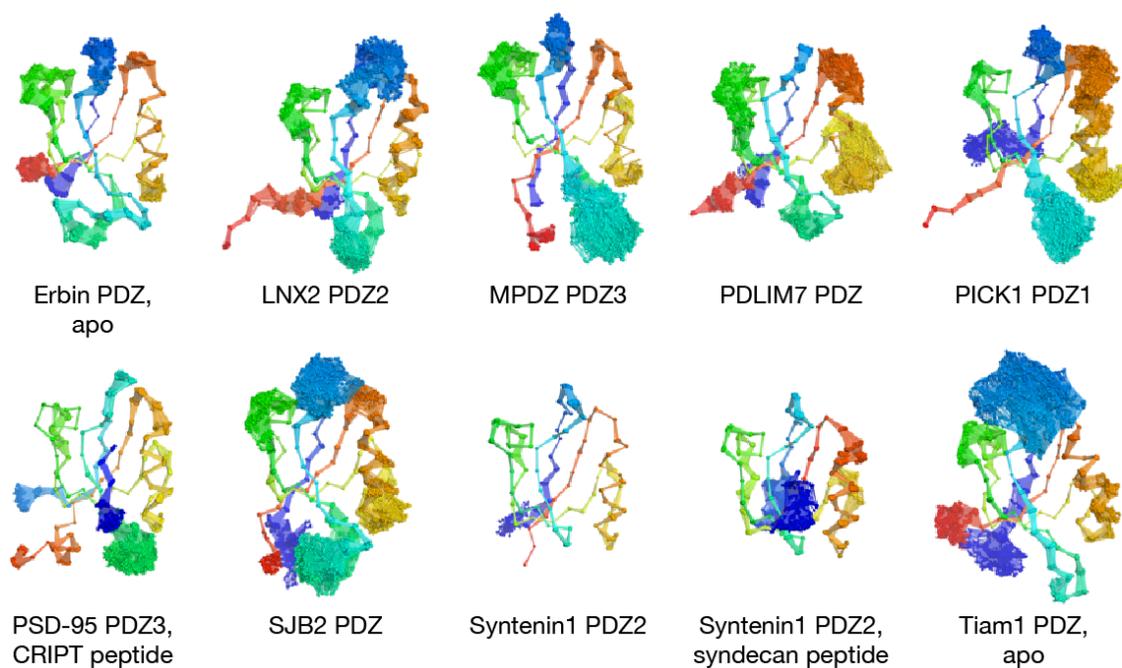


Figure 14. Sample ensembles for all PDZ targets. A single PDZ ensemble is shown for each PDZ domain, with C_{α} atoms shown as spheres. Colors correspond to different regions of the protein; substantial variability in the mobility of these regions is evident across the selection of ensembles.

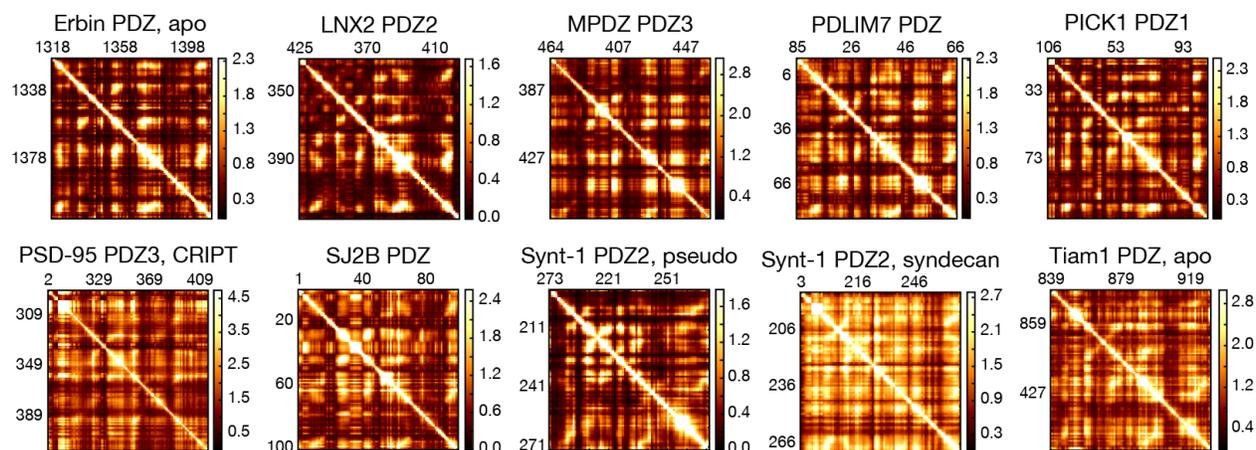


Figure 15. Raw MI matrices calculated from a super-ensemble for one data set of each target. A single, raw MI matrix representative of each PDZ domain target. Matrices are, in general, characterized by a strong diagonal and a blocky pattern of off-diagonal correlations. Several common features are evident, particularly the flower-shaped pattern towards the end of the first half of the diagonal of many of the matrices, or the dark, weakly-correlated bands corresponding with loops throughout the domain. Color maps are scaled to the 5th and 95th percentile of each matrix.

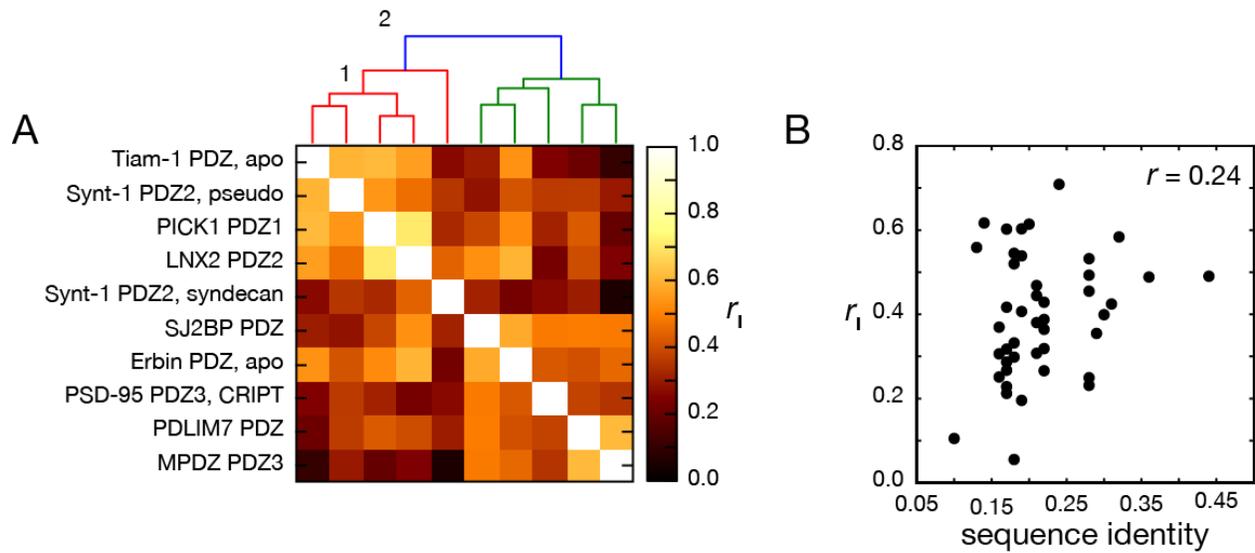


Figure 16. Coordinate MI is correlated between different homologs, but the pattern cannot be explained by sequence identity. **A.** Pearson correlation coefficients, r_1 , between the upper triangles of the average coordinate MI matrices of different PDZ domains at consensus positions of the MSA. On average, $r_1 = 0.40 \pm 0.15$, with minimum and maximum values of 0.05 and 0.72, respectively. Two clusters emerge upon hierarchical clustering of the matrix (Ward's method). **B.** Values of r_1 are only weakly correlated with sequence identity.

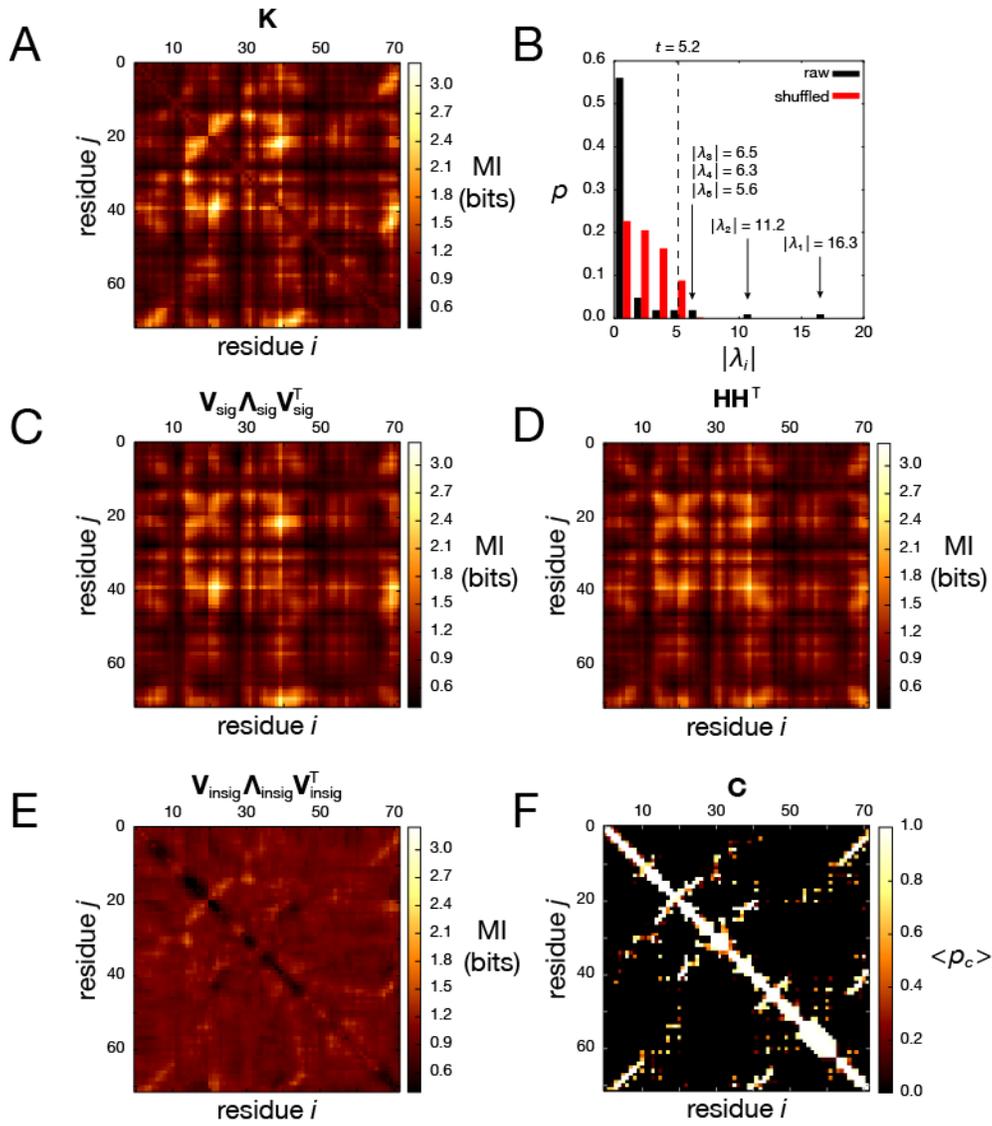


Figure 17. A non-trivial pattern of MI is conserved across homologous PDZ domains. A multiple sequence alignment (MSA) was used to identify conserved positions across ten datasets for nine PDZ domains. **A.** Average coordinate MI matrix, \mathbf{I} , across homologues. The median MI is 0.99 ± 0.34 bits, with minimum and maximum values of 0.39 and 3.24 bits, respectively. **B.** Eigenspectra for \mathbf{I} (black) and 1,000 shuffled equivalents (red). Using the maximum eigenvalue of the shuffled distribution as a cutoff, t , five significant eigenvalues, λ_1 – λ_5 , are identified which explain 54% of the variance in the data. **C.** The top five eigenmodes of \mathbf{I} were used to calculate another MI matrix, \mathbf{I}_{top} , which accounts for the global pattern of coupling found in \mathbf{I} . **D.** The MI matrix \mathbf{I}_{top} as computed using SNMF with factorization rank $k = 5$. **E.** The remaining 67 eigenmodes of \mathbf{I} were used to calculate another MI matrix, $\mathbf{I}_{\text{bottom}}$, which accounts for many of the local features \mathbf{I} . **F.** The average contact probability matrix over all super-ensembles, \mathbf{C} . Each pixel represents the probability of observing a contact between any of the atoms in two residues at any conserved position across homologs.

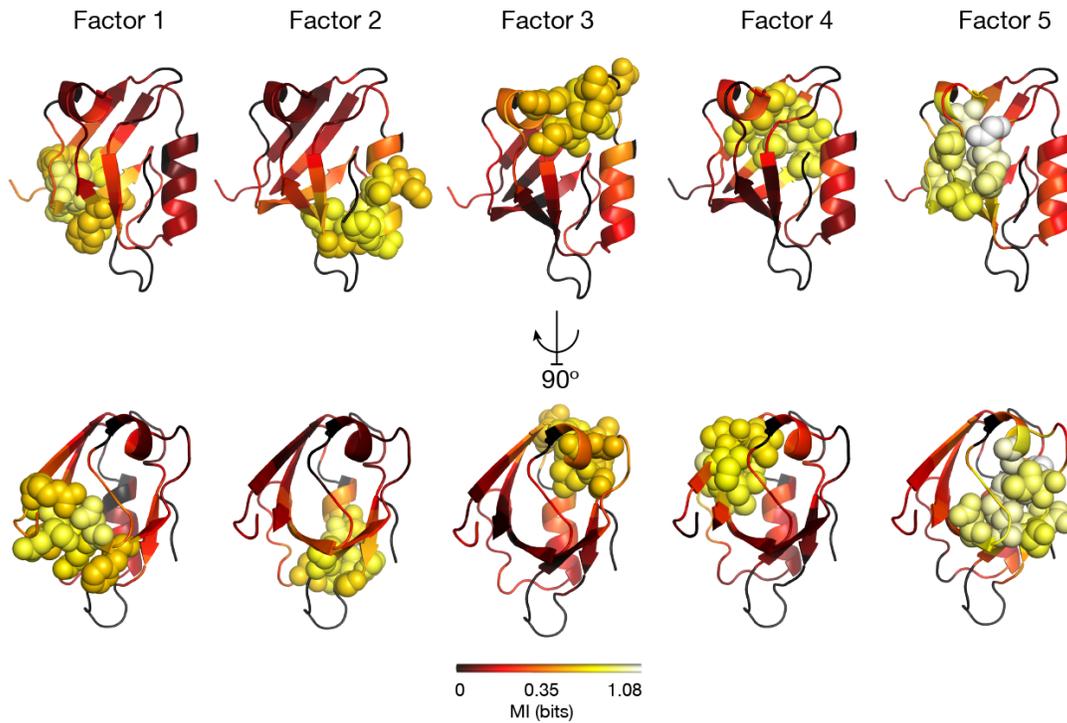


Figure 18. Consensus MI factors localize to distinct regions of the PDZ fold. Non-negative matrix factorization was performed on the consensus coordinate MI matrix \mathbf{I} assuming rank five, yielding matrix \mathbf{H} . Columns of \mathbf{H} were mapped to the structure of PSD-95 PDZ3 using the multiple sequence alignment. Colors scale with increasing magnitude of an element in any given factor, with residues in the 90th percentile of a given factor shown as spheres. Gapped positions in the alignment are colored grey. Strikingly, each factor maps to a distinct region of the PDZ domain, with highly-correlated residues spanning multiple elements of secondary structure.

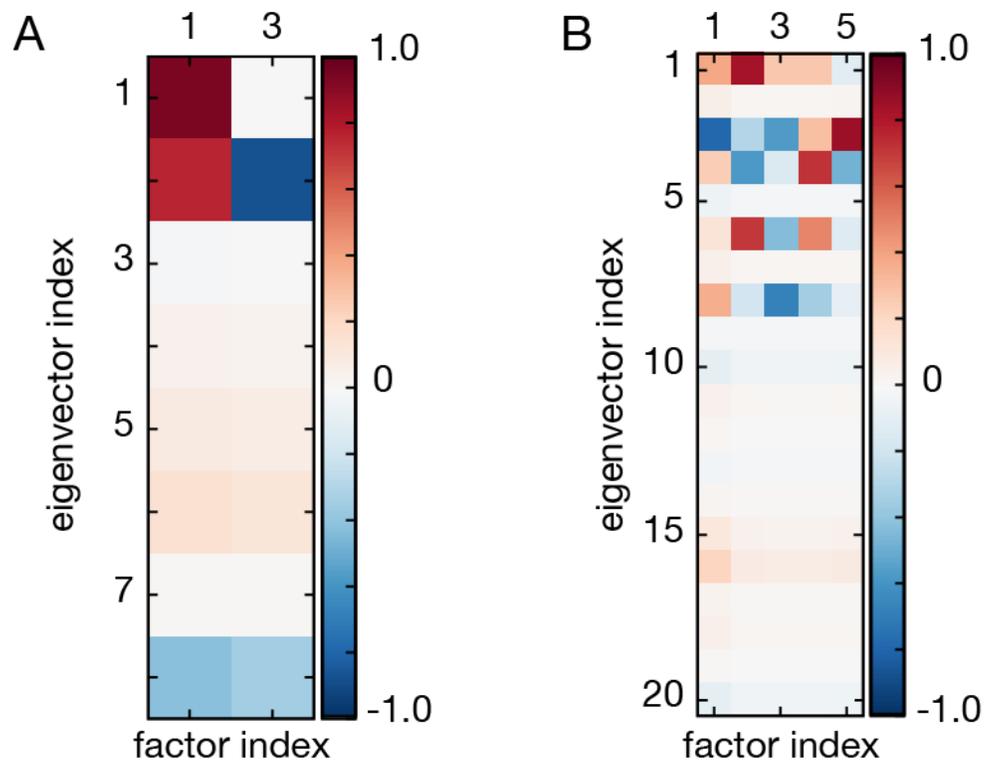


Figure 19. The eigendecomposition and symmetric non-negative matrix factorization of mutual information matrices separates information along different bases. Correlation coefficients between eigenvectors and factors produced by SymNMF were calculated for a rank-two factorization of the core PSD-95 PDZ3 MI matrix (**A**) and a rank-five decomposition of the consensus MI matrix (**B**). In both cases, factors correlate well with a subset of the top eigenvectors, but not individual eigenvectors. In the case of the consensus matrix decomposition, information corresponding to eigenvectors two, five, and seven appear to be systematically absent from the factors.

5 Tables

Table 1: Crystallographic statistics

Protein	PSD-95 PDZ3-CRIP1	PSD-95 PDZ3-CRIP2	PSD-95 PDZ3-CRIP3	PSD-95 PDZ3-CRIP4	PSD-95 PDZ3-CRIP5	PSD-95 PDZ3-CRIP6	PSD-95 PDZ3-CRIP7	PSD-95 PDZ3-CRIP8	PSD-95 PDZ3-CRIP9	PSD-95 PDZ3-CRIP10
Dataset	ds1-1	ds1-1	ds4-1	ds4-2	ds4-3	ds4-4	ds4-5	ds4-6	ds4-7	ds4-8
Wavelength	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945
Resolution Range	40.71–1.33 (1.376–1.33)	30.24–1.335 (1.383–1.335)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)	37.04–1.35 (1.298–1.35)
Space group	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2	P 41 3 2
Unit cell	91.028 91.028 91.028 90.90 90.90	90.726 90.726 90.726 90.90 90.90	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733	90.733 90.733 90.733 90.733 90.733
Total reflections	1779089	1302528	935573	935573	935573	935573	935573	935573	935573	935573
Unique reflections	27677 (740)	26502 (404)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)
Redundancy	6.4	6.8	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7
Completeness (%)	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.8	99.8
Mean I/sigma(I)	82.420 (1.093)	89.220 (0.719)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)	77.913 (0.925)
Wilson B-factor	14.26	13.88	14.19	14.19	14.19	14.19	14.19	14.19	14.19	14.19
R _{int}	0.0710	0.0490	0.0510	0.0510	0.0510	0.0510	0.0510	0.0510	0.0510	0.0510
R _{meas}	0.0710	0.0500	0.0520	0.0520	0.0520	0.0520	0.0520	0.0520	0.0520	0.0520
R _{rim}	0.0120	0.0150	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100
Mosaicity range (°)	0.05–0.09	0.08–0.22	0.04–0.08	0.04–0.08	0.04–0.08	0.04–0.08	0.04–0.08	0.04–0.08	0.04–0.08	0.04–0.08
Internal Non-isomorphism	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010
HKL2000 R.D. coefficient	0.15	0.17	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
CC1/2 (best shell)	0.478	0.222	0.244	0.244	0.244	0.244	0.244	0.244	0.244	0.244
CC1/2 (best shell) in refinement	0.478	0.222	0.244	0.244	0.244	0.244	0.244	0.244	0.244	0.244
Reflections used in refinement	27675 (740)	26500 (404)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)	25988 (486)
Reflections used for R-free	1500 (41)	1500 (23)	1500 (28)	1500 (28)	1500 (28)	1500 (28)	1500 (28)	1500 (28)	1500 (28)	1500 (28)
Toffset	1.0	1.0	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
PTLS	0.6	0.8	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
R _{work}	0.1188 (0.2607)	0.1231 (0.2114)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)	0.1159 (0.2352)
R _{free}	0.1402 (0.2947)	0.1549 (0.1891)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)	0.1372 (0.2473)
R _{work, super-ensemble}	0.1517 ± 0.0039	0.1563 ± 0.0049	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024	0.1543 ± 0.0024
R _{free, super-ensemble}	0.1901 ± 0.0057	0.2114 ± 0.0067	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035	0.1982 ± 0.0035
Number of non-hydrogen atoms	1618.0	1732.0	1623.0	1623.0	1623.0	1623.0	1623.0	1623.0	1623.0	1623.0
...macromolecules	1512.0	1634.0	1598.0	1598.0	1598.0	1598.0	1598.0	1598.0	1598.0	1598.0
...protein residues	110.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
RMS (bonds)	0.0100	0.0140	0.0120	0.0120	0.0120	0.0120	0.0120	0.0120	0.0120	0.0120
RMS (angles)	1.280	1.800	1.600	1.600	1.600	1.600	1.600	1.600	1.600	1.600
Ramachandran favored (%)	98.0	98.0	95.0	95.0	95.0	95.0	95.0	95.0	95.0	95.0
Ramachandran allowed (%)	2.1	2.9	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6
Ramachandran outliers (%)	0.0	1.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Rotamer outliers (%)	4.3	7.2	6.1	6.1	6.1	6.1	6.1	6.1	6.1	6.1
Cis/Trans (%)	5.3	3.7	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
Average B-factor	23.8	23.1	22.1	22.1	22.1	22.1	22.1	22.1	22.1	22.1
...macromolecules	24.4	22.5	21.9	21.9	21.9	21.9	21.9	21.9	21.9	21.9
...solvent	34.8	32.6	37.5	37.5	37.5	37.5	37.5	37.5	37.5	37.5

Table 1: Crystallographic statistics

Protein	PSD-95 PDZ3-CRIPT (3MW re-refine)	Syntenin (pseudo)	Syntenin (pseudo)	Syntenin-syndecan	
Dataset	ds1	ds1-1	ds3-1	ds3-1	
Wavelength	0.9075	0.97945	0.97945	0.97945	
Resolution Range	29.66–1.255 (1.3–1.255)	30.21–1.044 (1.081–1.044)	24.03–1.02 (1.057–1.02)	32.99–1.344 (1.369–1.344)	
Space group	P 1 2 1 1	P 1 2 1 1	P 1 2 1 1	C 1 2 1	
Unit cell	42.152 47.602 43.322 90.92 08 90.25 818 39.566 32.363 90.000 111.027 90.000 25.818 39.740 32.394 90.000 111.002 90.000 25.792 39.604 32.330 90.000 111.286 90.000 59.020 55.490 50.161 90.000 98.935 90.000 98619	191995	120339	130303	98619
Total reflections	39634 (643)	20799 (70)	21519 (70)	24212 (314)	
Redundancy	2.2	2	2	2	
Completeness (%)	0.95	0.75	0.9	0.62	
Mean I/sig(I)	23.598 (0.950)	25.128 (1.000)	30.600 (1.244)	20.705 (1.103)	
Wilson B-factor	14.2	15.47	15.61	14.48	
R merge	0.0520	0.051	0.076	0.057	
R meas	0.0600	0.059	0.083	0.066	
R pim	0.0280	0.03	0.033	0.037	
Mosaicity range (°)	0.06–0.14	0.26–0.49	0.20–0.60	0.18–0.81	
Internal Non-isomorphism	0.0060	0.005	0.001	0.003	
HKL2000 R.D. coefficient	0.0000	0.13	0.3	0.12	
CC1/2 (last shell)	0.820	0.67	0.814	0.513	
CC1/2 (entire)	0.910	0.78	0.89	0.64	
R (last shell) in refinement	0.100	0.10	0.10	0.10	
R (entire) in refinement	0.080	0.08	0.08	0.08	
Reflections used for R-free	39633 (643)	20798 (70)	21518 (70)	24210 (314)	
Totrefact	2017 (15)	1503 (6)	1492 (6)	1500 (10)	
pTLS	1	7	12	5	
R work	0.8	0.85	0.9	0.7	
R free	0.1145 (0.2089)	0.1080 (0.1287)	0.1152 (0.135)	0.1319 (0.3136)	
R work, super-ensemble	0.1462 (0.2132)	0.1287 (0.7085)	0.1374 (0.5393)	0.1734 (0.4118)	
R free, super-ensemble	0.1428 ± 0.0030	0.1161 ± 0.0030	0.1396 ± 0.0023	0.1440 ± 0.0026	
Number of non-hydrogen atoms	0.1906 ± 0.0051	0.1375 ± 0.0038	0.1662 ± 0.0040	0.2005 ± 0.0042	
...macromolecules	1709.0	1111	986	2007	
...protein residues	1516.0	1028	923	1881	
RMS (bonds)	0.002	0.002	0.002	0.002	
RMS (angles)	0.0098	0.014	0.015	0.011	
Ramachandran favored (%)	1.3	1.49	1.65	0.94	
Ramachandran allowed (%)	100.0	99	98	98	
Ramachandran outliers (%)	0.0	0.75	1.6	2.5	
Rotamer outliers (%)	0.0	0	0	0	
Cisalscore	1.3	2.5	2.8	4.1	
Average B-factor	22.0	18.73	17.13	3.69	
...macromolecules	20.7	17.66	39.25	18.22	
...solvent	32.6	31.96	31.01	36.61	

Table 1: Crystallographic statistics

Protein	Tiam1 (apo)	Erbin (apo)	PDLIM7 PDZ (SGC1)	PICK1 PDZ1 (SGC3)	LNX2 PDZZ (SGC5)	LNX2 PDZZ (SGC6)
Dataset	d81-1	d82-1	d82-1	d81-1	d81	d83
Wavelength	0.97945	0.97945	0.97945	0.97945	0.97945	0.97945
Resolution Range	34.32-1.3 (1.346-1.3)	36.23-1.358 (1.406-1.358)	40.18-1.163 (1.205-1.163)	29.56-1.088 (1.137-1.088)	32.4-1.082 (1.121-1.082)	34.38-1.069 (1.138-1.069)
Space group	P 32 2 1	C 1 2 1	P 2 1 2 1	C 1 2 1	C 1 2 1	C 1 2 1
Unit cell	45.173 45.173 71.5 90.90 120.623 151.61 288.90 117.017 90.46 832.56 103.57 588.90 90.90	88480	238653	99916	96300	117 400 90.000
Total reflections	312981	25500 (95)	42526 (186)	26886 (112)	27453 (71)	25948 (96)
Unique reflections	1624 (97)	2500 (95)	42526 (186)	26886 (112)	27453 (71)	25948 (96)
Redundancy	4.2	2.9	0.81	3	3	1.3
Completeness (%)	0.7	0.9	0.81	0.7	0.7	0.7
R_merit	37.223 (0.7620)	16.756 (0.142)	33.461 (1.191)	29.000 (1.000)	34.025 (0.213)	45.836 (1.100)
Wilson B-factor	14.15	15.54	15.57	15.6	15.62	15.57
R_merge	0.075	0.063	0.05	0.039	0.026	0.032
R_meas	0.076	0.075	0.055	0.047	0.031	0.035
R_pim	0.018	0.041	0.023	0.026	0.017	0.014
Mosicity range (°)	0.13-0.18	0.18-0.87	0.18-0.74	0.10-0.28	0.13-0.20	0.17-0.25
Internal Non-isomorphism	0.011	0.001	0.001	0.001	0.001	0.001
HKL2000 R.D. coefficient	0.03	0.39	0.51	0.08	0.03	0.1
CC1/2 (last shell)	0.177	0.611	0.658	0.381	0.367	0.435
CC1/2 (all shells)	0.177	0.611	0.658	0.381	0.367	0.435
R_free (estimated in refinement)	1624 (37)	25500 (95)	42512 (186)	26885 (112)	27449 (71)	25939 (96)
R_free (used for R-free)	1314 (50)	1500 (6)	1501 (6)	1500 (7)	1088 (4)	1087 (2)
Totfset	11	3	2	8	2	6
pTLS	0.5	0.75	0.35	0.35	0.3	0.15
R_work	0.1045 (0.1662)	0.1530 (0.4326)	0.1402 (0.3235)	0.1158 (0.2201)	0.1304 (0.3940)	0.1281 (0.4119)
R_free	0.1433 (0.1866)	0.1903 (0.8338)	0.1686 (0.4387)	0.1332 (0.1395)	0.1527 (0.5872)	0.1573 (0.4063)
R_work, super-ensemble	0.1440 ± 0.0026	0.1392 ± 0.0013	0.1480 ± 0.0059	0.1326 ± 0.0068	0.1312 ± 0.0050	0.1382 ± 0.0086
R_free, super-ensemble	0.2005 ± 0.0042	0.1872 ± 0.0025	0.1967 ± 0.0086	0.1701 ± 0.0095	0.1742 ± 0.0067	0.1716 ± 0.0099
Number of non-hydrogen atoms	1134	2128	1982	1157	985	934
...macromolecules	984	2056	1855	1085	888	860
...protein residues	176	176	176	176	176	176
RMS (bonds)	0.013	0.007	0.009	0.015	0.01	0.014
RMS (angles)	1.36	1.22	1.21	1.64	1.28	1.44
Ramachandran favored (%)	98	96	99	99	97	99
Ramachandran allowed (%)	1.6	4.2	0.84	1.4	2.7	0.84
Ramachandran outliers (%)	0	0	0.42	0	0	0
Rotamer outliers (%)	3.6	2.1	5.5	4.3	3.1	1.1
Cisalscore	7.5	1.46	3.22	5.84	1.66	2.3
Average B-factor	22.33	20.03	20.23	24.61	24.61	21.76
...macromolecules	19.13	19.79	19.28	20.08	23.6	20.27
...solvent	45.03	26.93	32.07	36.31	36.16	36.73

Table 1: Crystallographic statistics

	LNK2 PDZ2 (SGC5)	LNK2 PDZ3 (SGC7)	SUBP PDZ (SGC8)
Protein			
Dataset	d64	d61-1	d61-1
Wavelength	0.97945	0.97945	0.97945
Resolution Range	34.39–1.066 (1.104–1.066)	29.95–1.16 (1.202–1.16)	35.96–1.213 (1.257–1.213)
Space group	C 1 2 1	P 1 2 1	P 1 2 1
Unit cell	64.908 39.264 38.758 90.000 117.462 90.000	42.382 42.382 107.772 90.90 36.732 38.864 61.103 90.10 1777.90	129306
Total reflections	127042	545707	129306
Unique reflections	29842 (22)	29859 (32)	35310 (26)
Redundancy	4.9	4.8	3.1
Completeness (%)	97	98	97
Mean I/sigma(I)	32.207 (0.944)	47.582 (0.694)	23.696 (1.233)
Wilson B-factor	15.43	15.6	14.9
R merge	0.035	0.063	0.044
R meas	0.04	0.065	0.053
R pim	0.019	0.016	0.029
Mosaicity range (°)	0.16–0.36	0.08–0.26	0.23–0.54
Internal Non-isomorphism	0.011	0.001	0.001
HKL2000 R.D. coefficient	0.488	0.22	0.22
CC1/2 (last shell)	0.71	0.56	0.549
C _{1/2} (last shell)	0.11	0.06	0.06
Reflections used in refinement	29841 (2)	29859 (32)	35298 (256)
Reflections used for R-free	1149 (2)	1500 (18)	1498 (10)
Toffset	1	8	1
pTLS	0.4	0.6	0.45
R work	0.1201 (0.5897)	0.1191 (0.2571)	0.1416 (0.3412)
R free	0.1383 (0.8639)	0.1406 (0.3272)	0.1796 (0.4510)
R work, super-ensemble	0.1269 ± 0.0021	0.1459 ± 0.0029	0.1426 ± 0.0039
R free, super-ensemble	0.1574 ± 0.0038	0.1793 ± 0.0047	0.1982 ± 0.0055
Number of non-hydrogen atoms	917	1195	2520
...macromolecules	844	1121	2418
...protein residues	815	1095	2375
RMS (bonds)	0.015	0.014	0.01
RMS (angles)	1.65	1.55	1.27
Ramachandran favored (%)	99	97	94
Ramachandran allowed (%)	0.95	2.8	3.7
Ramachandran outliers (%)	0	0	2
Rotamer outliers (%)	3.3	4.7	2.2
Cis/score	0.58	3.95	4.76
Average B-factor	23.28	25.63	20.47
...macromolecules	22.33	24.65	20.09
...solvent	34.27	40.42	29.49

6 Works cited

1. Karplus, M. & Kuriyan, J. Molecular dynamics and protein function. *Proc Natl Acad Sci USA* **102**, 6679–6685 (2005).
2. Onuchic, J. N., Luthey-Schulten, Z. & Wolynes, P. G. Theory of protein folding: the energy landscape perspective. *Annu Rev Phys Chem* **48**, 545–600 (1997).
3. Perutz, M. F. Stereochemistry of cooperative effects in haemoglobin. *Nature* **228**, 726–739 (1970).
4. Boehr, D. D., McElheny, D., Dyson, H. J. & Wright, P. E. The dynamic energy landscape of dihydrofolate reductase catalysis. *Science* **313**, 1638–1642 (2006).
5. Fraser, J. S. *et al.* Hidden alternative structures of proline isomerase essential for catalysis. *Nature* **462**, 669–U149 (2009).
6. Monod, J., Wyman, J. & Changeux, J. On Nature of Allosteric Transitions — a Plausible Model. *J Mol Biol* **12**, 88–118 (1965).
7. Koshland, D. E., Némethy, G. & Filmer, D. Comparison of experimental binding data and theoretical models in proteins containing subunits. *Biochemistry* **5**, 365–385 (1966).
8. Cooper, A. & Dryden, D. T. F. Allostery without conformational change. *Eur Biophys J* **11**, 103–109 (1984).
9. Smock, R. G. & Gierasch, L. M. Sending signals dynamically. *Science* **324**, 198–203 (2009).
10. Sekhar, A. & Kay, L. E. NMR paves the way for atomic level descriptions of sparsely populated, transiently formed biomolecular conformers. *Proc Natl Acad Sci USA* **110**, 12867–12874 (2013).
11. Hanson, J. A. *et al.* Illuminating the mechanistic roles of enzyme conformational dynamics. *Proc Natl Acad Sci USA* **104**, 18055–18060 (2007).
12. Ren, Z. *et al.* A molecular movie at 1.8 Å resolution displays the photocycle of photoactive yellow protein, a eubacterial blue-light receptor, from nanoseconds to seconds. *Biochemistry* **40**, 13788–13801 (2001).
13. Schotte, F. *et al.* Watching a Protein as it Functions with 150-ps Time-Resolved X-ray Crystallography. *Science* **300**, 1944–1947 (2003).
14. Barends, T. R. M. *et al.* Direct observation of ultrafast collective motions in CO myoglobin upon ligand dissociation. *Science* **350**, 445–450 (2015).
15. Smith, J. L., Hendrickson, W. A., Honzatko, R. B. & Sheriff, S. Structural heterogeneity in protein crystals. *Biochemistry* **25**, 5018–5027 (1986).
16. Kuriyan, J., Petsko, G. A., Levy, R. M. & Karplus, M. Effect of anisotropy and anharmonicity on protein crystallographic refinement. An evaluation by molecular dynamics. *J Mol Biol* **190**, 227–254 (1986).
17. Kuzmanic, A., Pannu, N. S. & Zagrovic, B. X-ray refinement significantly underestimates the level of microscopic heterogeneity in biomolecular crystals. *Nat Commun* **5**, 3220 (2014).
18. van den Bedem, H., Dhanik, A., Latombe, J. C. & Deacon, A. M. Modeling discrete heterogeneity in X-ray diffraction data by fitting multi-conformers. *Acta Crystallogr D Biol Crystallogr* **65**, 1107–1117 (2009).
19. Keedy, D. A., Fraser, J. S. & van den Bedem, H. Exposing Hidden Alternative Backbone Conformations in X-ray Crystallography Using qFit. *Plos Comput Biol* **11**, e1004507

- (2015).
20. Lang, P. T. *et al.* Automated electron-density sampling reveals widespread conformational polymorphism in proteins. *Protein Sci* **19**, 1420–1431 (2010).
 21. Burnley, B. T., Afonine, P. V., Adams, P. D. & Gros, P. Modelling dynamics in protein crystal structures by ensemble refinement. *elife* **1**, e00311 (2012).
 22. Gros, P., van Gunsteren, W. F. & Hol, W. G. Inclusion of thermal motion in crystallographic structures by restrained molecular dynamics. *Science* **249**, 1149–1152 (1990).
 23. Pitera, J. W. & Chodera, J. D. On the Use of Experimental Observations to Bias Simulated Ensembles. *J Chem Theory Comput* **8**, 3445–3451 (2012).
 24. Tirion, M. Large Amplitude Elastic Motions in Proteins from a Single-Parameter, Atomic Analysis. *Phys. Rev. Lett.* **77**, 1905–1908 (1996).
 25. Atilgan, A. R. *et al.* Anisotropy of fluctuation dynamics of proteins with an elastic network model. *Biophysj* **80**, 505–515 (2001).
 26. Lange, O. F. & Grubmüller, H. Generalized correlation for biomolecular dynamics. *Proteins* **62**, 1053–1061 (2006).
 27. McClendon, C. L., Friedland, G., Mobley, D. L., Amirkhani, H. & Jacobson, M. P. Quantifying Correlations Between Allosteric Sites in Thermodynamic Ensembles. *J Chem Theory Comput* **5**, 2486–2502 (2009).
 28. Songyang, Z. *et al.* Recognition of unique carboxyl-terminal motifs by distinct PDZ domains. *Science* **275**, 73–77 (1997).
 29. Stiffler, M. A. *et al.* PDZ domain binding selectivity is optimized across the mouse proteome. *Science* **317**, 364–369 (2007).
 30. Doyle, D. A. *et al.* Crystal structures of a complexed and peptide-free membrane protein-binding domain: molecular basis of peptide recognition by PDZ. *Cell* **85**, 1067–1076 (1996).
 31. Fuentes, E. J., Gilmore, S. A., Mauldin, R. V. & Lee, A. L. Evaluation of energetic and dynamic coupling networks in a PDZ domain protein. *J Mol Biol* **364**, 337–351 (2006).
 32. Whitney, D. S., Peterson, F. C. & Volkman, B. F. A conformational switch in the CRIB-PDZ module of Par-6. *Structure* **19**, 1711–1722 (2011).
 33. Whitney, D. S. *et al.* Crumbs binding to the Par-6 CRIB-PDZ module is regulated by Cdc42. *Biochemistry* [acs.biochem.5b01342](https://doi.org/10.1021/acs.biochem.5b01342) (2016). doi:10.1021/acs.biochem.5b01342
 34. Petit, C. M., Zhang, J., Sapienza, P. J., Fuentes, E. J. & Lee, A. L. Hidden dynamic allostery in a PDZ domain. *Proc Natl Acad Sci USA* **106**, 18249–18254 (2009).
 35. McLaughlin, R. N., Poelwijk, F. J., Raman, A., Gosal, W. S. & Ranganathan, R. The spatial architecture of protein function and adaptation. *Nature* **491**, 138–142 (2012).
 36. Niethammer, M. *et al.* CRIPT, a novel postsynaptic protein that binds to the third PDZ domain of PSD-95/SAP90. *Neuron* **20**, 693–707 (1998).
 37. Lee, D. D. & Seung, H. S. Learning the parts of objects by non-negative matrix factorization. *Nature* **401**, 788–791 (1999).
 38. Da Kuang, Yun, S. & Park, H. SymNMF: nonnegative low-rank approximation of a similarity matrix for graph clustering. *J Glob Optim* **62**, 545–574 (2013).
 39. Bahar, I., Atilgan, A. R. & Erman, B. Direct evaluation of thermal fluctuations in proteins using a single-parameter harmonic potential. *Fold Des* **2**, 173–181 (1997).
 40. Haliloglu, T., Bahar, I. & Erman, B. Gaussian Dynamics of Folded Proteins. *Phys. Rev. Lett.* **79**, 3090–3093 (1997).

41. Law, A. B., Fuentes, E. J. & Lee, A. L. Conservation of side-chain dynamics within a protein family. *J. Am. Chem. Soc.* **131**, 6322–6323 (2009).
42. Poole, A. Collective Structural Modes in Proteins: A Case Study in the PDZ Domain. (2012).
43. Fuentes, E. J., Der, C. J. & Lee, A. L. Ligand-dependent dynamics and intramolecular signaling in a PDZ domain. *J Mol Biol* **335**, 1105–1115 (2004).
44. Niu, X. *et al.* Interesting structural and dynamical behaviors exhibited by the AF-6 PDZ domain upon Bcr peptide binding. *Biochemistry* **46**, 15042–15053 (2007).
45. Shepherd, T. R. *et al.* The Tiam1 PDZ domain couples to Syndecan1 and promotes cell-matrix adhesion. *J Mol Biol* **398**, 730–746 (2010).
46. Whitney, D. S., Peterson, F. C., Kovrigin, E. L. & Volkman, B. F. Allosteric activation of the Par-6 PDZ via a partial unfolding transition. *J. Am. Chem. Soc.* **135**, 9377–9383 (2013).
47. Mishra, P. *et al.* Dynamic scaffolding in a G protein-coupled signaling system. *Cell* **131**, 80–92 (2007).
48. Skelton, N. J. *et al.* Origins of PDZ domain ligand specificity. Structure determination and mutagenesis of the Erbin PDZ domain. *J Biol Chem* **278**, 7645–7654 (2003).
49. Appleton, B. A. *et al.* Comparative structural analysis of the Erbin PDZ domain and the first PDZ domain of ZO-1. Insights into determinants of PDZ domain specificity. *J Biol Chem* **281**, 22312–22320 (2006).
50. Elkins, J. M. *et al.* Structure of PICK1 and other PDZ domains obtained with the help of self-binding C-terminal extensions. *Protein Sci* **16**, 683–694 (2007).
51. Kang, B. S., Cooper, D. R., Devedjiev, Y., Derewenda, U. & Derewenda, Z. S. Molecular roots of degenerate specificity in syntenin's PDZ2 domain: reassessment of the PDZ recognition paradigm. *Structure* **11**, 845–853 (2003).
52. Kang, B. S., Devedjiev, Y., Derewenda, U. & Derewenda, Z. S. The PDZ2 domain of syntenin at ultra-high resolution: bridging the gap between macromolecular and small molecule crystallography. *J Mol Biol* **338**, 483–493 (2004).
53. Kalinin, Y. *et al.* A new sample mounting technique for room-temperature macromolecular crystallography. *J Appl Crystallogr* **38**, 333–339 (2005).
54. Otwinowski, Z. & Minor, W. in *Macromolecular Crystallography Part A* **276**, 307–326 (Elsevier, 1997).
55. Southworth-Davies, R. J., Medina, M. A., Carmichael, I. & Garman, E. F. Observation of decreased radiation damage at higher dose rates in room temperature protein crystallography. *Structure* **15**, 1531–1541 (2007).
56. Holton, J. M. A beginner's guide to radiation damage. *Journal of synchrotron radiation* **16**, 133–142 (2009).
57. Rajendran, C., Dworkowski, F. S. N., Wang, M. & Schulze-Briese, C. Radiation damage in room-temperature data acquisition with the PILATUS 6M pixel detector. *Journal of synchrotron radiation* **18**, 318–328 (2011).
58. Diederichs, K. Some aspects of quantitative analysis and correction of radiation damage. *Acta Crystallogr D Biol Crystallogr* **62**, 96–101 (2005).
59. Adams, P. D. *et al.* PHENIX: a comprehensive Python-based system for macromolecular structure solution. *Acta Crystallogr D Biol Crystallogr* **66**, 213–221 (2010).
60. Joosten, R. P., Long, F., Murshudov, G. N. & Perrakis, A. The PDB_REDO server for macromolecular structure model optimization. *IUCrJ* **1**, 213–220 (2014).

61. Emsley, P., Lohkamp, B., Scott, W. G. & Cowtan, K. Features and development of Coot. *Acta Crystallogr D Biol Crystallogr* **66**, 486–501 (2010).
62. Shapiro, S. S. & Wilk, M. B. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* **52**, 591 (1965).
63. Jones, E., Oliphant, T., Peterson, P. others. SciPy: Open Source Scientific Tools for Python. (2001). at <<http://www.scipy.org/>>
64. Bakan, A., Meireles, L. M. & Bahar, I. ProDy: protein dynamics inferred from theory and experiments. *Bioinformatics* **27**, 1575–1577 (2011).
65. Lu, M. & Ma, J. Normal mode analysis with molecular geometry restraints: Bridging molecular mechanics and elastic models. *Arch. Biochem. Biophys.* **508**, 64–71 (2011).
66. Cock, P. J. A. *et al.* Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics* **25**, 1422–1423 (2009).
67. Stögbauer, H., Kraskov, A., Astakhov, S. A. & Grassberger, P. Least-dependent-component analysis based on mutual information. *Phys Rev E Stat Nonlin Soft Matter Phys* **70**, 066123 (2004).
68. van der Walt, S., Colbert, S. C. & Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* **13**, 22–30 (2011).
69. Boutsidis, C. & Gallopoulos, E. SVD based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition* **41**, 1350–1362 (2008).
70. Pedregosa, F. *et al.* Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research* **12**, 2825–2830 (2011).
71. Tsai, J., Taylor, R., Chothia, C. & Gerstein, M. The packing density in proteins: standard radii and volumes. *J Mol Biol* **290**, 253–266 (1999).

Chapter IV

Perturbing with electric fields: Direct observation of protein mechanics

1 Introduction

1.1 The function and evolution of proteins are deeply connected to mechanics

The fundamental biological properties of proteins—binding, catalysis, and allosteric communication—emerge from a global pattern of interactions between all constituent atoms. Often, this pattern is organized in the tertiary structure so as to produce the concerted motions of amino acid residues, defining transitions between a small number of functional states. This conformational cycling within proteins and protein complexes draws analogies to macroscopic machines¹ and lies at the heart of many biological processes: DNA replication², metabolism^{3,4}, transport⁵, cellular motility^{6,7}, and signal transduction⁸. Even when no conformational changes are apparent, different functional states of proteins can have a different pattern and extent of rigidity^{9,10}—entropic variations that obviously also contribute to their relative free energies¹¹ and therefore influence state transitions¹². Thus, the biology of proteins is deeply connected to their mechanics, that is, the motions a protein can perform and the forces constraining these motions. As in macroscopic machines¹, a comprehensive description of internal mechanics is the key in explaining how structure leads to function¹³. Unlike conventional machines, however, proteins are marginally stable evolved materials whose mechanics are governed by weak, heterogeneously cooperative interactions for which we yet have no good physical models.

1.2 Current biophysical methods cannot provide the data required to create adequate mechanical models of proteins

Current biophysical methods provide an incomplete experimental basis for making mechanical models of proteins. Nuclear magnetic resonance (NMR) spectroscopy¹⁴ and more recently, room-temperature crystallography¹⁵ provide information on both the structure and dynamics of local environments of atoms, and have been used to characterize weakly populated excited states of proteins^{3,14,16}. However, the complexity of disentangling conformational transitions occurring on multiple timescales, the difficulty of seeing collective motions in a comprehensive and model-independent manner, and the inability to generally relate the measured parameters to physical forces limit our understanding. Single molecule force spectroscopy provides a route to relate macroscopic conformational transitions to applied forces, but is limited in the kind of atomic detail required to define the underlying intramolecular mechanics. Time-resolved crystallography (TRX) offers, in principle, a direct route to observing concerted motions with high temporal and spatial resolution¹⁷. If a motion can be triggered within a protein crystal, one can directly obtain snapshots of electron density, reflecting atomic positions and fluctuations, as a function of time. Synchrotrons¹⁸ and X-ray free electron lasers¹⁹ provide the short X-ray pulses necessary to acquire such snapshots. However, existing TRX methods often rely on photoexcitation of chromophores as the means to induce motions²⁰. Even in the few proteins in which this is possible, such excitation acts at a fixed location, is not tunable, and deposits an amount of energy that likely far exceeds the energetic changes involved in most protein conformational changes.

1.3 EFX: a new method for revealing protein mechanics in atomic detail.

Here, we describe the development of a new method for studying protein mechanics and its application to a model system—a PDZ domain—in which prior work establishes both local and allosteric functional properties²¹⁻²³. The method, EFX, combines the use of strong electric field (EF) pulses to drive motions within protein crystals with simultaneous readout by picosecond-

scale X-ray pulses. EFX satisfies the three key characteristics required for a general mechanical analysis of proteins: (a) the application of forces of controlled magnitude, direction and duration, (b) the existence of defined, modifiable actuators (the charges) distributed throughout the protein structure on which these forces act, and (c) readout of conformational changes with high spatial and temporal resolution. We show that EFX can reveal protein motions associated with biological function and permit direct refinement of the atomic structures of low-lying excited states. This work initiates a path towards a full description of protein mechanics.

2 Results

2.1 Theoretical and practical considerations

The idea of EF-X is conceptually simple; many elementary charges and local dipoles are present in proteins (Figure 1 A), and with the application of sufficiently large external electric fields, it should be possible to exert forces on them that cause motions of atoms throughout the protein structure. If the electric field can be applied in conjunction with timed X-ray diffraction in protein crystals, it should be possible to experimentally observe all of these motions with high spatial and temporal detail (Figure 1 B). What are the practical considerations? First, whether an applied electric field can drive a conformational change or not depends on the degree to which that change corresponds to a displacement $\Delta\vec{x}$ of net charge q along the electric field, \vec{E} . For any residue (or other group of atoms), one can associate a transition dipole moment $\Delta\vec{\mu} = \sum_i q_i \Delta\vec{x}_i$ with each motion, where i is an index over atoms. Examples of transition dipole moments for various motions within protein structures are given in Table 1 (in units of elementary charge times distance (eÅ); $1 \text{ eÅ} \approx 4.8 \text{ D}$). Given $\Delta\vec{\mu}$, what sort of electric field strength do we need to induce motions? The energetic effect due to an external electric field equals $-\Delta\vec{\mu} \cdot \vec{E}$, and its significance

depends on how it compares to thermal energy $k_B T$; for example, a weakly populated excited state increases in occupancy by ~ 2.7 fold when its energy relative to the ground state is lowered by $1 k_B T$. Table 1 shows that electric field strengths of ~ 1 MV/cm should be in the right range to drive motions within protein structures by $1 k_B T$, that is, to induce subtle motions of the sort that can be observed through high-resolution diffraction methods. Fields of 1 MV/cm are dangerously large from a laboratory point of view, but are close to physiological; for example, 0.125 MV/cm corresponds to ~ 100 mV across a cell membrane. Such voltages influence conformational transitions in proteins such as ion channels⁵ and GPCRs²⁴, and are therefore in a range consistent with biological relevance.

From a practical point of view, there are several potential experimental complications. Of these, the main issue of concern is crystal heating. To explain, protein crystals typically have large solvent channels through which ionic currents will flow under applied electric fields, and by the Joule-Lenz law, will cause heating proportional to the square of the current. If sufficiently large, this effect can lead to dielectric breakdown, arcing, destruction of the crystal, and an abrupt end to the experiment. However, theoretical calculations with estimated conductivities of protein crystals²⁵ and typical composition of crystallization solutions strongly indicate that electric fields on the order of 1 MV/cm should be tolerated for pulse durations up to the low microsecond regime (Supplementary Information A). Together with minimum pulse-width limits of our current high-voltage system (~ 10 ns) and electrode design, this defines a window of timescales for these experiments at present (Figure 2 A). These limits can be extended through further technical development but provide a sufficient basis for initial investigation of EFX.

2.2 Molecular dynamics simulations suggest that external electric field will induce significant structural changes

To explore the question of whether or not field strengths of these magnitudes will induce motions in protein crystals, we developed a molecular dynamics framework using the AMBER12²⁶ package for studying the dynamical properties of protein molecules embedded in crystal lattices. This framework includes the implementation of a module for the application of anisotropic forces during simulation, which allows us to study the effect of crystal perturbation by an external electric field. We thus sought to identify the electric field magnitude needed to drive conformational changes in a lattice of 16 LNX2 PDZ2 molecules²⁷ (Figure 3 A).

We thus performed five molecular dynamics simulations of the lattice in the absence of electric field and in the presence of fields along the crystallographic x -axis with magnitudes of 10, 40, 60, and 80 mV/Å, on the order of what is experimentally feasible with our high-voltage power supply and pulse generator. While some systematic changes are qualitatively evident at higher field magnitudes (Figure 3 B), we sought a more quantitative approach to estimating the probability that conformational heterogeneity present towards the end of simulations was real and not simply an artifact of simulation. To that end, we used a t -statistic to compare structural displacements between symmetrically equivalent chains over the different simulations (Figure 3 C). This approach reveals that conformational changes induced are small but detectable at higher field strengths.

2.3 System design

Based on these considerations, we built a custom setup for room temperature X-ray diffraction of protein crystals under strong electric field pulses on the sub-microsecond timescale (Fig. 1 C–E, Figure 2). Protein crystals are mounted across the orifice of a glass capillary filled with crystallization solution and containing a metal wire as the ground electrode (Figure 1 C). The high-

voltage pulse is introduced from a top counter electrode filled with crystallization solution that makes a liquid contact with the crystal (Figure 1 D). A critical feature is to apply an electrically insulating glue around the crystal sides, attaching it to the bottom electrode (Figure 1 C–D); this forces the crystal to experience the bulk of the voltage drop, improves the spatial profile of the electric field, and ensures mechanical adhesion of the protein crystal to the capillary. At the experimental temperature (16°C), protein crystals are quite sensitive to humidity fluctuations, resulting in physical stresses on the crystal and loss of diffraction. Liquid contact of the crystallization solution with the protein crystal also addresses this problem. To prevent the liquid junction from drying out, we equipped the top electrode with a reservoir from which fresh solution could be provided by a computer-controlled gentle back-pressure (Figure 1 D). This electrode system was integrated into a synchrotron X-ray facility designed for time-resolved crystallography (BioCARS, Advanced Photon Source; Figure 1 E and Figure 2).

2.4 Application of EFX to a PDZ domain

As an initial model system, we chose the second PDZ domain of LNX2 (LN $X2^{PDZ2}$)^{27,28}, an E3 ubiquitin ligase (Figure 4 A). The PDZ domain family consists of 90–100 residue mixed α/β protein domains that generally bind the C-termini of target proteins between the α_2 helix and β_2 strand²⁹. A wide range of data demonstrate the existence and functional relevance of allosteric coupling of the ligand binding site to a few distant surfaces³⁰, especially the α_1 helix^{31,32} and the β_2 - β_3 loop³³ (see also Chapters 2 and 3). The sequence and structure of LN $X2^{PDZ2}$ indicate that it is a canonical PDZ domain, with no special features that compromise the generality of this study. Specifically, LN $X2^{PDZ2}$ has no known functional voltage dependence, providing an important test that EFX can be generically used in the context of randomly available formal and partial charges for analysis of protein mechanics.

We performed EFX experiments on LNX2^{PDZ2} with voltage pulses of +5–8 kV to 50–100 μm thick crystals, resulting in field strengths of $\sim 0.5\text{--}1$ MV/cm. The pulse durations ranged from 50–500 ns, and diffraction was collected with single 100 ps X-ray pulses, with a protocol that permits us to examine the atomic structure just before the electric pulse (voltage-OFF dataset) and at any specified time delay after initiation of the electric pulse (voltage-ON dataset) (Figure 4 B, C). The OFF dataset provides a reference structure for study of EF-induced effects. As predicted by our calculations, LNX2^{PDZ2} crystals (and other protein crystals; Table 2 and Figure 5) readily tolerated hundreds of 100–500 ns electric field pulses of ~ 1 MV/cm and X-ray pulses without substantial loss of diffraction. We collected a time series from a single LNX2^{PDZ2} crystal, consisting of an OFF data set and ON datasets at 6 kV, at 50, 100, and 200 ns delays from the rising edge of the electric field pulse (Figure 4 B, C; Table 3); variability in timing is less than 1 ns and are therefore negligible given the timescale of this experiment.

An important analytic tool comes from understanding how the electric field affects the symmetry of the crystal lattice. In general, the unit cell of a protein crystal can be constructed from a set of symmetry operations $\{S\}$ —combinations of translations and rotations—that define its so-called space group. For example, the LNX2^{PDZ2} crystals have C2 space group, which in addition to translational symmetry, has two kinds of rotational symmetry elements (Figure 4 D). As a consequence, there are four symmetric LNX2^{PDZ2} monomers per unit cell. What happens if an electric field is applied in a certain direction? Clearly, protein molecules in the crystal lattice with different orientations relative to the electric field will undergo different changes and will no longer be symmetric. The general rule is that all crystal symmetry operators S that do not preserve the orientation of the electric field will be violated (“broken”: $S \circ \vec{E} \neq \vec{E}$). For example, in the case of LNX2^{PDZ2}, the electric field is imposed along the crystallographic x -axis (a), perpendicular to all

rotation axes of the C2 space group (Figure 4 E); thus, all rotation operators are broken by the electric field. Now, the four LNX2^{PDZ2} molecules in the unit cell are no longer equivalent, but instead reduce to two molecules that see the EF in one direction (we will refer to these molecules as “*up*”), and two that see the EF in the opposite direction (the “*down*” molecules). In essence, if the *up* molecule experiences +6 kV, the *down* molecule experiences -6 kV, and so the force acting on otherwise equivalent atoms in these structures is opposite in direction. Though it need not be strictly symmetric, we would naïvely expect this to cause an opposite motion of atoms from their mean positions in the OFF state (Figure 6 A).

This breaking of symmetry provides a powerful way to study the effect of the electric field on the protein structure. We can compare the structures of the *up* and *down* molecules within the unit cell, an internally controlled experiment that isolates the effect of the electric field on atoms. In contrast, artifacts due to radiation damage and heating are insensitive to the direction of the electric field and cancel out in the comparative analysis (see Methods). In crystallographic terms, we compute an internal difference Fourier map in which we subtract the *down* electron density from the *up* electron density (Figure 6 A). This procedure is similar to the construction of anomalous difference maps, a routine process in crystallographic studies that also exploits a form of symmetry breaking. In the *up-down* difference electron density map, the hallmark of an EF-induced structural effect is to see peaks of opposite sign around the position of an atom in the voltage-OFF state (red and blue, Figure 6 A).

The *up-down* map shows pervasive evidence of subtle electric field-induced atomic motions (Figure 6 B–F). Just as proposed, we observe coordinated shifts of backbone, sidechain, and solvent atoms in opposite directions between the *up* and *down* molecules (Figure 6 C–F). Overall, the structural response to the EF is distributed broadly over the protein tertiary structure, in both

core and surface sites, with some of the strongest signals around the β_2 - β_3 , α_1 - β_4 , and α_2 - β_6 segments (Figure 6 B). To quantitatively examine the electric field-induced signal, we integrated the absolute difference electron density above a noise threshold (IADDAT) in a volume shell around the protein backbone³¹ (Figure 6 G). The *up-down* effect in the OFF state provides a measure of noise in the analysis (black trace, Figure 6 G). By comparison, we observe a robust signal in the presence of the EF that evolves over time from 50 ns–200 ns (blue, green, and red traces, Figure 6 G and Figure 6 H). Regions of significant EF-induced motions do not simply reflect solvent exposure or thermal (B) factors that are related to positional disorder (for all cases, $p > 0.1$, Fisher Z-test) (Figure 6 G). Many of the affected residues do not have formally charged side chains, indicating that they move due to local dipoles or due to structural coupling with other charged residues in the protein (Table 3). Note that the rate of evolution of the signal is heterogeneous over the structure (Figure 6 G, H); in some regions, it continues over the full time period (e.g. peaks “1” and “3”), whereas in other regions, it is complete at intermediate times (e.g. peak “2”). In addition, we find good correlation of the signal with independent processing of portions of the data (Figure 7 A) and between the different time points, with a clear increase in amplitude over time (Figure 7 B, C).

In summary, these data demonstrate the ability to stimulate and record subtle atomic motions throughout a protein structure with the application of ~ 1 MV/cm EF pulses and time-resolved crystallography. An analysis of crystal growth conditions, diffraction quality, and symmetry from the Protein Data Bank database suggests that many protein crystals should be amenable to the EFX experiment, including use of the *up-down* difference method due to symmetry breaking.

Direct modeling of electric field-induced excited states

We refined atomic structures of the *up* and *down* states of the LNX2^{PDZ2} domain at 200 ns from the onset of the electric field. Since the field only subtly biases the ground state conformation, we used the method of extrapolated structure factors (ESF) to carry out the refinement^{34,35}. This technique serves as a lens for excited states, permitting visualization by computationally expanding their relative occupancy. The ground state (OFF model, Table 7) was used as a starting point for refinement in small steps, with progress well supported by *R* factors (overall decrease: $\Delta R_{\text{work}} = -6.96\%$, $\Delta R_{\text{free}} = -5.92\%$; Figure 8). Propagation of errors suggests that the ESF structures at 200 ns have an effective resolution of 2.3 Å (Table 7).

The structures demonstrate EF-induced perturbations of nearly every type of physical chemical interaction throughout the protein structure—induction of side chain rotamer flips (Figure 9 A), continuous displacement of backbone, side chains, and bound waters (Figure 9 B, Figure 10 B–E), propagated rotamer shifts suggesting collective motions over extended regions of the structure (Fig. 9 C), breaking and re-forming of hydrogen bonds (Figure 9 D), global motions of entire secondary structure elements (Figure 9 E), and complex coordinated changes in large regions (Figure 9 F). Figure 10 shows additional examples. In addition, we find that the electric field can bias the occupancy of pre-existing conformational states in the voltage-OFF state (Figure 11). For example, S410 (Figure 11 A), N415 (Figure 11 B), and D368 (Figure 11 C) all show partial occupancy of two different rotameric states in a high-resolution (1.1 Å, room temperature) OFF structure (Figure 11), but are differentially forced into one of these configurations depending on the direction of the applied electric field (middle and right panels). Thus, rather than induce non-physiological states, EFX appears to expose low-lying conformational states that are energetically near to the ground state.

These data strongly validate the broad goals of EFX—to globally perturb and record subtle motions in a mechanistically unbiased manner at atomic resolution. A key feature is the ability to *actively* populate and directly model the structures of low-lying excited states around the ground states of protein molecules, the configurations most likely to be relevant over the functional reaction coordinate. In addition, the fact that we can collect datasets at various time delays after the initiation of the electric field pulse means that, in principle, we can observe these motions as they happen in time and make experimental movies of the temporal evolution of protein motions²⁰.

2.5 The biological relevance of EFX-induced motions

What do the EF-induced motions tell us about the biology of the PDZ domain? Simple inspection shows that backbone motions accumulate in four parts of the protein—the α_1 helix and the β_1 - β_2 , β_2 - β_3 , and α_2 - β_6 segments—all known to be functionally coupled to ligand binding (Figure 2, Figure 12 A, B). The partially buried α_1 helix forms a central component of allosteric communication in PDZ domains (e.g., in Par-6³², PDZ2³⁶ and PDZ5³⁷ of GRIP-1, PDZ6 of GRIP-1³⁸, and PDZ2 of PTP-BL³⁹). Consistent with this, residues in this helix undergo subtle EF-induced shifts roughly parallel to the helical axis (Figure 12 A), moving away from the ligand binding site in going from the *up* to *down* state. A number of residues leading into and out of the α_1 helix also display a pattern of rotameric transitions (largely without any associated backbone motion) even though many carry no obvious net charge (e.g. α_1 - β_4 region, Figure 9 F). The α_1 - β_4 region represents the allosteric surface at which binding of other proteins regulates PDZ ligand affinity^{32,39}. The β_1 - β_2 and β_2 - β_3 segments move and β_2 - β_3 becomes more ordered (Figure 10), transitions reminiscent of ligand-induced conformational changes PTP-1E^{PDZ230} and PSD95^{PDZ340}.

Finally, the α_2 - β_6 region, which displays slow, ligand-dependent dynamics and conformational change in several PDZ domains, shows an EF-induced displacement that is apparently coupled to a rotameric flip of the S410 side chain as it switches between two hydrogen-bonding networks (Figure 9 A). Interestingly, this switch positions the terminal amine group of K344, a conserved buried cationic residue, to make a hydrogen bond with an ordered water molecule near the ligand C-terminal carboxylate group (Figure 10 E). This configuration is similar to that in the ligand-bound state of several PDZ homologs (e.g. syntenin-1⁴¹, PDB: 1OBY).

To more rigorously test the relationship of EF-induced motions to PDZ function, we analyzed the pattern of systematic displacements of main-chain atoms between apo and liganded structures in 11 diverse homologs of the PDZ family (Figure 12 C, E). This study reveals functional motions that are conserved in the PDZ family. Ligand-induced motions shared by these homologs are most pronounced in the β_1 - β_2 , β_2 - β_3 , and α_2 - β_6 segments, and in both the α_1 and α_2 helices (Figure 12 C), comprising most regions with induced motions in the EF-X experiment. These regions are also linked by the protein sector^{31,42,43}—a group of amino acid positions that statistically co-evolves in the entire PDZ protein family—suggesting that the pattern of ligand-induced motions is an evolutionarily conserved feature of the PDZ family (Figure 12 D). Statistical comparison of the pattern of conserved apo to liganded motions (Figure 12 B) with the EF-induced up to down motions (Figure 12 E) shows strong overall correlation ($p < 0.001$, Fisher Z-test). This result is particularly interesting because in principle, ligand binding, and electric fields could impose forces in a protein structure in a manner completely distinct from each other. Thus, EFX samples motions in the protein structure that significantly reflect its biologically relevant mechanical modes.

2.6 From structure to mechanics

A central missing tool in our study of proteins is a method to stimulate and record biologically relevant motions over a broad range of timescales and with atomic resolution. Here, we show that strong but physiological electric fields can be used to examine a wide range of functional conformational changes within proteins—backbone shifts, side chain shifts and rotameric changes, changes in dynamic (dis)order, changes in hydrogen bonding, and solvent rearrangement. Since the model system studied here represents a typical protein with no unusual features (e.g. large dipoles, specific patterns of charged residues, or relevant voltage dependence) we expect that this method, with further development, can be broadly used to investigate the structural basis of protein function. Supporting this, we find that several protein crystals are robust to EF pulses in the range demonstrated here, and that the analytic methods enabled by crystal symmetry breaking are likely to be a general feature of this experiment. In future work, it will be of great interest to extend EFX to broader timescales of motions, to characterize motions in proteins with complex multistate conformational changes, and to study the structures of membrane proteins under physiological electric fields.

However, to go beyond the descriptive level of motions to the underlying physics, it is necessary to infer the spatial distribution of forces, and energies, associated with the observed conformational transitions. In this regard, it is informative to compare EF-X to single-molecule force spectroscopy (FS), or colloquially, "pulling" experiments⁴⁴. An electric field of 1 MV/cm (or 10^8 N/C) exerts 16 pN per elementary charge, a force sufficient to unzip a leucine zipper protein⁴⁵. Thus, an exciting prospect is to obtain direct force and free energy estimates for both gradual and discrete conformational changes as in force spectroscopy, but with the atomistic detail and temporal resolution made possible by EFX. This goal is complicated by the cooperative action of amino acids, but EFX provides a potential path to address this problem as well. We can collect

EFX data while varying the duration, orientation, spatial pattern, and magnitude of applied forces and statistically group residues that move together into collective modes⁴⁶. These modes may represent the basic mechanical units underlying protein function, providing a basis for a low-dimensional mapping of the relevant dynamical features of proteins.

This initial report of EFX does not yet comprise a simple, turnkey method. Crystal handling, electrode design, data analysis, and structure refinement all leave substantial room for improvement. However, this work provides an experimental basis for building physical models for protein mechanics, the critical link between structure and function.

3 Methods

3.1 Molecular dynamics simulation protocol

The AMBER FF99SB force field^{47,48} was used to model the structures and energies of protein models derived from PDB entries, and the general Amber force field (GAFF)⁴⁹ was applied in cases requiring modeling of organic solvents. For organic solvents, *ab initio* calculation at B3LYP/6-31*G//HF/6-31G* was performed using the Gaussian 03 software package (Gaussian, Inc., Wallingford, CT). Restrained electrostatic potential charges were derived using the RESP program⁵⁰ in AMBER12²⁶ taking the *ab initio* ESP at HF/6-41G* as input. Residue topologies were generated using the Antechamber module of AMBER12.

A model system for crystal simulations is derived from the unit cell reported for a given PDB entry, which includes protein molecules, organic solvents, ions, and water. A “supercell” composed of a number of repeating unit cells was constructed from the PDB coordinates of LNX2 PDZ2 (PDB ID 2VWR) using the *supercell* command from the *psico* Python module. The *AddToBox* command in AMBER12 was used to add solvents and ions to the simulation box.

MD simulations were performed with periodic boundary conditions to produce isothermal-isobaric ensembles at experimental temperature or 298.15 K using the PMEMD component of AMBER12, modified with the additional option to apply an external anisotropic perturbation—an external electric field (EEF) with units of mV/Å—along an arbitrary direction relative to the supercell. The Particle Mesh Ewald (PME) method was used to calculate the full electrostatic energy of a unit cell in a macroscopic lattice of repeating images^{51,52}. The integration of the equations of motion was conducted at a time step of 2 fs in the production phase. Covalent bonds involving hydrogen atoms were frozen with the SHAKE algorithm⁵³. Temperature was regulated

using the Langevin dynamics with a collision frequency of 5 ps^{-154} . Pressure regulation was achieved with isotropic position scaling and the pressure relaxation time was set to 1 ps.

Simulations were performed over three phases: (1) a pre-equilibration phase, in which the number of water molecules needed to maintain the volume of the MD box to within 0.5% of experimental conditions is determined. If inadequate, rewrite topology and restart; (2) an equilibration phase, in which the temperature of the system is gradually raised such that it relaxes without experiencing dramatic changes in force; and (3) a sample phase, in which the system is simulated under NTP conditions for at least 100 ns (with snapshots every 10 ps) with or without EEF.

Typically, for a given target, simulations with no field. After an equilibration period, multiple simulations were then initialized at multiple field strengths up to around $100 \text{ mV}/\text{\AA}$. Analysis of trajectories generated from these simulations was performed using elements from the the MDAnalysis⁵⁵ and Biopython⁵⁶ Python modules and NumPy⁵⁷. Internal distance matrices \mathbf{D} were calculated for all C_β atoms (for glycines, C_α atoms) of each chain χ at each snapshot t of trajectory with applied EEF of magnitude φ . To calculate \mathbf{D} for a pair of atoms i and j , each with coordinates x, y, z ,

$$\mathbf{D}_{\chi,i,j}^{\varphi,\tau} = \sqrt{\left(x_{\chi,i}^{\varphi,\tau} - x_{\chi,j}^{\varphi,\tau}\right)^2 + \left(y_{\chi,i}^{\varphi,\tau} - y_{\chi,j}^{\varphi,\tau}\right)^2 + \left(z_{\chi,i}^{\varphi,\tau} - z_{\chi,j}^{\varphi,\tau}\right)^2}.$$

The effect of EEF on a given chain varies based on the relative orientation of the protein in the field, so not all $\mathbf{D}_{\chi,i,j}^{\varphi,\tau}$ are necessarily equivalent. In a lattice with chains which are not translationally symmetric, each chain will be a member of a symmetry class X .

Next, the significance of motions over simulations in the presence of an EEF (e.g., with $\varphi = 100 \text{ mV/\AA}$) was assessed relative to the simulations performed in the absence of field. First, $\mathbf{D}^{0,\tau}$, which captures the conformational fluctuations in the absence of the electric field, is analyzed to give a lower limit on the types of conformational changes that can be expected during simulation. To this end, we split $\mathbf{D}^{0,\tau}$ into two halves, $\mathbf{D}^{0,a}$ and $\mathbf{D}^{0,b}$, where a and b refer to all snapshots in the first or second half of the trajectory, respectively. To maximize the statistical power of the analysis, we put all distances on the same scale and calculated a t -statistic matrix, \mathbf{t}_X^0 , where

$$\mathbf{t}_{X,i,j}^0 = \frac{\sum_{\chi} |\langle \mathbf{D}_{\chi,i,j}^{0,a} \rangle - \langle \mathbf{D}_{\chi,i,j}^{0,b} \rangle|}{\sqrt{\sum_{\chi} \text{var}(\mathbf{D}_{\chi,i,j}^{0,a}) + \text{var}(\mathbf{D}_{\chi,i,j}^{0,b})}}.$$

Now, a second t -statistic matrix, \mathbf{t}_X^{100} , can be calculated by comparing $\mathbf{D}^{100,\tau}$ to $\mathbf{D}^{0,\tau}$, where

$$\mathbf{t}_{X,i,j}^{100} = \frac{\sum_{\chi} |\langle \mathbf{D}_{\chi,i,j}^{0,b} \rangle - \langle \mathbf{D}_{\chi,i,j}^{100,b} \rangle|}{\sqrt{\sum_{\chi} \text{var}(\mathbf{D}_{\chi,i,j}^{0,b}) + \text{var}(\mathbf{D}_{\chi,i,j}^{100,b})}}.$$

The empirical cumulative distribution functions (ECDFs) of the upper triangle of each matrix can then be calculated and compared, yielding a quantitative means of assessing the false discovery rate for conformational change over the course of simulation.

3.2 Electronics

We designed a custom experimental set up in which a +/- 10 kV power supply (Spellman HV) charges a pulse generator (IXYS Colorado), from which high-voltage (HV) pulses can be triggered by a TTL signal (e.g. at BioCARS, from the beamline field-programmable gate array, or FPGA, panel). The pulse generator provides square pulses in so-called "half-bridge" configuration: the voltage is raised by connecting a charged HV capacitor to output, and is drained by connecting to

another large capacitor connected to ground. This prevents charge stored by the HV cables from draining through the protein crystal. Integrity of the conductive path to the tip of the capillary and its associated propagation delay were determined using an HV probe prior to experiments.

3.3 Safety considerations

Custom high-voltage cables (Gater Industries) were "hi-pot" tested for leakage current by the power group staff at the Advanced Photon Source. Cables were approved for use up to 8 kV (DC). The counter electrode was designed to avoid any path through air of less than 1 cm to the grounded cable connector exterior. The inhibit feature of the power supply was connected to the laser interlock system of the beamline hutch, ensuring that high voltage would be turned off on personnel entry into the beamline hutch. The power supply voltage and the counter electrode backpressure were controlled from the control room, using a DAC signal and a network-connected microcontroller, respectively.

3.4 Electrode construction

An RG-11 cable, terminated on one side with an HV connector (LEMO) mating with the pulse generator and on the other side with an SHV connector, was connected through two adapters to the housing of the "counter electrode" (top electrode, see Extended Data Figure 1 for illustrations). This housing was prototyped in-house using a 3-D printer (Form Labs) and custom fabricated commercially (PolyJet technology, PartSnap, Irving, TX), and contained a cylindrical glass insert terminated with a silicone gasket. Through this housing runs a thin metal wire (75 μm , Cooner Wire) with a dielectric coating, except at the tip. The wire is guided to the crystal through a glass capillary (Drummond Scientific, "0.5 λ ", 140 μm orifice). The electrode housing is filled with crystallization solution to assure chemical and physical stability of the crystal. In addition, a small

port allows for injection of liquid and the application of air backpressure by a custom microcontroller-driven pump. Bottom electrodes were prepared starting from glass capillaries (Drummond Scientific, "0.25 λ ") with a ~ 100 μm orifice. Each capillary was cut in half and aminosilanized at the original tip surface to improve its adhesive ability. A 75 μm thick uncoated stainless steel wire (Cooner Wire) was threaded until just below this orifice. The capillary was inserted in a reusable goniometer base (Mitegen) and soaked, in inverted position, in the appropriate protein crystallization solution. See Figure 5 for more details.

3.5 Protein expression, purification, and crystallization

LN $X2^{\text{PDZ2}}$ was previously crystallized by the Structural Genomics Consortium (SGC) (PDB entry 2VWR). We obtained an expression strain, BL21(DE3)-R3-pRARE2, and plasmid (pNIC28 derivative) encoding LN $X2^{\text{PDZ2}}$ from the SGC (www.thesgc.org; construct LN $X2A$ -c033). Expression, purification, and crystallization followed the protocol for PDB entry 2VWR as available on the SGC website. The full LN $X2^{\text{PDZ2}}$ construct includes residues 336–424 from *Homo sapiens* LN $X2$, retaining the F338L mutation described by the SGC, an N-terminal cloning artifact (positions 334–335), and a C-terminal ligand motif Glu-Ile-Glu-Leu (positions 425–428). LN $X2^{\text{PDZ2}}$ protein was expressed as an N-terminal hexahistidine fusion in *E. coli* BL21(DE3)-R3-pRARE2 cells and was purified by nickel affinity chromatography (Ni-NTA agarose, Qiagen), cleavage of the TEV tag by 1 U ProTEV per 50 μg protein during dialysis into 50 mM HEPES pH 7.5, 500 mM NaCl, 5% glycerol, 0.5 mM TCEP, a final step of size exclusion chromatography. Protein was concentrated to 20 mg/mL. Two protocols yielded suitable crystals. In the first protocol, concentrated protein was dialyzed twice for 12 and 6 h, respectively, against 5% glycerol (350 μL into 2 L), after dilution to 3.5 mg/mL using dialysis buffer. Note that protein will slowly precipitate in this buffer at -20 $^{\circ}\text{C}$. PDZ2 was crystallized by the hanging drop vapor diffusion

method. The crystallization buffer consisted of 19% PEG-300, 48 mM citric acid, 35 mM NaH_2PO_4 and 5% glycerol. Drops were set up with 0.55 μL protein and 1 μL well solution and incubated at 20 °C. In the second protocol, concentrated protein was diluted to 3.5 mg/mL with 10% glycerol. PDZ2 was then crystallized by the hanging drop vapor diffusion method. The crystallization buffer consisted of 27-31% PEG-300, 43 mM citric acid and 35 mM NaH_2PO_4 . Drops were set up with 1 μL protein and 1 μL well solution and incubated at 20 °C.

3.6 Crystal mounting

Crystals were mounted across the orifice of the pre-soaked bottom electrode (see Electrode construction, above, and Figure 5), attached to a magnetic goniometer base. Manipulations were carried out within a custom-made humidity chamber under a stereomicroscope. Sylgard 184 (Dow-Corning) was prepared to just before full curing and then stored in 1 mL aliquots on ice. Sylgard was manually applied around the crystal using a thin fish wire, taking care to not overcoat the crystal. Immediately after mounting, a Mitegen polyester sleeve⁵⁸ containing 15 μL of 50/50 crystallization solution and water at one end, was slid around the goniometer base to provide appropriate vapor pressure for the crystal. In the beamline hutch, we placed the bottom electrode on the goniometer and approximately positioned the counter electrode above the bottom electrode using an XYZ translation stage (Thorlabs). At this point, the Mitegen sleeve was cut to allow close, camera-guided approach of the counter electrode until a liquid junction with the crystal was established.

3.7 Data collection and reduction

Data were collected at the 14-ID beamline at the BioCARS facility at the Advanced Photon Source, Argonne National Laboratory. The cryostream temperature was set to 289 K. Diffraction

was collected using the Rayonix MX340-HS detector, with undulators U23 at 10.74 mm and U27 at 15.85 mm, resulting in an approximate wavelength range of 1.02–1.16 Å. Slit settings were 200 µm horizontal by 70 µm vertical. For the presented data set, collection proceeded in four 180° passes with 4°, 4°, 2° and 1° steps, respectively, and with matching offsets to maximize coverage of reciprocal space. An overview of experimental parameters per data set is given in Table 2. Laue data were processed by using the programs Precognition and Epinorm (Zhong Ren; www.renzresearch.com), with concurrent processing of OFF and ON frames. The PDZ2 data were integrated to 1.8 Å (see Table 3) and merged in space group P1 using the C2 unit cell dimensions. The orientation of the imposed electric field within the crystal lattice frame of reference was established directly from indexed diffraction patterns.

High-resolution room-temperature data were collected at the Stanford Synchrotron Radiation Lightsource (SSRL), BL11-1. Crystals of LNX2^{PDZ2} were mounted and protected by polyester tubing (Mitegen). The cryostream temperature was set to 277 K. Diffraction was collected using the PILATUS 6M PAD detector at a wavelength of 0.9795 Å. The presented data were collected from a single crystal and indexed, integrated, scaled and merged in HKL2000⁵⁹ (HKL Research). The data did not exhibit signs of substantial radiation damage (HKL2000 radiation damage coefficients of 0.01 and 0.03 for the two data collection passes; values >0.1–0.15 indicate significant damage⁶⁰) or non-isomorphism (coefficient 0.001 in both passes).

3.8 Refinement (C2 OFF models)

To obtain a suitable reference model for phasing difference maps, and as a starting point for refinement against extrapolated structure factors (below), we refined the structure of LNX2^{PDZ2} in the absence of electric field (OFF). We proceeded in two steps: (1) refinement against a high-

resolution (1.1 Å) data set collected at SSRL BL11-1 at 277 K, followed by refinement of an OFF model against the data collected at BioCARS, APS at 289 K. Refinement against the high-resolution data set followed standard protocols. Initial phases were determined by molecular replacement using a cryogenic structure of LNX2^{PDZ2} (model PDB ID: 2VWR). After simulated annealing, a model was refined by alternating rounds of automated refinement in PHENIX⁶¹ and manual adjustments in Coot⁶². Alternate conformations were placed where supported by averaged kick and F_o-F_c maps. The final model had no Ramachandran outliers. From this model, we derived a model without alternate conformations by further refinement, again without Ramachandran outliers (Table 10).

Initial phases for the BioCARS OFF model were determined by direct placement of this high-resolution single-conformer model of LNX2^{PDZ2}. To account for small differences in unit cell dimensions, the position of the starting model was refined by rigid-body refinement in PHENIX. Positions of solvent molecules were refined in Coot. Well-supported alternate conformations were modeled in Coot, adding duplicate conformations at neighboring residues, followed by real-space refinement, to relieve backbone strain. Further, limited automated refinement was performed in PHENIX. Anisotropic displacement parameters were refined only for residues with substantial difference density at atomic positions. Table 9 presents data collection and refinement statistics. Final Ramachandran statistics were 99% favorable, 1% allowed. Note that for calculation of internal difference maps it is essential that the model used for phasing be refined in the space group of the unperturbed crystal lattice to guarantee that the exact position of symmetry elements will be maintained. We subsequently expanded the refined model to the reduced-symmetry space group's asymmetric unit using PDBSET (CCP4/6.4.0⁶³).

3.9 Internal difference maps

Difference map Fourier coefficients were calculated directly from .hkl files with merged structure factors obtained from Precognition, using custom MATLAB scripts performing the following operations: (1) match structure factors F_{hkl} and $F_{\bar{h}\bar{k}\bar{l}}$ and calculate differences $\Delta F_{hkl} = F_{hkl} - \gamma_{\bar{h}\bar{k}\bar{l}} F_{\bar{h}\bar{k}\bar{l}}$ with $\gamma_{\bar{h}\bar{k}\bar{l}}$ correction coefficients for absorption anisotropy derived from OFF data (below); (2) read in phases of the corresponding structure factors from the C2 OFF model expanded into C1 using PDBSET (CCP4/6.4.0); (3) calculate weights according to

$$w_{hkl} = \left(1 + \frac{\sigma^2(\Delta F)}{\langle \sigma^2(\Delta F) \rangle} + 0.05 \frac{|\Delta F|^2}{\langle |\Delta F|^2 \rangle} \right)^{-1}$$

following refs. ^{19,20} and modified as in ref. X with a term reducing the contribution of any single structure factor difference (the precise weight does not have a substantial effect, data not shown). Following Schmidt et al. ⁶⁴, we find that the appearance of our difference density maps is improved if structure factors corresponding to large lattice spacings are rejected (here $d_{hkl} > 4 \text{ \AA}$), since structural changes tend to result in small-scale electron density differences.

In the analysis of "ordinary" difference amplitudes, e.g. $F_{hkl}^{\text{ON}} - F_{hkl}^{\text{OFF}}$ such as in the construction of extrapolated structure factors and ordinary difference maps, we can expect the effects of anisotropic absorption to cancel out. Internal difference maps are affected, however, as F_{hkl} and $F_{\bar{h}\bar{k}\bar{l}}$ scatter in different directions and will be absorbed to a different extent. To correct for anisotropic absorption, we used local scaling⁶⁵ as implemented in SOLVE, because Precognition does not efficiently capture the type of absorption anisotropy observed. To calculate correction coefficients $\gamma_{\bar{h}\bar{k}\bar{l}}$, we treat the $F_{\bar{h}\bar{k}\bar{l}}^{\text{OFF}}$ as the derivative data set, and F_{hkl}^{OFF} as the native data set (note that these are strictly equivalent in the C2 space group of the unperturbed crystal) and use default settings in SOLVE. Now, $\gamma_{\bar{h}\bar{k}\bar{l}} = \tilde{F}_{\bar{h}\bar{k}\bar{l}}^{\text{OFF}} / F_{\bar{h}\bar{k}\bar{l}}^{\text{OFF}}$, with the tilde indicating the local-scaled value.

The constructed difference map Fourier coefficients were converted to MTZ format using F2MTZ (CCP4/6.4.0) and map coefficients were calculated in Phenix (FFT) with a grid spacing of $d = 0.3 \text{ \AA}$, or half the default grid spacing. Absolute difference density around the protein was integrated in UCSF Chimera⁶⁶. Maps were manipulated using standard volume operations ("vop"). Absolute difference density was calculated and masked based on the C2 OFF model which had been expanded to C1.

3.10 Refinement against extrapolated structure factors

Since the electric field breaks C2 symmetry, refinement of its effects was done in the P1 space group. The C2 OFF model was used as a starting model for refinement. Because residual C2 translational symmetry (yielding C1 symmetry) might interfere with refinement, a P1 unit cell was chosen containing only one *up* and one *down* chain, indicating their orientations relative to the electric field direction. This unit cell requires a rotation around the c^* axis by $\arctan(b/a)$ (here: 31.1°). To this end, the C2 OFF model was "expanded" in PDBSET (CCP4/4.6.0) using symmetry operations (1) X, Y, Z, and (2) (1-X), Y, (1-Z). The resulting model was rotated in PDBSET by the indicated angle. A new SCALE record was subsequently generated, also using PDBSET. Note that anisotropic thermal ellipsoids are rotated by PDBSET, but this rotation appears to be slightly inaccurate.

Extrapolated structure factors were calculated as $F^{ESF} = N(F_o^{\text{ON}} - F_o^{\text{OFF}}) + F_o^{\text{OFF}}$, with $N = 1/(1 - f)$, the extrapolation factor³⁴. N enhances deviations between ON and OFF structure factors and is traditionally interpreted as the fraction of molecules excited by an optical pulse. Here, N simply increases the effective occupancy of excited states, facilitating structure refinement. The value of N was chosen as a trade-off between two criteria: map quality, which

deteriorates with increasing N , and the appearance of difference electron density peaks consistent with internal difference electron density maps, which initially increases with increasing N (systematic optimization of N in a site-specific manner will be explored in future work). Refinement, starting from the OFF model, was performed mostly manually in Coot, with determination of R factors in PHENIX, combined with bulk solvent scaling and occupancy refinement, after every 5–10 modifications. Progress of the refinement is shown in Figure 7. Towards the end of refinement, substantial geometric deviations from ideality accumulated, likely because refinement in Coot sometimes leads to nonphysical bonds and angles at the edges of refined regions. To this end, a few rounds of mild overall coordinate refinement (PHENIX, 10-15 microcycles, small geometric weights) were included, as marked by asterisks in Figure 7 A. Anisotropic displacement parameters originating from the OFF model were retained throughout and not subjected to refinement, except for a few atoms (such as at the side chain of T359 and C_α of L395) with clear difference signal and for a few added solvent molecules (isotropic B -factors only). Final Ramachandran statistics were 97% favorable/1.2% allowed, 1.8% unfavorable. Electron density maps and composite omit maps were calculated in PHENIX with 0.3 Å grid spacing and default settings. Reflections in the R_{free} test set were included in final map calculations.

3.11 Comparison to homologous PDZ domains

Thirteen pairs of high-resolution (≤ 2 Å) X-ray structures were chosen for PDZ domains with and without ligand (12 from the PDB, with mutants and heteromeric structures were excluded: NHERF-1^{PDZ1}: 1G9O, 1GQ4; PALS-1^{PDZ}: 4UU6, 4UU5; Tiam-1^{PDZ}: 3KZD, 4GVC; ZO-1^{PDZ1}: 4OEO, 4OEP; Erbin^{PDZ}: 2H3L, 1MFG; Dishevelled^{PDZ}: 2F0A, 1L6O; PDZK-1^{PDZ3}: 3R68, 3R69; Shank^{PDZ}: 1Q3O, 1Q3P; GRIP-1^{PDZ6}: 1N7E, 1N7F; PTP-1E^{PDZ2}: 3LNX, 3LNY; LNX1^{PDZ2}: 3VQF, 3VQG; Syntenin-1^{PDZ2}: 1R6J, 1OBX); augmented with a pair of structures for PSD-95^{PDZ3};

Ranganathan lab, unpublished results). All structures were aligned in PyMOL, using “super” for backbone atoms, first to the *down* state of LNX2^{PDZ2} at 200 ns, and then within each pair. For all backbone atoms with matching positions within LNX2^{PDZ2} residues 337-420, displacements (Δr) from unbound (apo) to bound (liganded) were then calculated. Overlap between Δr for different homologs was quantified by the cosine of their angle (e.g. 1 for same direction, 0 for orthogonal displacement, -1 for opposite direction). Atoms with $|\Delta r| < 0.1 \text{ \AA}$ were excluded from analysis. Since most displacements for LNX1^{PDZ2} fell below this cutoff, it was excluded from analysis. Syntenin-1^{PDZ2} was excluded from further analysis since its displacements bore no significant resemblance to those of the remaining eleven PDZ domains. Average displacements in Figure 12 C represent the median magnitude of apo to liganded displacement over homologs and the average direction, that is, the average of unit vectors representing the direction of displacement for each homolog.

3.12 Statistics

Statistical significance of observed correlations was assessed in two steps. First, observed quantities were transformed to stabilize variance, reduce kurtosis, and approximate a normal distribution. Specifically, IADDAT values (Figure 6) were square-root transformed, and *B*-factors (Figures 6 and 12) and displacements (Figure 12) were log-transformed. To test for statistical significance of correlations, sample correlation coefficients for the data displayed in Figures 6 and 12 were transformed using the Fisher *Z*-transformation, and tested for deviations from zero assuming a normal distribution. For the reported statistical comparison of Figure 12, we assumed that individual residues can be considered independent (yielding $p < 0.001$). This is appropriate for well-determined data. As a conservative approach, one can take the shorter of the correlation length scales of *B*-factors and observed displacements (~2 residues) as a measure of data

dependence, instead, yielding a reduced number of independent samples and $p < 0.01$. Accessible surface area was quantified using ASAView based on the OFF model.

4 Figures

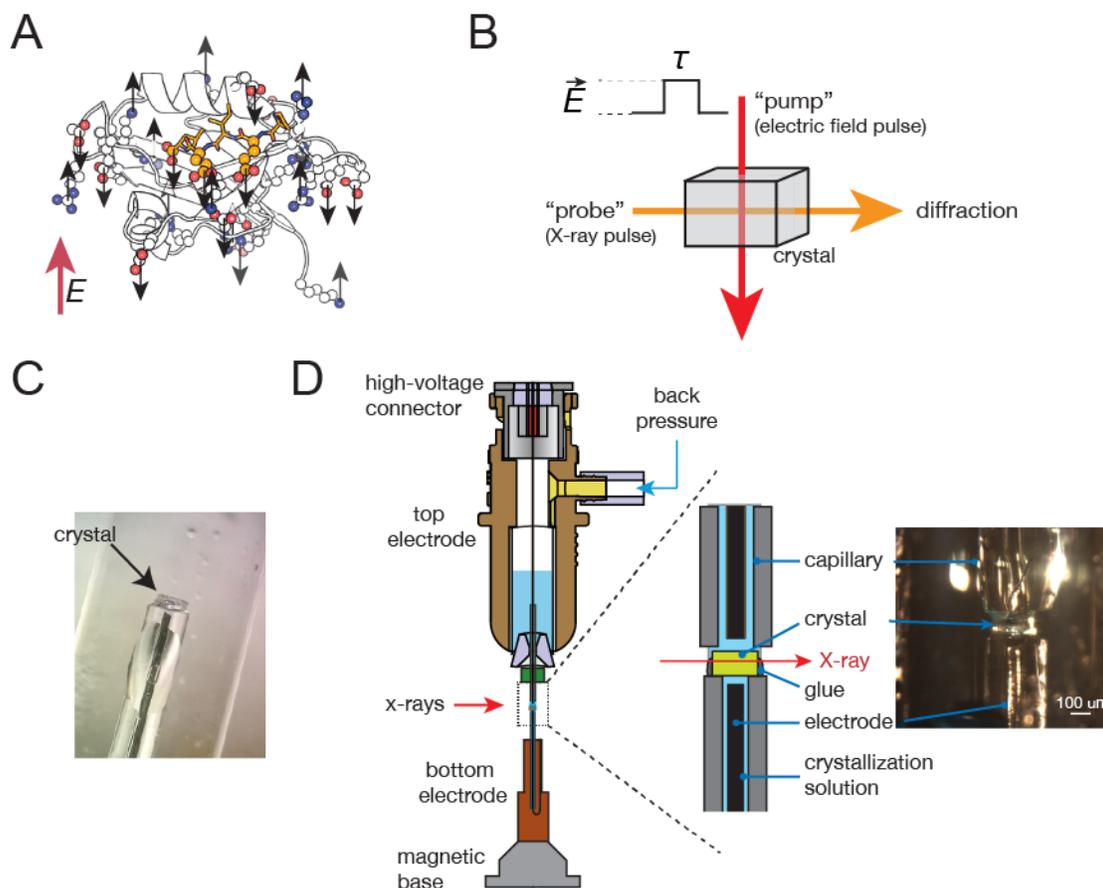


Figure 1. EFX, principles and implementation. **A.** A sampling of charged residues in the structure of the LNX2^{PDZ2} protein (PDB: 2VWR), illustrating the broad distribution of potential actuators for applied electric fields. Black arrows indicate forces imposed on the charges by the applied external electric field (in red). **B.** EFX involves stimulation of motions in protein crystals (the “pump”) by an applied electric field (\vec{E}) of duration τ , and readout by much faster X-ray pulses (the “probe”). **C.** A LNX2^{PDZ2} crystal mounted across the orifice of a glass capillary, filled with crystallization solution and a metal electrode. The crystal is sealed onto the capillary by electrically insulating glue. **D.** Schematic and detail of the experimental setup. The crystal is mounted on the bottom electrode and the high-voltage is delivered from a top counter electrode through a liquid junction comprised of crystallization solution. Slight back-pressure on a reservoir of crystallization solution keeps the crystal hydrated throughout the experiment. **E.** A view of the assembled experiment, showing the directions of the electric field and X-ray pulses.

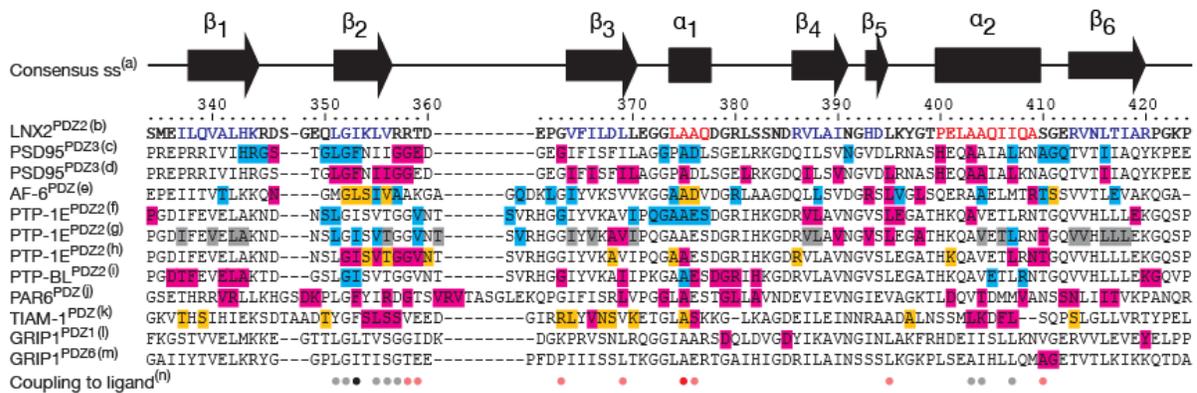


Figure 2. Multiple sequence alignment and mapping of literature results on allostery for PDZ domains. Select PDZ domains were aligned based on X-ray structures using Promals3D⁶⁷ with limited manual adjustment of gapped regions. (a) Consensus secondary structure: the ligand binds between the β_2 strand and the α_2 helix. (b) Sequence of LNX2^{PDZ2}. (c) Sequence of PSD95^{PDZ3} with clusters of residues for which backbone chemical shift in the ligand-bound state respond in a correlated manner to mutations; clusters shown in blue and magenta⁶⁸. (d) Residues in PSD95^{PDZ3} for which mutations on average have a significant effect on ligand affinity³³. (e) Residues in AF-6^{PDZ} with intrinsic millisecond dynamics affected by ligand-binding (blue), or exhibiting millisecond chemical exchange in the ligand-bound state only (magenta)⁶⁹. Residues experiencing a substantial change in chemical shift upon ligand binding in yellow⁶⁹. (f) Sequence of human PTP-1E^{PDZ2}, with colors indicating the two pathways of energetic correlations identified by molecular dynamics simulations⁷⁰ (blue: pathway I; magenta: pathway II); (g) Residues of PTP-1E^{PDZ2} displaying a change in dynamics on ligand binding³⁰, as determined from relaxation dispersion measurements on ¹³C-labeled methyl groups and backbone order parameters. Residues in blue and magenta are significantly affected; magenta residues are the "distal surfaces" identified; gray: no significant response; white residues not assessed. (h) Residues of PTP-1E^{PDZ2} with chemical shift change upon binding to Fas receptor C-terminal peptide⁷¹ indicated in yellow ($\Delta\delta > 0.2$), magenta ($\Delta\delta > 0.3$). (i) Residues of mouse PTP-BL^{PDZ2} showing slow chemical exchange (magenta) or large chemical shift change (blue) upon titration with its modulatory PDZ1 domain as published³⁹ and per pers. comm. with Dr. G. Vuister (f) Sequence of PAR6^{PDZ} with residues undergoing chemical exchange in response to positional exchange of L164 and K165, the conserved positively charged residue at the base of the ligand binding pocket (as published²³ and per pers. comm. with Dr. B.F. Volkman). (k) Significant Tiam-1^{PDZ} chemical shift changes upon binding of a peptide derived from Syndecan-1⁷² (l) Residues in GRIP1^{PDZ1} that form crucial interface contacts with its modulatory PDZ2 domain (only a few residues were tested based on crystallographic inspection)³⁶. (m) Region of GRIP-1^{PDZ6} shown to undergo a large conformational change in response to ligand binding, as identified by X-ray crystallography. (n) Aggregate evidence for conformational or energetic coupling to the ligand. Gray, black: residues in direct contact with the ligand in LNX2^{PDZ2} (PDB entry 2VWR; minimum distance between C, N, O, or S atoms $< 5 \text{ \AA}$); red, pink: allosteric positions. Lighter shades: supported by four or five studies; darker shades: at least six studies.

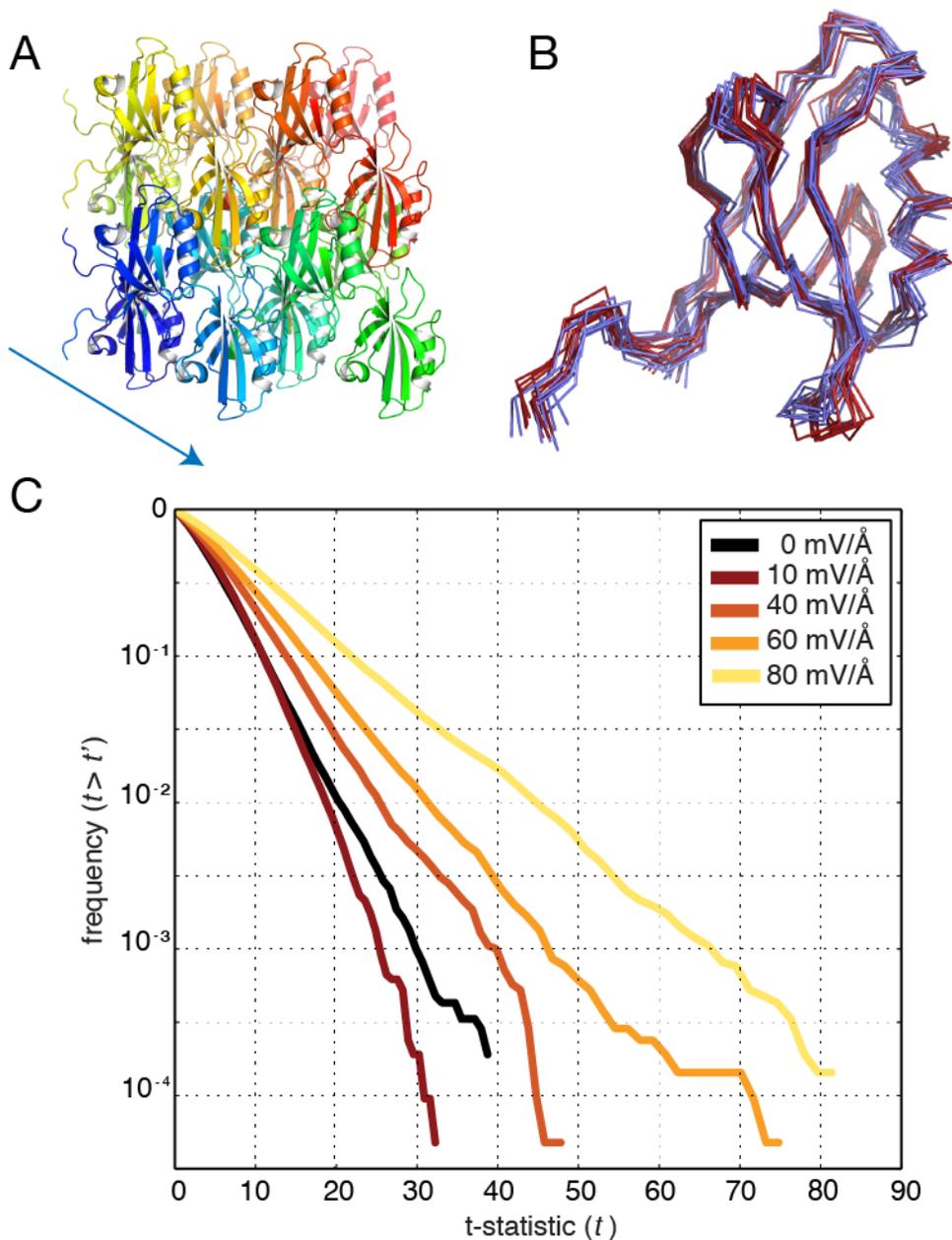


Figure 3. Comparison of molecular dynamics simulations in the absence and presence of an external electric field (EEF) reveal probability of observing changes in protein conformation. **A.** A section of lattice, or supercell, comprised of 16 LNX2^{PDZ2} molecules was created using *supercell* command from the *psico* Python module. The EEF was applied along the crystallographic *x*-axis as indicated by the blue arrow. **B.** A set of equivalent chains from the supercell as simulated in the presence of an 80 mV/Å field, superposed. Blue chains were taken from the first snapshot at 10 ps, while red chains were taken from a snapshot at 50 ns. Qualitatively, deflections can be seen in several areas, including the α_1 helix and the β_2 - β_3 loop. **C.** The *t*-statistics for coordinate shifts within translationally-equivalent chains from 1–50 ns and 50–100 ns of each trajectory without

applied EEF were compared to calculate a null distribution for motion. The empirical cumulative distribution function (ECDF) was then calculated for this distribution. ECDFs were then calculated from t -statistics over the 50–100 ns interval for each EEF strength relative to the same time window without EEF. For a t -statistic of 20, the overall discovery rate is approximately 0.01% for the no-EEF case and a little more than 0.1% with an 80 mV/Å EEF applied, yielding a false discovery rate of 10%. The observation that the 10 mV/Å ECDF yields a lower true-positive rate in general suggests that the field is so weak that conformational change cannot be ascribed to it.

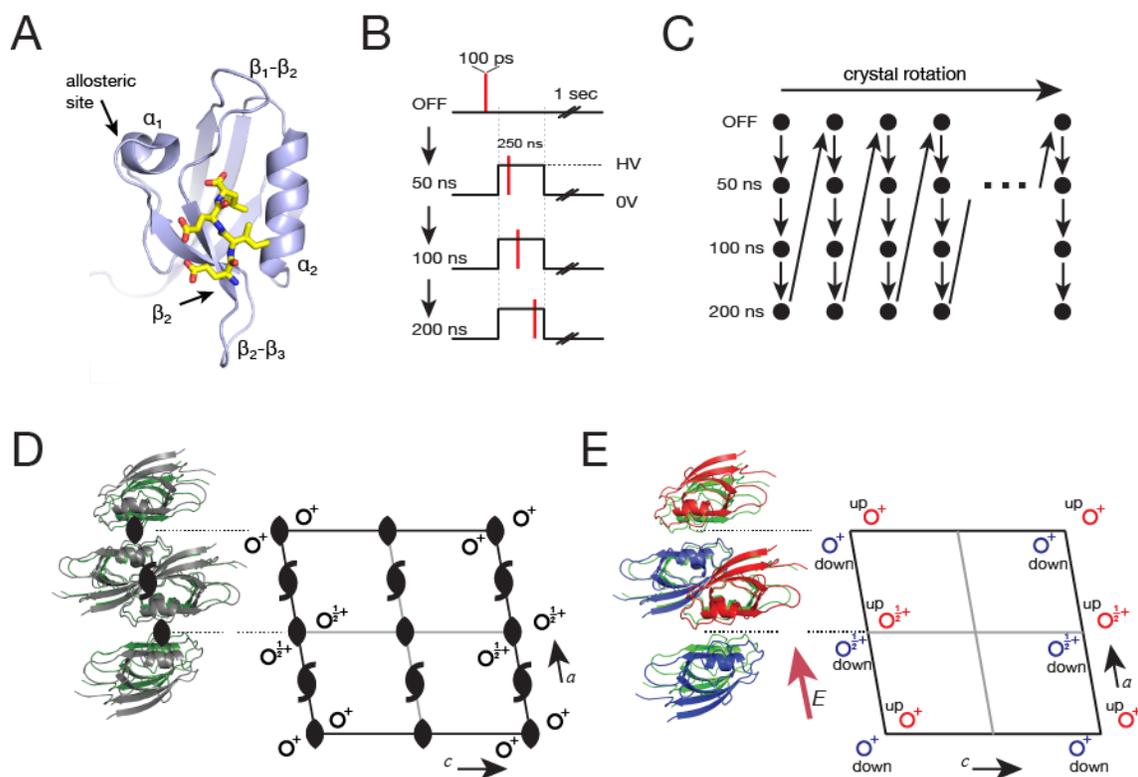


Figure 4. An EF-X experiment in the LNX2^{PDZ2} domain. **A.** As with all PDZ domains, LNX2^{PDZ2} is approximately a six-stranded α/β -sandwich with two helices; target ligands (in yellow stick bonds) bind in a groove between the β_2 and α_2 segments. The binding site is structurally, functionally, and evolutionarily coupled to allosteric sites on the β_2 - β_3 segment and the β_1 - β_4 segment (through the carboxylate binding loop (β_1 - β_2) and the α_1 helix). **B.** For each crystal orientation, the data collection protocol involves four sequential X-ray exposures: no-voltage (OFF), and three time delays (50, 100, 200 ns) following the onset of the voltage pulse; one second is allowed between pulses for crystal cooling. **C.** The protocol in **B** is repeated for a series of crystal rotations to collect a full diffraction dataset. Arrows indicate the sequence of events. **D.** LNX^{PDZ2} crystallizes in C2 space group, which includes two kinds of rotational symmetry; this results in four molecules per unit cell and one molecule per asymmetric unit. **E.** With the electric field \vec{E} (applied along a), all rotational symmetry is broken. This results in a new unit cell with two molecules per asymmetric unit (red and blue)—one experiencing $+\vec{E}$, and one experiencing $-\vec{E}$.

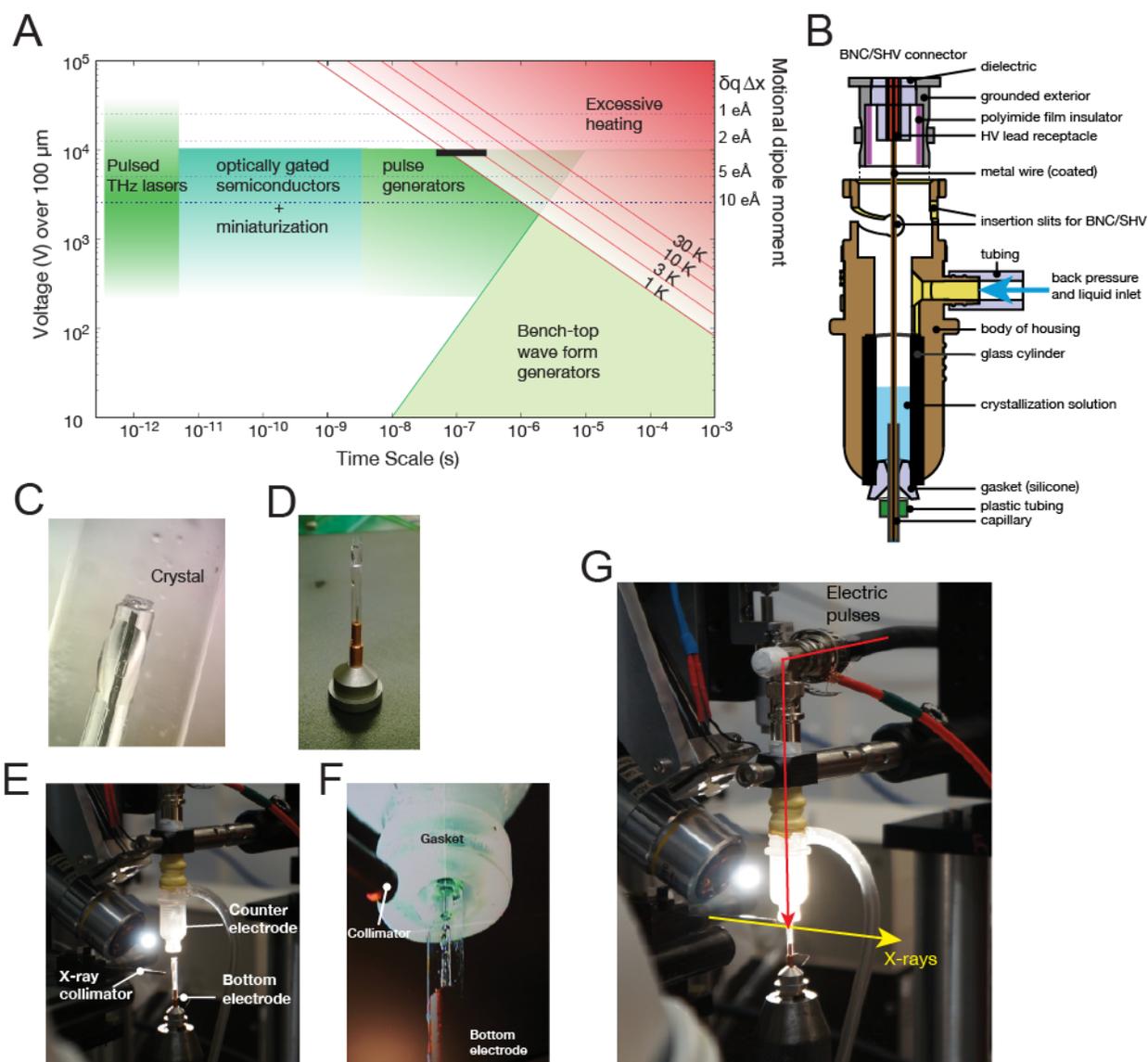


Figure 5. The experimental setup of EFX. **A.** A plot relating the applied voltage across a 100 μm thick crystal (left vertical axis) and the size of transition dipole moments of conformational changes that can be excited by 1 kT (right vertical axis) to the duration of the applied electric field. Feasible methods of generating strong electric field pulses are indicated as green and cyan shaded areas. Waveform and pulse generators can provide pulses down to the nanosecond time-scale. For faster pulses, terahertz pulses lasers with strong electric field components and visible/IR pulsed lasers can rapidly gate optical semiconductors; such systems are already present at third-generation synchrotron and X-ray free- electron laser facilities. The black bar indicates the approximate range covered by the current experiments. **B.** Schematic cross section of the counter electrode. The blue arrow indicates the potential to apply back pressure to drive flow through the capillary. **C.** Crystals are mounted on top of capillaries containing a metal electrode and soaked in crystallization solution. **D.** The capillary with crystal is mounted in a reusable goniometer base and protected

from humidity fluctuations with a polyester sleeve (Mitegen) containing 50% (v/v) crystallization solution. This assembly forms the bottom electrode. **E.** The counter electrode and bottom electrode are assembled at the beam line to allow rotation around the capillary axis. **F.** Once the sleeve is trimmed to just above the level of the crystal, the counter electrode is brought in using a translation stage (camera view of the approach). **G.** Overview of the final set up and with the direction of the X-ray and electric field pulses.

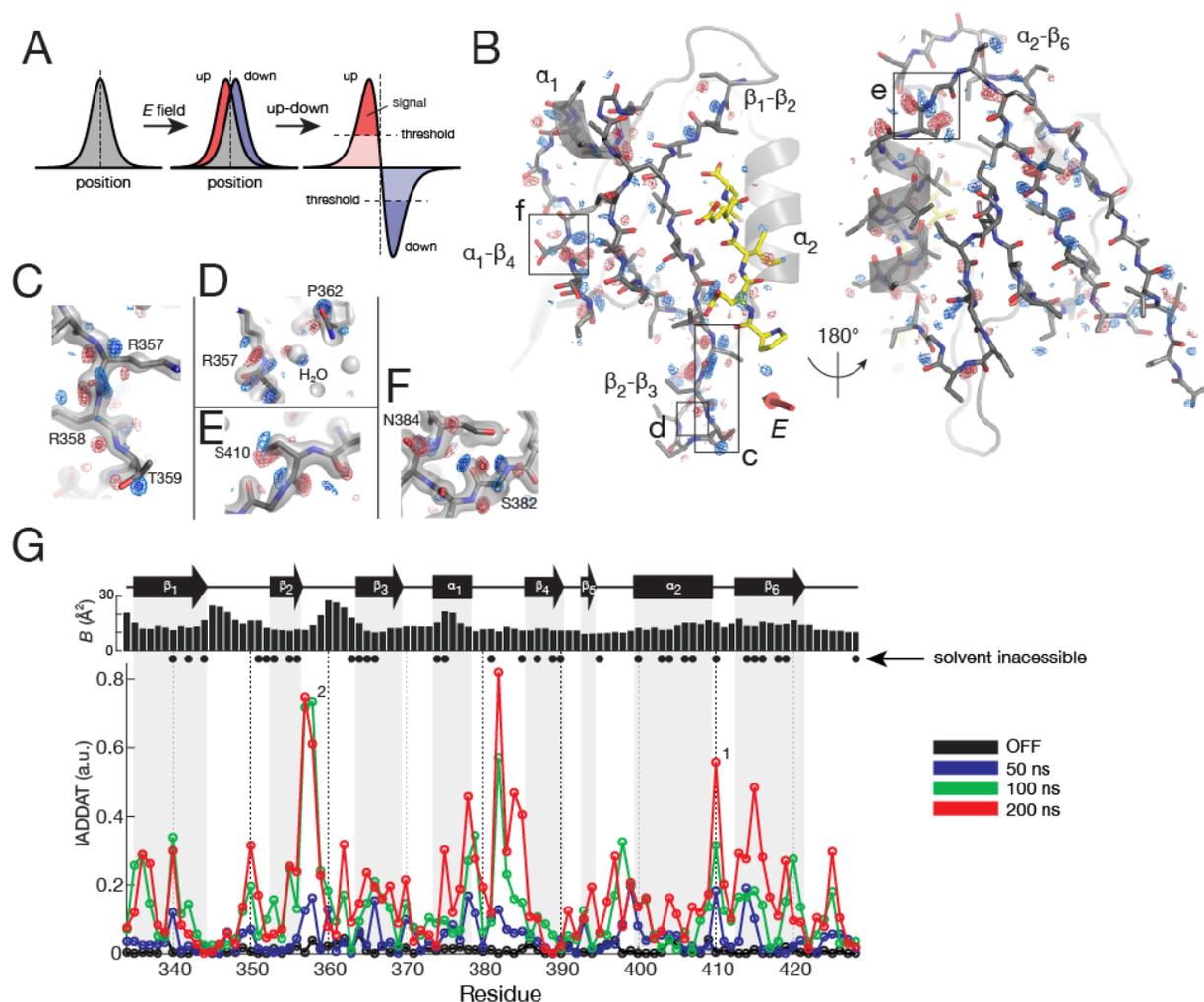


Figure 6. The *up-down* internal difference analysis. **A.** A schematic representation of the calculation. In the simplest case, application of the electric field should shift the electron density distribution for an atom in the *up* and *down* molecules (red and blue) in opposite directions around its centroid in the voltage-OFF molecule (gray) (left and middle panels). Subtracting the *up* and *down* densities and applying a noise threshold (right panel), we expect to observe peaks of positive and negative difference density (red and blue) surrounding an atom in the OFF state. This is the hallmark of an electric field-induced motion. **B.** A global view of the *up-down* internal difference map for LNX2^{PDZ2}, with regions highlighted in panels **c-f** boxed and labeled. The red 3D arrow indicates the direction of the electric field, and bound ligand is indicated in yellow stick bonds. Maps were contoured at $+3.5$ and $+4.5 \sigma_{\text{OFF}}$ (light and dark red, respectively) and -3.5 and $-4.5 \sigma_{\text{OFF}}$ (light, dark blue) and are displayed within a 1.8 \AA shell of atoms. **c-f.** The data show clear evidence of electric field-induced motions—opposing red and blue density—throughout the protein structure for main chain, side chain, and solvent atoms. **G.** A plot of integrated abolute difference electron density above threshold (IADDAT) for the *up-down* molecules in all datasets as a function of LNX2^{PDZ2} primary structure. The OFF difference density gives a measure of noise in the analysis, and the blue, green and red traces show the time evolution of electric field-induced effects. The signal evolves heterogeneously, with some positions (e.g. “1” and “3”) showing continuous motion throughout and other positions (e.g. “2”) showing completion of motion in the

time scale of the experiment. For comparison, the graphs above indicate buried residues (solvent inaccessibility < 0.15) and refined isotropic B -factor for the voltage OFF model.

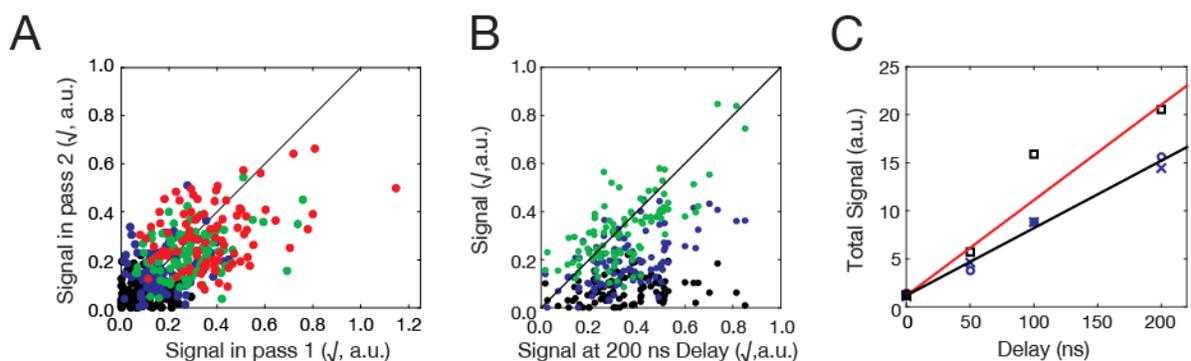


Figure 7. Internal consistency and temporal evolution of internal difference map signal. Analysis for the data presented in Figure 6. **A.** Consistency of estimated signal per residue derived from two data collection passes on the same crystal (black: OFF; blue: 50 ns; green: 100 ns; red: 200 ns). Overall correlation coefficient 0.59. Signal: integrated absolute difference density above $2.5 \sigma_{\text{OFF}}$ within 1.5 \AA of the protein backbone; square-root transformed to stabilize variance. Per-time-point correlation coefficients: -0.07 (OFF, $p > 0.1$), 0.23 (50 ns; $p = 0.01$); 0.35 (100 ns; $p < 10^{-3}$) and 0.34 (200 ns; $p < 10^{-3}$). **B.** Consistency of the obtained signal per residue between time points. Correlation coefficients: 0.17 (OFF; $p = 0.05$), 0.55 (50 ns; $p = 1 \times 10^{-9}$) and 0.72 (100 ns; $p < 10^{-20}$). The diagonal is shown for reference. Note that slight correlation in the OFF data set may indicate imperfect correction for anisotropic absorption. **C.** Signal integrated along the entire protein backbone in passes 1 and 2 (blue crosses and circles, respectively) and over the entire data set (squares). The red line indicates a naïve expectation of a $\sqrt{2}$ -fold increase in signal-to-noise ratio.

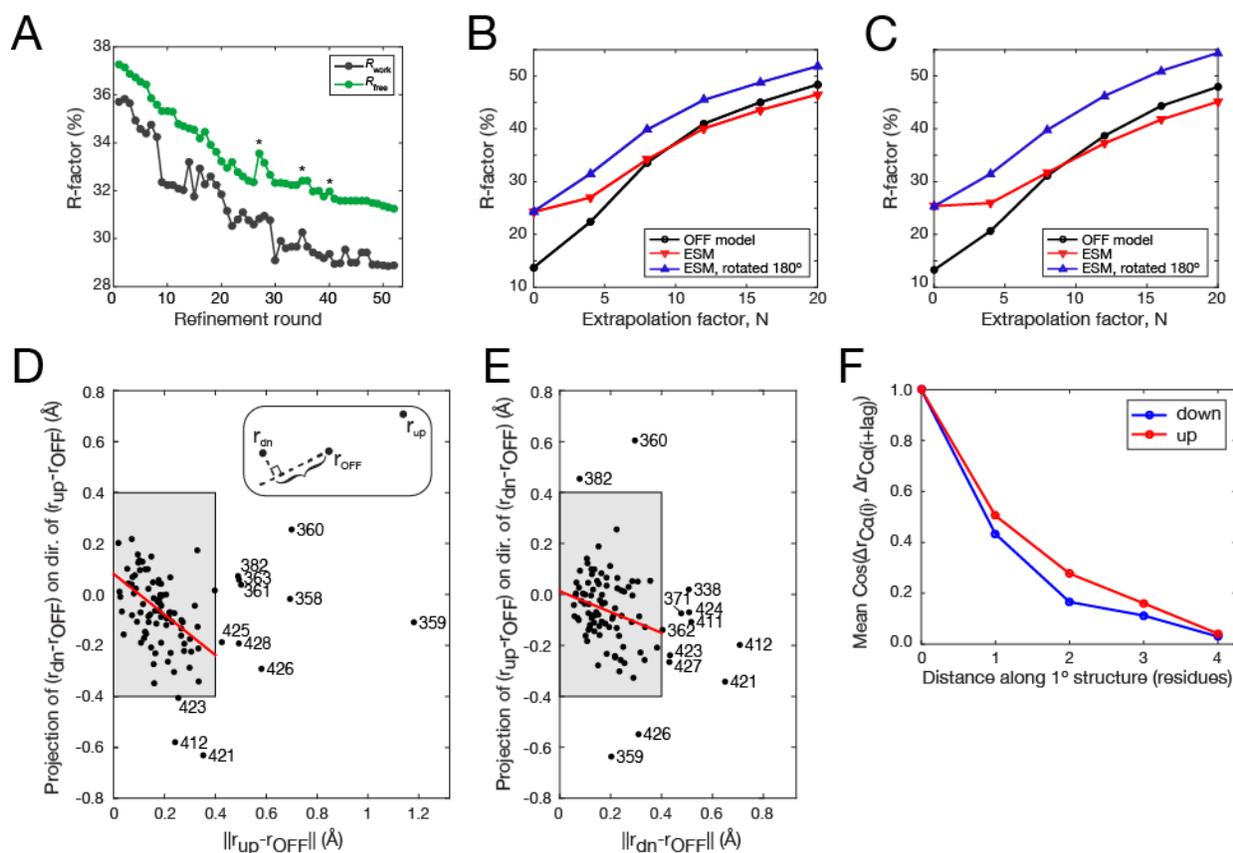


Figure 8. Refinement, voltage-ON model (200 ns). **A.** Progress of refinement against extrapolated structure factors. Rounds marked by asterisks involved automated refinement with mild stereochemistry constraints to reduce deviations from optimal geometry due to manual refinement in Coot. Fluctuations in R_{work} appear to be mostly due to the PHENIX bulk solvent scaling calculation used in R factor calculation. **B–C.** Negative correlation between movements in the *up* and *down* states: **B.** R factor for comparison of extrapolated structure factors, as a function of the degree of extrapolation, N , as derived from data set 2 (150 ns), against calculated structure factors (F_c) derived from (1) the OFF model (black), (2) the excited state model presented in the main text (red), and (3) an "upside-down" ESM obtained by 180° rotation around the C2 two-fold rotation axis (blue), all derived from data set 1. N relates to the fraction f of OFF signal subtracted as $N = 1/(1 - f)$. No refinement against data set 2 was performed except for bulk solvent scaling. No test set was assigned. **C.** For comparison, the same analysis as in panel **C**, comparing the OFF model and 200 ns ESM model to the 100 ns data as presented in the main text (that is, from the same crystal). **D–F.** Relation between C_α displacements in the *up* and *down* conformations. **d.** Projection of the *down* displacement on the direction of the *up* displacement, and **e** the *up* displacement on the *down* displacement direction (all displacements are relative to the OFF model; models were superimposed using "super" in PyMOL, based on C, C_α and N atoms of the protein backbone, including only residues 338–356, 362–380, 384–408 and 412–419; that is, excluding N- and C-terminal regions and mobile parts of the β_2 - β_3 , α_1 - β_4 and α_2 - β_6 loops). Shown are, for example $\Delta\vec{r}^{\text{up}} \cdot \Delta\vec{r}^{\text{down}} / \|\Delta\vec{r}^{\text{down}}\|$ versus $\|\Delta\vec{r}^{\text{down}}\|$, as illustrated in the inset. For small displacements, a simple inverse dependence is expected. This is tested by robust linear regression for (projected) displacements smaller than 0.4 Å (red line fits to data in gray boxes, using default settings in

MATLAB). **D.** slope = -0.80 ± 0.16 , intercept = $0.081 \pm 0.031 \text{ \AA}$; correlation coefficient: -0.44. **E.** slope = -0.41 ± 0.17 , intercept = $0.012 \pm 0.033 \text{ \AA}$; correlation coefficient: -0.27. **F.** Average cosine between displacements of nearby C α atoms as a function of distance along the primary structure.

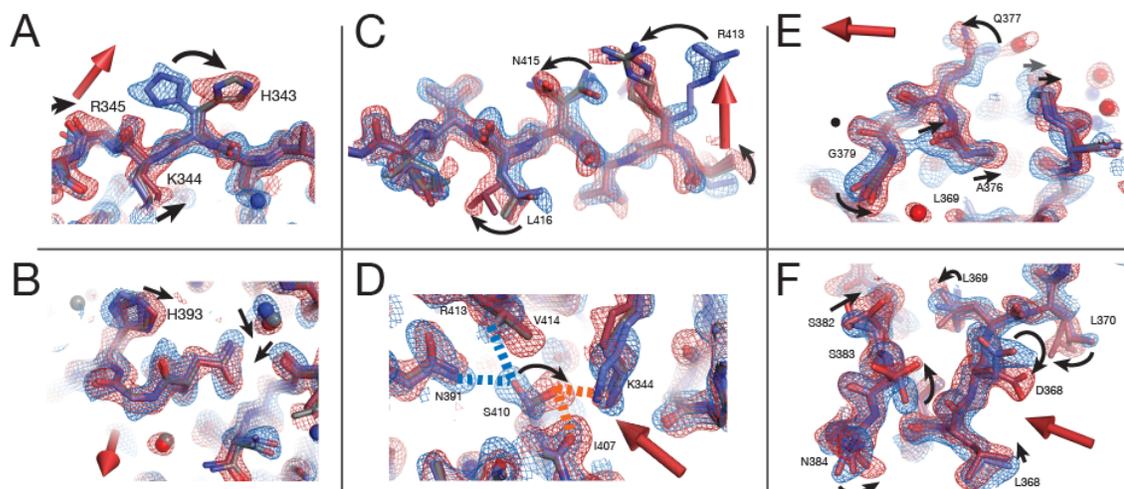


Figure 9. A gallery of electric field-induced structural effects. **A–F** show refined models and associated $2F_o - F_c$ electron density for the up (red) and down (blue) LNX2^{PDZ2} structures in the crystal unit cell; in each panel the direction of the electric field is indicated by the solid 3D arrow. The data show examples of rotamer flips (H343, **A**), continuous displacements (H393, **B**), sequential series of rotamer flips (R413, N415, L416, **C**), re-arrangements of hydrogen bonding networks (S410, **D**), systematic motions of secondary structure elements (the α_1 helix, **E**), and more complex combinations of these effects in large regions (**F**). The sign convention is such that a positive charge (and atoms coupled to it) in the *up* model would move in the direction of the field and in the *down* model with move against the field. These data show that motions occur both at solvent exposed (**A–C**, **E**) and buried (**D**, **F**) regions, and comprise modulation of a variety of different physical mechanisms.

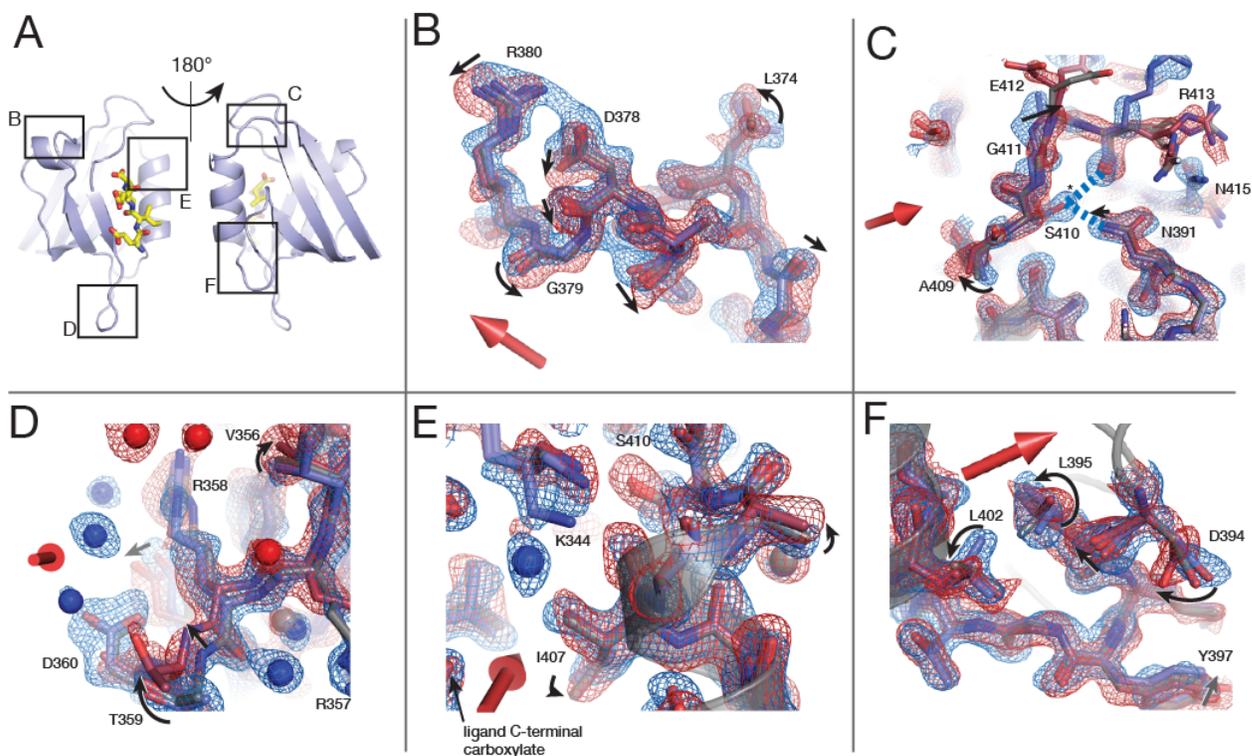


Figure 10. Additional views of conformational changes due to the electric field. **A.** Reference model indicating regions examined in panels **B–F**. **B–F.** Maps and models as in Figure 9 with motions indicated by arrows and residues coupled to ligand binding in PDZ domains shown in boldface. **B.** Top view of the α_1 helix, waters omitted and the side chain of Q377 truncated for clarity. **C.** Transverse shift of the α_2 - β_6 loop, and perturbed *down* state of S410, forming new hydrogen bonds to R413 and N391 (dashed blue lines). **D.** Upward motion of the β_2 - β_3 loop and change in dynamic disorder of protein and solvent. **E.** Conformational changes at the top of the ligand-binding pocket, with motion of the terminal amine of the K344 towards the ligand carboxylate group in the *down* state. **F.** Coupled rotameric changes of L402 (α_2 helix), L395 and D394.

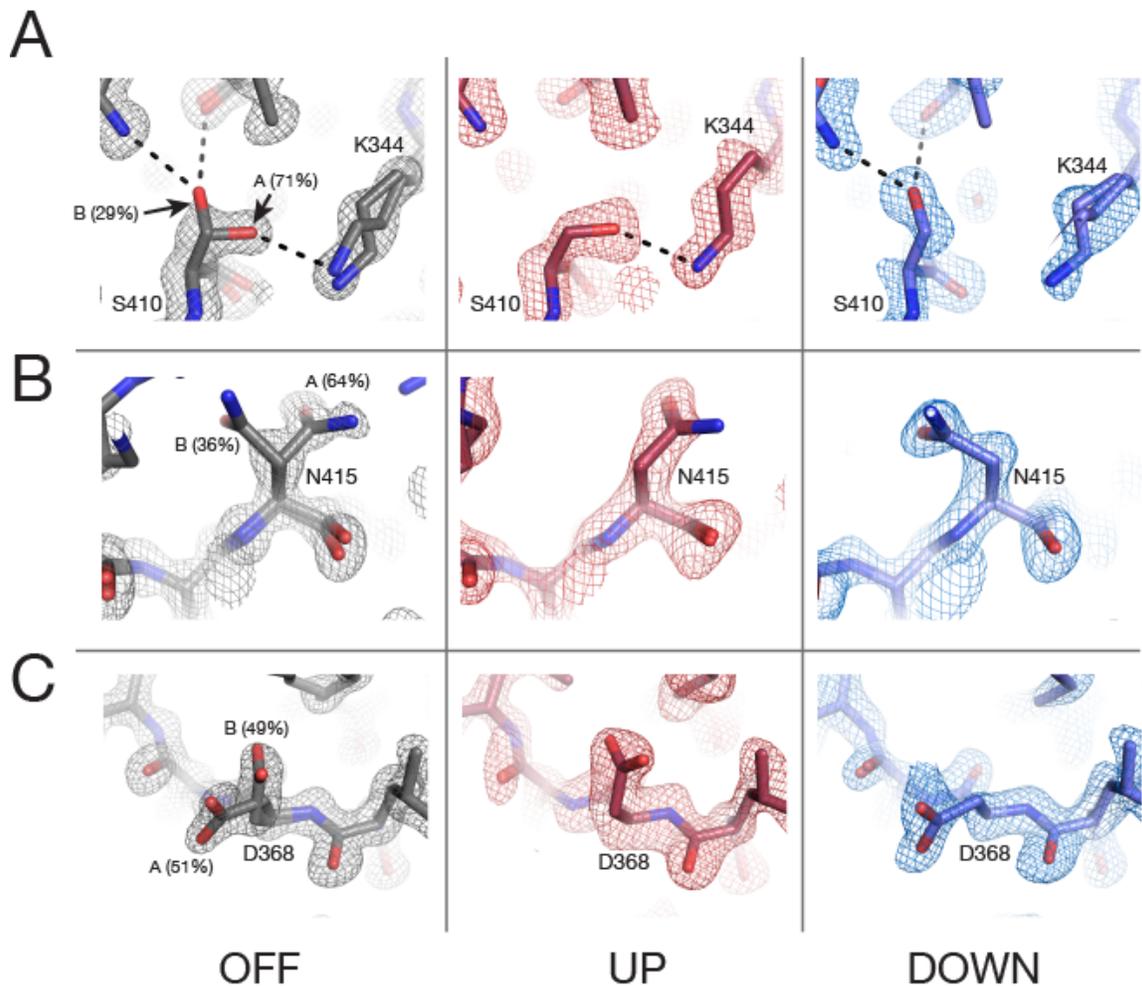


Figure 11. Biasing of preexisting conformational states in the LNX2^{PDZ2} ground state structure by the external electric field. **A.** In a high-resolution (1.1 Å) room temperature structure of the voltage-OFF ground state of LNX2^{PDZ2}, S410 shows partial occupancy in two rotameric states that define different hydrogen bonding patterns (left panel). This conformational degeneracy is biased in the presence of the electric field (8 kV, 200 ns delay), such that the *up* and *down* models each adopts one of the two ground state configurations (middle and right panel). **B–C.** Similarly, the N415 (**B**) and D368 (**C**) side chains show two alternate rotameric states in the OFF models (left panels), but become locked into one configuration in the ON models, depending on the direction of the applied field (middle and right panels). Thus, the electric field alters the relative free energy of naturally preexisting states, consistent with the notion that EF-X subtly explores the low-lying energetic architecture of the protein ground state.

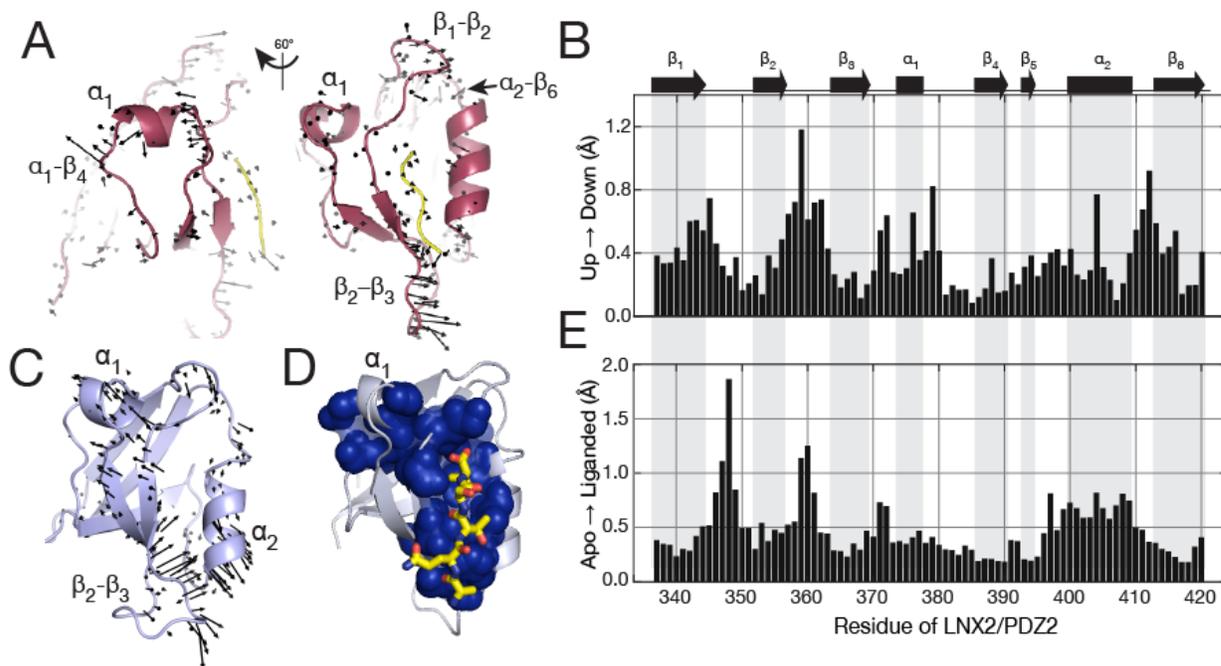


Figure 12. The relationship of electric field induced conformational change and PDZ function. **A.** Electric field-induced structural changes in LNX2^{PDZ2} (8 kV, 200 ns dataset); shown are two views, with vectors representing the displacements of main chain atoms transitioning from the up to down models. The motions are most prominent in the β_1 - β_2 , β_2 - β_3 , α_1 , and α_2 - β_6 regions. **B.** A plot of average displacement magnitudes of backbone atoms for each residue in going from the *up* to *down* state, providing a more global view of the electric field-induced effects. **C.** Systematic motions of backbone atoms due to ligand binding (vectors show apo to liganded) in high-resolution structures of 11 diverse homologs of the PDZ family. The motions occur in similar regions as in panel A, but also include the α_2 helix. **D.** The protein sector (blue spheres), a group of collectively evolving amino acid positions in a large sequence alignment of the PDZ protein family (PFAM 27.0); the sector connects the ligand binding pocket to the β_2 - β_3 segment and to the α_1 - β_4 surface through the carboxylate binding loop (CBL) and the α_1 helix. **E.** The mean displacements of backbone atoms per corresponding LNX2^{PDZ2} residue in the apo to liganded transition in the ensemble of 11 PDZ homologs. Statistical comparison of this pattern with that for the up to down transition (**B**) shows strong correlation ($p < 0.001$ Fisher Z-test).

5 Tables

Conformational change	Transition dipole moment (eÅ)	Energetic bias at 1 MV/cm (<i>kT</i>)
180° flip of a water molecule ⁷³	0.8	0.3
180° flip of peptide bond ⁷³	1.5	0.6
Rotamer flip of a protonated histidine	5.0	2.0
20° turn of 3-turn α helix dipole ⁷⁴	5.3	2.1
2-ion hop in KcsA channel ^{75,76}	7.0	2.8
GPCR gating (net, m2R GPCR) ²⁴	~20	~8
Gating of K ⁺ channel ⁷⁷	~100	~40

Table 1. Estimates of dipole moments associated with conformational changes. Indicated is the energetic bias in units of thermal energy at 20°C, assuming the motion dipole moment is parallel to the electric field. Transition dipole moments were estimated based on the indicated references and for the histidine side chain based on measurements in PyMOL.

Experiment	Protein*	Voltage (kV) [†]	τ (ns) [†]	X-ray delay (ns)	EF-pulses [‡] per frame	Total # frames [§]	Total # EF-pulses	Assessment
2-03	Lysozyme	5	200	150	5	154	770	Too mosaic
(same)		8	200	60	5	63	315	
2-05	Lysozyme	8	200	150	5	62	310	Multiple lattices
2-06	Lysozyme	8	200	150	5	61	305	Multiple lattices
(same)		8	200	60	5	70	355	
2-15	Syntenin ^{PDZ2}	6	500	450	5	63	315	Multiple lattices
2-17	LNx2 ^{PDZ2}	8	200	150	10	62	620	Multiple lattices
2-18	LNx2 ^{PDZ2}	8	200	150	7	62	434	Multiple lattices
2-22	LNx2 ^{PDZ2}	8	200	150	7	63	441	Good, limited completeness
(same)		8	500	450	7	62	434	
(same)		8	1,000	950	7	9	63	
2-56	Lysozyme	4	200	150	7	61	427	Multiple lattices
(same)		5	200	150	7	3	21	Multiple lattices
(same)		8	200	150	7	3	21	Explosion
2-64	Lysozyme	4	200	150	5	40	200	Electrostriction
(same)		5	200	150	1	61	61	Weak diffraction
(same)		8	200	150	1	71	71	Weak diffraction
2-66	Lysozyme	6	200	150	1	481	481	Weak diffraction
(same)		8	200	150	1	241	241	Weak diffraction
3-06	Lysozyme	6.4	200	150	5	80	400	Weak diffraction
3-08	Lysozyme	6.4	200	150	3	181	543	Multiple lattices
3-09	Lysozyme	6.6	200	150	5	111	555	Multiple lattices
3-10	Lysozyme	6.4	200	150	3	120	360	Too mosaic
3-17	PICK1 ^{PDZ}	3.0	200	150	5	182	910	Good
(same)		4.0	200	150	10	46	460	
3-24	LNx2 ^{PDZ2}	6.0	200	150	1	213	213	Good

3-25	LNX2 ^{PDZ2}	6.0	200	150	1	404	404	Good (Table 7)
3-35	LNX2 ^{PDZ2}	6.0	250	50	1	363	363	See Main Text.
(same)		6.0	250	100	1	363	363	See Main Text.
(same)		6.0	250	200	1	363	363	See Main Text.
3-44	LNX2 ^{PDZ2}		200	150	1	182	182	Weak diffraction
3-48	LNX2 ^{PDZ2}		200	150	1	145	145	Weak diffraction
3-49	LNX2 ^{PDZ2}	5.0	200	150	1	235	235	Good (Table 7)
3-62	NaK2K	±1.0	200	150	4	198	792	Too Mosaic
(same)		-3.0	200	150	4	31	124	
(same)		-5.0	200	150	4	31	124	
(same)		-7.0	200	150	4	62	248	
(same)		-7.0	550	500	4	15	60	1,408 total
3-80	NaK2K	-2.4	200	150	5	386	1,930	Too Mosaic
3-82	NaK2K	-5.0	200	150	8	91	728	Too Mosaic

Table 2. Experimental parameters for select data sets. Experiments were performed using "pink beam" radiation with a bandwidth of ~5%¹⁸. The use of a pink beam places limits acceptable crystal mosaicity to about 0.1°. As a result, diffraction patterns obtained were often unsuitable for further analysis. *Most crystals were 50–100 µm thick based on visual inspection. †Electric field pulse duration. ‡Note that multiple electric field (EF) and X-ray pulses were typically required per frame. §Frames with EF. Because OFF frames were also collected, the total number of X-ray exposures was typically twice or three times the number of EF pulses.

Dataset	OFF	ON (50ns)	ON (100ns)	ON (200ns)
Number of images	361	361	361	359
Resolution*		100-1.8Å (1.88-1.8Å)		
		<i>Single reflections</i>		
$R_{\text{merge}} (F^2)$	0.092	0.079	0.095	0.081
$R_{\text{merge}} (F)$	0.053	0.046	0.055	0.047
$\langle F/\sigma_F \rangle$	39	46	38	45
Total observations	64,421	63,822	62,033	60,114
Unique observations	11,978	11,847	11,651	11,490
Redundancy	5.4	5.4	5.3	5.2
		<i>Single and Harmonic reflections</i>		
Unique observations	12,049	11,942	11,729	11,568
Completeness* (P1)	74.9% (29.1%)	74.2% (27.8%)	72.9% (25.6%)	71.9% (24.2%)
Completeness* (C2)	79.3% (38.6%)	78.7% (37.0%)	77.5% (34.3%)	77.0% (33.9%)

Table 3. Data reductions statistics for LNX2^{PDZ2} time series data set. Data were indexed, integrated, scaled and merged in Precognition. Completeness statistics were determined with mtzdump. Note that, unlike in conventional data reduction software, only reflections passing several cutoffs are included in the final merged data set in Precognition. This biases estimates of completeness.

Data set	Residues with peaks near backbone O	<i>P</i> *
OFF > +3.5 σ	367	>0.1
< -3.5 σ	339, 358, 409, 426	>0.1
Both	--	>0.1
50 ns > +3.5 σ	340, 347, 370, 379, 381, 399, 410, 414	>0.1
< -3.5 σ	349, 381, 382, 385, 390, 398, 414, 427	>0.1
Both	381, 414	>0.1
100 ns > +3.5 σ	340, 357, 358, 359, 365, 370, 378, 379, 382, 384, 408, 409, 410, 414, 420	0.002
< -3.5 σ	335, 336, 340, 355, 357, 358, 362, 382, 398, 415, 420	0.05
Both	340, 357, 358, 382, 420	0.01
200 ns > +3.5 σ	340, 362, 370, 378, 379, 382, 384, 409, 410, 414	>0.1
< -3.5 σ	337, 350, 355, 357, 358, 362, 365, 382, 391, 394, 397, 404, 410, 415, 416	0.007
Both	362, 382, 409, 410	0.09

Table 4. Occurrence of internal difference density peaks near backbone oxygen atoms. *Probability that at least this many residues will have peaks of the indicated type near backbone oxygen atoms, based on binomial sampling among the observed peaks in each internal difference electron density map. That is, the increased map variance and increased number of peaks at large deviations from zero are, *a priori*, excluded from causing the significance of obtained results.

Deviation (σ)	Nearest Residue	Near backbone (<1.5 Å)?
5.25	N384	Y
5.00	S410	N
4.93	S410	Y
4.89	L342	N
4.88	D394	N
4.87	V356	N
4.81	R357	N
4.76	S382	Y
4.74	P362	Y
4.67	R358	N
4.66	R380	N
4.60	R358	Y
4.56	D394	N
4.51	S383	Y
4.48	R413	N
4.42	D378	Y
4.42	L416	Y
4.41	V414	Y
4.36	N415	Y
4.34	R358	Y
4.32	--	N
4.29	R386	N
4.28	Y397	N
4.27	E371	N
4.24	--	N
4.19	Y397	Y
4.19	E401	Y
4.09	R357	Y
4.08	D385	N
4.08	S382	Y

Table 5. Internal difference electron density peaks at 200 ns. Shown are the 30 strongest internal difference electron density peaks. Note that each peak occurs twice with positive and twice with negative sign when using the C2 unit cell. Hence, we report peak strengths as positive numbers. The location of peaks was determined in Coot using a map sampled at 0.3 Å (grid sampling rate = 3). The symbol '--' indicates that no residue atoms were found within 3 Å. We find 15 of 30 peaks near the protein backbone (expected by chance: 8.9; $p = 0.007$). We find 14 out of 28 peaks near charged residues ($p = 0.012$ by simple counting; $p = 0.047$ by overall residue Van der Waals volumes). Note that the standard deviation of the 200 ns map is $1.3\times$ larger than for the OFF map, such that the for the highest peak $5.25 \sigma_{\text{ON}} \approx 6.88 \sigma_{\text{OFF}}$.

Deviation (σ)	Nearest Residue	Near backbone (1.5 Å)?
4.75	R413	N
4.65	Y397	N
4.61	R413	N
4.31	K423	N
4.29	--	N
4.27	Q339	N
4.21	L428	N
4.20	Q408	N
4.17	E412	N
4.15	Q339	Y
4.11	Q339	N
4.10	--	N
4.10	I426	N
4.08	E349	N

Table 6. Internal difference electron density peaks for OFF data. Shown are the internal difference electron density (DED) peaks above 4.08 for comparison with Table 5 (determined in the same manner). Localization near charged residues is not statistically significant ($p > 0.1$), while peaks are biased away from the protein backbone ($p = 0.99$).

Dataset	OFF			ON (150 ns)		
	Crystal e25	Crystal e49	Merged*	Crystal e25	Crystal e49	Merged
Number of images	218	180	398	212	180	(392)
Resolution*			100-1.8Å (1.88-1.8Å)			
			<i>Single reflections</i>			
$R_{\text{merge}}(F^2)^{\#}$	0.103	0.095	0.096	0.099	0.098	0.107
$R_{\text{merge}}(F)^{\#}$	0.056	0.055	0.051	0.055	0.057	0.054
$\langle F/\sigma_F \rangle$	28	32	43 ^b	29	30	39 ^b
Total observations	27,973	29,671	57,644	22,409	29,033	51,442
Unique observations	9,668	11,018	n/a	7,966	10,898	n/a
Redundancy**	2.9	2.7	4.8	2.8	2.7	4.5
			<i>Single and Harmonic reflections</i>			
Unique observations	9,721	11,099	11,899	8,037	10,960	11,456
Completeness**^a	61.2%	69.8%	75.0%	50.5%	68.9%	72.2%
(P1)	(15.7%)	(25.8%)	(31.3%)	(5.40%)	(24.4%)	(26.3%)
Completeness**^a	67.8%	77.5%	79.7%	57.3%	76.8%	78.0%
(C2)	(23.0%)	(36.4%)	(41.8%)	(8.7%)	(35.9%)	(37.7%)

Table 7. Data reduction statistics for two additional crystals of LNX2^{PDZ2} and their merged data set. Data for crystals e25 and e49 were “cut” at 2 and 3 σ , respectively, for merging in Epinorm. *Merged reflections (of each crystal separately) were merged in MATLAB with weights based on their associated errors. **For data merged from both crystals, reported redundancy based on single and harmonic reflections instead. [#]For data merged from both crystals, we report the R factor for merging between these two data sets, unweighted, relative to unweighted mean; weighted $R_{\text{merge}}(F)$ are 0.037 and 0.035 for OFF and ON data, respectively. (a) Completeness in P1 after removal of reflections forbidden by residual translational symmetry; completeness calculated using CCP4/6.4.0 mtzdump. (b) Geometric means (arithmetic means ~ 90 , dominated by outliers).

Position	Amino acid	ASA*	Comments†	
382	S	Ser	0.45	
357	R	Arg	0.62	Allosteric position.
358	R	Arg	0.30	Allosteric position.
410	S	Ser	0.14	Allosteric, H-bond to K344 (main text)
415	N	Asn	0.10	
384	N	Asn	0.62	Conserved position.
378	D	Asp	0.39	
385	D	Asp	0.01	Most conserved position.
362	P	Pro	0.51	
350	Q	Gln	0.66	First position of "GLGF" motif.
375	A	Ala	0.00	Allosteric position.
340	V	Val	0.00	
383	S	Ser	0.44	
425	E	Glu	0.49	Ligand position -3 from C-terminus.
413	R	Arg	0.54	
336	E	Glu	0.58	
397	Y	Tyr	0.76	
416	L	Leu	0.01	Conserved position.
379	G	Gly	0.69	
414	V	Val	0.00	Conserved position.

Table 8. Top 20 residues by signal near the backbone at 200 ns. Signal (integrated absolute difference electron density above $2.5 \sigma_{\text{OFF}}$). Polar residues (R, K, E, D, H, Q, N, S, T, Y) are slightly overrepresented ($p = 0.056$, two-sided test, $N = 20$). Charged residues (positive charge: His, Arg, Lys; negative charge: Asp, Glu), are not significantly overrepresented ($p = 0.33$). *Accessible surface area (fractional): Average accessible surface area is 0.37, not significantly different from the protein as a whole (0.37). Fractions of residues with ASA < 0.10, 0.15 and 0.20 are not significantly smaller than for the protein as a whole (all $p > 0.05$ by the bootstrap method). ASA was calculated using default settings in ASAView. † "Conserved" is defined as among the 10 most conserved positions in the HMM profile for PFAM00595 (≥ 1.45 bits). The assessment of allostery is based on the bottom row of Figure 2.

	LNx2 ^{PDZ2} (OFF)	LNx2 ^{PDZ2} (200 ns)	Extrapolated Differences (8x)
Data collection[‡]			
Space group	C2 [†]	P1	P1
Cell dimensions			
<i>a, b, c</i> (Å)	65.30, 39.45, 39.01	38.15, 38.15, 39.01	38.15, 38.15, 39.01
α, β, γ (°)	90, 117.54, 90	113.31, 113.31, 62.28	113.31, 113.31, 62.28
Resolution (Å)	30.08-1.80 (2.0-1.8)*	30.08-1.80 (2.0-1.8)	30.08-1.80 (1.86-1.80)
<i>R</i> _{sym} or <i>R</i> _{merge}	0.088 (0.051)	0.087 (0.053)	n/a
<i>I</i> / σ <i>I</i>	20.7 (37.1)	20.4 (39.9)	6.98 (0.67)
Completeness (%) [‡]	75.1 (42.5)	72.4 (38.1)	70.2 (17.8)
Redundancy	5.8 (3.9)	5.7 (3.6)	n/a
Refinement			
Resolution (Å)	30-1.8 (1.88-1.8)	30-1.8	30-1.8 (1.88-1.8) [§]
No. reflections [¶]	6,565 (288)	11,568	11,291 (288)
<i>R</i> _{work} / <i>R</i> _{free} (%) [¶]	13.2/14.8		28.9/31.3
No. atoms (excl. H)	929		1,883
Protein	829		1,712
Ligand/ion	0		6
Water	94		165
B-factors	21.9		16.7
Protein	19.3		16.0
Ligand/ion	n/a		49.5
Water	35.6		22.5
R.m.s deviations			
Bond lengths (Å)	0.020		0.018
Bond angles (°)	1.63		1.80

Table 9. Data collection and refinement statistics for LN_X2PDZ₂ (EFX experiment). All data were collected from a single crystal of LN_X2PDZ₂ using Laue crystallography. [‡]See **Table 3** for data collection and reduction statistics as reported traditionally for Laue crystallography, including assessment of completeness in both C2 and P1. Reported data collection statistics refer to P1. [†]For the purpose of refinement of the OFF model, P1 reflections were merged according to C2 symmetry (merging *R* factor for this: 0.077). *Highest resolution shell is shown in parentheses. Data reduction in Precognition (Renz Research) differs from conventional data reduction in that weak spots are discarded *a priori*, resulting in low apparent completeness and high apparent signal and *R*_{merge}, especially at high-resolution. Note also that data statistics are reported after global scaling. Subsequent local scaling slightly affects statistics but this scaling mode does not report full last shell statistics. Extrapolated differences were assessed in Xtriage (PHENIX). [§]Data were retained to the resolution of the two “parent” data sets (OFF and 200 ns); effective resolution based on propagation of errors: 2.3 Å; completeness over 30-2.3 Å: 89.9%. [¶]Test sets comprised 5 and 10% of reflections for refinement of the OFF model and refinement against extrapolated structure factors, respectively. [¶]The matching number of reflections in the high-resolution shell is coincidental.

LNx2 ^{PDZ2} (high-resolution)		
Data collection		
Space group	C2	
Cell dimensions		
<i>a, b, c</i> (Å)	64.91, 39.29, 38.80	
α, β, γ (°)	90, 117.41, 90	
Resolution (Å)	34.45-1.05 (1.05-1.01)*	
R_{sym} or R_{merge}	0.051 (0.54)	
$I/\sigma I$	12.84 (0.45)	
Completeness (%) [†]	77.6 (3.0)	
Redundancy	5.8 (1.2)	
Refinement		
	With alternate conformations	No alternate conformations
Resolution (Å)	34.45-1.05	34.45-1.05
No. reflections	35,251 (137)	35,251 (137)
$R_{\text{work}}/ R_{\text{free}}$	11.9/13.4	12.6/14.0
No. atoms (non-H)	1,039	824
Protein	929	719
Ligand/ion	0	0
Water	104	99
B-factors	19.2	19.7
Protein	17.1	17.3
Ligand/ion	n/a	n/a
Water	37.1	36.2
R.m.s deviations		
Bond lengths (Å)	0.022	0.023
Bond angles (°)	1.86	1.88

Table 10. Data collection and refinement statistics for LNx2^{PDZ2} by room-temperature crystallography. Based on a data collected from a single crystal. *Highest resolution shell is shown in parentheses; data were retained based on $CC_{1/2} > 50\%$; $I/\sigma I$ falls below 2 at 1.08 Å. [†]Completeness over 50–1.5 Å: 98.2%. Completeness falls below 50% ($I/\sigma I = 1$) at 1.1 Å.

6 Works cited

The work presented here is essentially a reproduction of a draft of a paper currently under consideration for publication. The authors of this manuscript are, in order, Doeke R. Hekstra, myself, Michael A. Socolich, Robert W. Henning, Vukica Šrajer, and Rama Ranganathan.

1. Alberts, B. The cell as a collection of protein machines: preparing the next generation of molecular biologists. *Cell* **92**, 291–294 (1998).
2. Patel, S. S., Wong, I. & Johnson, K. A. Pre-steady-state kinetic analysis of processive DNA replication including complete characterization of an exonuclease-deficient mutant. *Biochemistry* **30**, 511–525 (1991).
3. Boehr, D. D., McElheny, D., Dyson, H. J. & Wright, P. E. The dynamic energy landscape of dihydrofolate reductase catalysis. *Science* **313**, 1638–1642 (2006).
4. Noji, H., Yasuda, R., Yoshida, M. & Kinosita, K. Direct observation of the rotation of F1-ATPase. *Nature* **386**, 299–302 (1997).
5. Tao, X., Lee, A., Limapichat, W., Dougherty, D. A. & MacKinnon, R. A gating charge transfer center in voltage sensors. *Science* **328**, 67–73 (2010).
6. Howard, J. The mechanics of force generation by kinesin. *Biophysj* **68**, 245S–253S–253S–255S (1995).
7. Vale, R. D., Reese, T. S. & Sheetz, M. P. Identification of a novel force-generating protein, kinesin, involved in microtubule-based motility. *Cell* **42**, 39–50 (1985).
8. Gether, U. Uncovering molecular mechanisms involved in activation of G protein-coupled receptors. *Endocr. Rev.* **21**, 90–113 (2000).
9. Monod, J., Wyman, J. & Changeux, J. On Nature of Allosteric Transitions — a Plausible Model. *J Mol Biol* **12**, 88–118 (1965).
10. Perutz, M. F. Stereochemistry of cooperative effects in haemoglobin. *Nature* **228**, 726–739 (1970).
11. Cooper, A. & Dryden, D. T. F. Allostery without conformational change. *Eur Biophys J* **11**, 103–109 (1984).
12. Popovych, N., Sun, S., Ebricht, R. H. & Kalodimos, C. G. Dynamically driven protein allostery. *Nat. Struct. Mol. Biol.* **13**, 831–838 (2006).
13. Karplus, M. & McCammon, J. A. Molecular dynamics simulations of biomolecules. *Nat Struct Biol* **9**, 646–652 (2002).
14. Kay, L. E. Protein dynamics from NMR. *Nat Struct Biol* **5 Suppl**, 513–517 (1998).
15. Fraser, J. S. *et al.* Hidden alternative structures of proline isomerase essential for catalysis. *Nature* **462**, 669–U149 (2009).
16. Sekhar, A. & Kay, L. E. NMR paves the way for atomic level descriptions of sparsely populated, transiently formed biomolecular conformers. *Proc Natl Acad Sci USA* **110**, 12867–12874 (2013).
17. Moffat, K. Time-resolved biochemical crystallography: a mechanistic perspective. *Chem. Rev.* **101**, 1569–1581 (2001).
18. Graber, T. *et al.* BioCARS: a synchrotron resource for time-resolved X-ray science.

- Journal of synchrotron radiation* **18**, 658–670 (2011).
19. Glowacka, J. M. *et al.* Time-resolved pump-probe experiments at the LCLS. *Opt Express* **18**, 17620–17630 (2010).
 20. Ren, Z. *et al.* A molecular movie at 1.8 Å resolution displays the photocycle of photoactive yellow protein, a eubacterial blue-light receptor, from nanoseconds to seconds. *Biochemistry* **40**, 13788–13801 (2001).
 21. Fuentes, E. J., Gilmore, S. A., Mauldin, R. V. & Lee, A. L. Evaluation of energetic and dynamic coupling networks in a PDZ domain protein. *J Mol Biol* **364**, 337–351 (2006).
 22. Petit, C. M., Zhang, J., Sapienza, P. J., Fuentes, E. J. & Lee, A. L. Hidden dynamic allostery in a PDZ domain. *Proc Natl Acad Sci USA* **106**, 18249–18254 (2009).
 23. Whitney, D. S., Peterson, F. C., Kovrigina, E. L. & Volkman, B. F. Allosteric activation of the Par-6 PDZ via a partial unfolding transition. *J. Am. Chem. Soc.* **135**, 9377–9383 (2013).
 24. Ben-Chaim, Y. *et al.* Movement of gating charge is coupled to ligand binding in a G-protein-coupled receptor. *Nature* **444**, 106–109 (2006).
 25. Morozova TYa *et al.* Ionic conductivity, transference numbers, composition and mobility of ions in cross-linked lysozyme crystals. *Biophys. Chem.* **60**, 1–16 (1996).
 26. Case, D. A. *et al.* The Amber biomolecular simulation programs. *Journal of Computational Chemistry* **26**, 1668–1688 (2005).
 27. Roos, A. K. *et al.* Crystal Structure of the second PDZ domain of Human Numb-Binding Protein 2. (2008). doi:10.2210/pdb2vwr/pdb
 28. Rice, D. S., Northcutt, G. M. & Kurschner, C. The Lnx family proteins function as molecular scaffolds for Numb family proteins. *Mol. Cell. Neurosci.* **18**, 525–540 (2001).
 29. Sheng, M. & Sala, C. PDZ domains and the organization of supramolecular complexes. *Annu. Rev. Neurosci.* **24**, 1–29 (2001).
 30. Fuentes, E. J., Der, C. J. & Lee, A. L. Ligand-dependent dynamics and intramolecular signaling in a PDZ domain. *J Mol Biol* **335**, 1105–1115 (2004).
 31. Lockless, S. W. & Ranganathan, R. Evolutionarily conserved pathways of energetic connectivity in protein families. *Science* **286**, 295–299 (1999).
 32. Peterson, F. C., Penkert, R. R., Volkman, B. F. & Prehoda, K. E. Cdc42 regulates the Par-6 PDZ domain through an allosteric CRIB-PDZ transition. *Mol Cell* **13**, 665–676 (2004).
 33. McLaughlin, R. N., Poelwijk, F. J., Raman, A., Gosal, W. S. & Ranganathan, R. The spatial architecture of protein function and adaptation. *Nature* **491**, 138–142 (2012).
 34. Genick, U. K. *et al.* Structure of a protein photocycle intermediate by millisecond time-resolved crystallography. *Science* **275**, 1471–1475 (1997).
 35. Tenboer, J. *et al.* Time-resolved serial crystallography captures high-resolution intermediates of photoactive yellow protein. *Science* **346**, 1242–1246 (2014).
 36. Long, J. *et al.* Supramodular nature of GRIP1 revealed by the structure of its PDZ12 tandem in complex with the carboxyl tail of Frs1. *J Mol Biol* **375**, 1457–1468 (2008).
 37. Feng, W., Shi, Y., Li, M. & Zhang, M. Tandem PDZ repeats in glutamate receptor-interacting proteins have a novel mode of PDZ domain-mediated target binding. *Nat Struct Biol* **10**, 972–978 (2003).
 38. Im, Y. J. *et al.* Crystal structure of GRIP1 PDZ6-peptide complex reveals the structural basis for class II PDZ target recognition and PDZ domain-mediated multimerization. *J Biol Chem* **278**, 8501–8507 (2003).
 39. van den Berk, L. C. J. *et al.* An Allosteric Intramolecular PDZ-PDZ Interaction Modulates

- PTP-BL PDZ2 Binding Specificity †. *Biochemistry* **46**, 13629–13637 (2007).
40. Doyle, D. A. *et al.* Crystal structures of a complexed and peptide-free membrane protein-binding domain: molecular basis of peptide recognition by PDZ. *Cell* **85**, 1067–1076 (1996).
 41. Kang, B. S., Cooper, D. R., Devedjiev, Y., Derewenda, U. & Derewenda, Z. S. Molecular roots of degenerate specificity in syntenin's PDZ2 domain: reassessment of the PDZ recognition paradigm. *Structure* **11**, 845–853 (2003).
 42. Halabi, N., Rivoire, O., Leibler, S. & Ranganathan, R. Protein sectors: evolutionary units of three-dimensional structure. *Cell* **138**, 774–786 (2009).
 43. Süel, G. M., Lockless, S. W., Wall, M. A. & Ranganathan, R. Evolutionarily conserved networks of residues mediate allosteric communication in proteins. *Nat Struct Biol* **10**, 59–69 (2003).
 44. Svoboda, K., Schmidt, C. F., Schnapp, B. J. & Block, S. M. Direct observation of kinesin stepping by optical trapping interferometry. *Nature* **365**, 721–727 (1993).
 45. Gebhardt, J. C. M., Bornschrögl, T. & Rief, M. Full distance-resolved folding energy landscape of one single protein molecule. *Proc Natl Acad Sci USA* **107**, 2013–2018 (2010).
 46. Schmidt, M., Rajagopal, S., Ren, Z. & Moffat, K. Application of singular value decomposition to the analysis of time-resolved macromolecular x-ray data. *Biophys J* **84**, 2112–2129 (2003).
 47. Wang, J., Cieplak, P. & Kollman, P. A. How well does a restrained electrostatic potential (RESP) model perform in calculating conformational energies of organic and biological molecules? *Journal of Computational Chemistry* **21**, 1049–1074 (2000).
 48. Hornak, V. *et al.* Comparison of multiple Amber force fields and development of improved protein backbone parameters. *Proteins* **65**, 712–725 (2006).
 49. Wang, J., Wolf, R. M., Caldwell, J. W., Kollman, P. A. & Case, D. A. Development and testing of a general amber force field. *Journal of Computational Chemistry* **25**, 1157–1174 (2004).
 50. Bayly, C. I., Cieplak, P., Cornell, W. & Kollman, P. A. A well-behaved electrostatic potential based method using charge restraints for deriving atomic charges: the RESP model. *The Journal of Physical Chemistry* **97**, 10269–10280 (1993).
 51. Darden, T., Perera, L., Li, L. & Pedersen, L. New tricks for modelers from the crystallography toolkit: the particle mesh Ewald algorithm and its use in nucleic acid simulations. *Structure* **7**, R55–60 (1999).
 52. Sagui, C., Pedersen, L. G. & Darden, T. A. Towards an accurate representation of electrostatics in classical force fields: efficient implementation of multipolar interactions in biomolecular simulations. *J Chem Phys* **120**, 73–87 (2004).
 53. Miyamoto, S. & Kollman, P. A. Molecular dynamics studies of calixspherand complexes with alkali metal cations: calculation of the absolute and relative free energies of binding of cations to a calixspherand. *J. Am. Chem. Soc.* **114**, 3668–3674 (1992).
 54. Izaguirre, J. A., Catarello, D. P., Wozniak, J. M. & Skeel, R. D. Langevin stabilization of molecular dynamics. *J Chem Phys* **114**, 2090–2098 (2001).
 55. Michaud-Agrawal, N., Denning, E. J., Woolf, T. B. & Beckstein, O. MDAAnalysis: a toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry* **32**, 2319–2327 (2011).
 56. Cock, P. J. A. *et al.* Biopython: freely available Python tools for computational molecular

- biology and bioinformatics. *Bioinformatics* **25**, 1422–1423 (2009).
57. van der Walt, S., Colbert, S. C. & Varoquaux, G. The NumPy Array: A Structure for Efficient Numerical Computation. *Comput. Sci. Eng.* **13**, 22–30 (2011).
 58. Kalinin, Y. *et al.* A new sample mounting technique for room-temperature macromolecular crystallography. *J Appl Crystallogr* **38**, 333–339 (2005).
 59. Otwinowski, Z. & Minor, W. in *Macromolecular Crystallography Part A* **276**, 307–326 (Elsevier, 1997).
 60. Diederichs, K. Some aspects of quantitative analysis and correction of radiation damage. *Acta Crystallogr D Biol Crystallogr* **62**, 96–101 (2005).
 61. Adams, P. D. *et al.* PHENIX: a comprehensive Python-based system for macromolecular structure solution. *Acta Crystallogr D Biol Crystallogr* **66**, 213–221 (2010).
 62. Emsley, P., Lohkamp, B., Scott, W. G. & Cowtan, K. Features and development of Coot. *Acta Crystallogr D Biol Crystallogr* **66**, 486–501 (2010).
 63. Winn, M. D. *et al.* Overview of the CCP4 suite and current developments. *Acta Crystallogr D Biol Crystallogr* **67**, 235–242 (2011).
 64. Schmidt, M. *et al.* Ligand migration pathway and protein dynamics in myoglobin: a time-resolved crystallographic study on L29W MbCO. *Proc Natl Acad Sci USA* **102**, 11704–11709 (2005).
 65. Matthews, B. W. & Czerwinski, E. W. Local scaling: a method to reduce systematic errors in isomorphous replacement and anomalous scattering measurements. *Acta Crystallographica Section A* **31**, 480–487 (1975).
 66. Pettersen, E. F. *et al.* UCSF Chimera--a visualization system for exploratory research and analysis. *Journal of Computational Chemistry* **25**, 1605–1612 (2004).
 67. Pei, J. & Grishin, N. V. PROMALS3D: multiple protein sequence alignment enhanced with evolutionary and three-dimensional structural information. *Methods Mol Biol* **1079**, 263–271 (2014).
 68. Poole, A. Collective Structural Modes in Proteins: A Case Study in the PDZ Domain. (2012).
 69. Niu, X. *et al.* Interesting structural and dynamical behaviors exhibited by the AF-6 PDZ domain upon Bcr peptide binding. *Biochemistry* **46**, 15042–15053 (2007).
 70. Kong, Y. & Karplus, M. Signaling pathways of PDZ2 domain: a molecular dynamics interaction correlation analysis. *Proteins* **74**, 145–154 (2009).
 71. Kozlov, G., Gehring, K. & Ekiel, I. Solution Structure of the PDZ2 Domain from Human Phosphatase hPTP1E and Its Interactions with C-Terminal Peptides from the Fas Receptor †,‡. *Biochemistry* **39**, 2572–2580 (2000).
 72. Shepherd, T. R. *et al.* The Tiam1 PDZ domain couples to Syndecan1 and promotes cell-matrix adhesion. *J Mol Biol* **398**, 730–746 (2010).
 73. Pethig, R. Protein-water interactions determined by dielectric methods. *Annu Rev Phys Chem* **43**, 177–205 (1992).
 74. Hol, W. G., van Duijnen, P. T. & Berendsen, H. J. The alpha-helix dipole and the properties of proteins. *Nature* **273**, 443–446 (1978).
 75. Zhou, Y. & MacKinnon, R. The occupancy of ions in the K⁺ selectivity filter: charge balance and coupling of ion binding to a protein conformational change underlie high conduction rates. *J Mol Biol* **333**, 965–975 (2003).
 76. Zhou, Y., Morais-Cabral, J. H., Kaufman, A. & MacKinnon, R. Chemistry of ion coordination and hydration revealed by a K⁺ channel-Fab complex at 2.0 Å resolution.

- Nature* **414**, 43–48 (2001).
77. Khalili-Araghi, F. *et al.* Calculation of the gating charge for the Kv1.2 voltage-activated potassium channel. *Biophys J* **98**, 2189–2198 (2010).

Chapter V

Conclusions and future directions

1 In summary

1.1 The response of the PDZ proteins to perturbation is complex

So, what have we learned? Together, these studies explore the mechanical response of various PDZ domains to three different types of perturbation—mutation, thermal agitation, and strong electric fields. While each study certainly stands on its own as far as specific conclusions are concerned, one can ask just how complementary conclusions are across the studies.

In general, we find that the response of proteins to each type of perturbation is complex but somewhat consistent. In PSD-95 PDZ3, the mutation of position 330 from glycine to threonine leads to the propagation of functionally relevant changes throughout the domain, with structural changes occurring at distinct locations somewhat removed from the binding site such as the α_1 helix. These changes are moderately consistent with super-ensembles built for wild-type PSD-95 PDZ3, where the first mode describes the fluctuation of the C-terminus of the CRIPT ligand with a number of other positions, including the α_1 helix. Averaging over homologous super-ensembles reveals a similar mode present in the case of many other proteins as well. Furthermore, structural studies of another PDZ domain, LNX2 PDZ2, in the presence of an electric field reveal a similar enrichment for large displacements in and around α_1 relative to the rest of the domain. Finally, all of these results are, at least with respect to the α_1 helix, consistent with patterns of coevolution in the domain.

That being said, the the degree of overlap one should expect between each approach is not at all clear, and there are certainly differences between the datasets as well. Specific future aims for

the experiments have been discussed in their respective chapters, so here we will consider some potentially interesting future experiments to assess their complementarity.

2 Future directions

2.1 Development of a mechanical model for conformational change in the PSD-95 PDZ3 mutant cycle

In Chapter II, we suggest that higher-order mutagenesis of key residues can be used to reveal otherwise hidden allostery in proteins. We show that, in the case of PSD-95 PDZ3, targeted mutation and careful structural analysis reveal a coupled path of interactions spanning the domain in an anisotropic manner, running from one end to the other along the ligand binding cleft. Importantly, this pattern is only evident upon considering the second-order cycle with wild-type and the G330T protein bound to either the wild-type CRIPT ligand or the mutant T₂F ligand. Introduction of another mutation, H372A, effectively breaks this pattern.

An important question remains—what underlying dynamics would yield this pattern, and how can they be revealed experimentally? Is this pattern reflective of dynamic allostery? In many ways, the ensemble refinement approach detailed in Chapter III is well-suited to answer this question. Consider the two cycles discussed. Four super ensembles could be constructed in the case of each cycle—for the first, a cycle with wild-type and the G330T mutant bound to each ligand, and for the second, a cycle with G330T and H372A-G330T bound to each ligand. The data could then be analyzed at two levels. First, decomposition of the mutual information matrix for a given super-ensemble would reveal something about the physical motions of residues in that protein. For example, this might reveal a mechanistic basis for the differences between the wild-type CRIPT and G330T CRIPT structures. The hypothesis that the H372A mutation mechanically disrupts the

allosteric pathway could also be tested—one would perhaps expect smaller, more fragmented modes in the case of the proteins with this mutation.

At the same time, our approach allows us to bypass these anecdotal sorts of comparisons. We could, for example, directly compare the resulting mutual information matrices \mathbf{I}^a to a reference matrix \mathbf{I}^{ref} (e.g. wild-type PDZ3 bound to CRIPT), where a denotes some mutation (e.g. G330T). The calculation of difference mutual information matrices would thus be particularly interesting. Explicitly, $\Delta\mathbf{I}^a = \mathbf{I}^a - \mathbf{I}^{\text{ref}}$. The average over columns of $\Delta\mathbf{I}^a$ is thus a vector with the average degree of difference mutual information for a given residue—analogous to Δr as presented in Chapter II. Similarly, for another mutation, b (e.g., the T₂F mutation), $\Delta\mathbf{I}^b = \mathbf{I}^b - \mathbf{I}^{\text{ref}}$ and the second order calculation can also be made, i.e., $\Delta\Delta\mathbf{I}^{a,b} = \Delta\mathbf{I}^{a,b} - (\Delta\mathbf{I}^a + \Delta\mathbf{I}^b)$, with the average over columns of $\Delta\Delta\mathbf{I}^{a,b}$ analogous to $\Delta\Delta r$. At the simplest level, analysis of elements of $\langle\Delta\mathbf{I}^a\rangle$ or $\langle\Delta\Delta\mathbf{I}^{a,b}\rangle$ would be a better approach for identifying significant fluctuations than the Δr or $\Delta\Delta r$ analysis—it does not require the heuristic backend developed by Stroud and Fauman¹, which may or may not still be appropriate after 20 years. But, why simply take the average? Either distance matrix can be decomposed to reveal patterns of higher-order coupling as well. This procedure could thus be used to reveal the coupling underlying the patterns observed in Chapter II, providing a path to a potentially direct mechanism for an otherwise inferential model. In principle, this line of work can be carried out using existing data and analytical tools presented in Chapters II and III.

Alternatively, the EFX method described in Chapter IV could be applied to crystals of several of these mutants in order to induce motions. Coupling between residues could then be directly assessed and compared to the inferred structural couplings from the analysis of $\Delta\Delta r$. From a practical perspective, this would be challenging—the presence of ~1 M sodium citrate in the PSD-

95 PDZ3 crystallization buffer leads to rapid Joule heating and subsequent crystal detonation at even lower field strengths—but certainly not impossible. Assuming that excess salt could be removed from crystals of the different mutants, it would be interesting to perform a series of crystallographic experiments. For each mutant, one would ideally collect diffraction dataset in the presence of an electric field of either polarity along three orthogonal axes (or fewer, depending on symmetry-breaking). For each structure, then, a matrix with $n \times 3$ dimensions can be constructed, where n is the number of atoms in the protein and each matrix element is thus the sum of displacement over two polarities along a given axis. Principal components analysis may then reveal the dominant collective features, which can be compared to expectation based on $\Delta\Delta r$ or super-ensemble analysis.

2.2 Towards a simple mechanical model from EFX

Indeed, the combined use of super-ensemble analysis and systematic perturbation with EFX represents a promising approach for development of more general models for protein mechanics. A stated goal of EFX is to enable the development of a model which reduces the complexity of protein dynamics to some series of effective variables which capture most of the essential features of protein function. While EFX can in principle yield such a model, a substantial amount of work remains in optimizing many aspects of the experiments. In the meantime, hybrid approaches such as those outlined above in §1.2 may be quite helpful as we begin to investigate the mechanical responses of different proteins in the presence of an electric field. As shown in Chapter III, analysis of super-ensembles yields patterns of conformational change in proteins, and these patterns can be decomposed to yield a few collective features, or modes. The modes identified through this analysis have several key features—they are relatively independent, and they arise from the collective action of non-trivial groups of amino acids. These modes thus serve as a simple model

for what one might expect from a comprehensive EFX experiment with relatively weak fields—if a given protein primarily moves along the modes identified through the super-ensemble analysis, then the EFX experiment should recapitulate this pattern if the perturbation is near kT . In the case where the data are in accordance, it would be wise to focus theoretical efforts on understanding how the protein architecture could give rise to such modes.

2.3 Comparison of mechanical models to patterns of coevolution

A central question throughout each of the presented studies addresses the degree to which patterns of physical coupling relate to patterns of amino acid coevolution in the PDZ protein family. This question arises from the assumption that coevolution between amino acids typically implies some sort of physical interaction, as protein sectors are found to be physically contiguous and consistent with allosteric networks within proteins. It is thus tempting to interpret protein sectors as some sort of combined mechanical mode for a protein family, with individual proteins having features which somehow relate to this general pattern.

The work discussed in this document suggests that this may be at least partially the case. In the study presented in Chapter II, higher-order mutagenesis and structural analysis reveals a pattern of structural shifts which is largely consistent with the PDZ family sector, which suggests that its instantiation in PSD-95 PDZ3 might be quite similar to the average over the family. Studies of PSD-95 PDZ3 and LNX2 PDZ2 are less consistent with the PDZ sector, however. As discussed above, both super-ensemble analysis and electric field perturbation reveal mechanical coupling and large physical changes in regions consistent with the sector, particularly near the ligand C-terminus and the α_1 helix. That being said, substantial signal is also present in regions of the proteins that do not strongly coevolve in both cases as well. Furthermore, averaging over

homologous super-ensembles reveals no single mechanical mode or combination of modes which is solely consistent with the sector, either. What can account for the discrepancy between the conformational shifts with mutagenesis and the physical perturbation data?

Perhaps the answer lies in the nature of the mutagenic perturbations employed in the first study. These mutations were made within the protein sector and effectively represent a series of perturbations along the path from the β_2 - β_3 loop to the α_1 helix, and they certainly have a different physical effect on the protein than thermal agitation or the application of electric field. In the case of the G330T mutation to PSD-95 PDZ3, the torsional properties of the β_2 - β_3 loop are altered considerably. In the case of the T₂F mutation to the CRIPT ligand, a bulky, hydrophobic group of atoms must somehow be accommodated on the surface of the protein. Multiple atoms with different chemical properties are present or absent from the structures, depending on the mutation—substantial perturbations made to local environments, in contrast to the distributed nature of the thermal or electric field perturbations throughout the protein. Furthermore, structural response which approximates the sector is only evident with higher-order mutation.

This comparison suggests that further experimentation is needed to directly test the hypothesis that some form of the sector is instantiated as a physical mode in any given protein without directly relying on mutagenesis. Essentially, one would like to deposit energy anisotropically within the protein, with some manner of spatial control. EFX is well-suited to this task—consider the following experiment, which is perhaps an imperfect physical analog to the work of Ota and Agard². Crystals of a wild-type PDZ domain such as PSD-95 PDZ3 could be prepared under low-salt conditions in complex with a special ligand. The geometry of peptide ligand binding results in the side-chain of the -1 position of the ligand (in the case of CRIPT, a serine) directly engaged with solvent. A modified form of CRIPT could be prepared with a

mutation, S₁C. The presence of a cysteine would enable labeling of the peptide with small, high-Z ions, such as lanthanides, which would be highly sensitive to the presence of an electric field. The peptide could thus be “shaken” along different dimensions with an electric field, and the structural response read out through X-ray crystallography. It is also possible that high-Z ions could be soaked into crystals (as in isomorphous replacement) and exert such effects non-covalently, although this would require a degree of luck—coupling with the protein would need to be quite strong, and there is no way to predict derivatization sites *a priori*.

It would also be interesting to consider EFX experiments on multiple members of protein families for which the binding of metals, ions, or other charged substrates (e.g., adenosine triphosphate (ATP)) is a matter of function. In the case of ion binding, the choice here is obvious—ion channels are an ideal target, as application of an electric field along the pore of a channel would almost certainly lead to the anisotropic propagation of energy throughout the protein (in fact, this is already underway with the NaK2K channel, albeit in the context of a “simpler” goal—the solution to the structure of the channel in the presence of field). In the case of charged substrates, the catalytic domain of an enzyme, cyclic AMP-dependent protein kinase (PKA; catalytic domain PKAc) would be an interesting target. The reaction cycle of PKA was recently characterized by X-ray crystallography, including several sub-2 Å structures of transition states³. These results show that structures of PKAc bound to certain transition state mimics are quite similar to the actual transition state structures, so it might be interesting to perform EFX on PKAc crystals with different transition state analogues bound.

Together, these studies might reveal general mechanisms by which energy is propagated anisotropically through structures, and they would test the hypothesis that this occurs along

pathways consistent with sectors. Studies in enzymes could also prove invaluable in investigating the conformational and energetic landscape of catalysis in unprecedented detail.

3 Wrapping up

A central question proposed at the beginning of this work related to the correspondence between protein and machine. This remains an open question—we have only begun to develop the experimental and analytical tools required to probe the material properties of proteins in atomic detail. It is not hard to imagine that, sometime in the future, this problem will begin to clarify, and, when it does, the conventional strategy of studying every possible protein individually will seem antiquated. Franz Reuleaux summarized this transition well in 1876, writing of the emergence of a general theory of machines:

In earlier times men considered every machine as a separate whole, consisting of parts peculiar to it; they missed entirely or saw but seldom the separate groups of parts which we call mechanisms. A mill was a mill, a stamp a stamp and nothing else, and thus we find the older books describing each machine separately from beginning to end.⁴

Perhaps someday our understanding of proteins will reach a similar state, wherein shared mechanism becomes clear and we no longer need to describe each type of protein separately.

4 Works cited

1. Stroud, R. M. & Fauman, E. B. Significance of structural changes in proteins: expected errors in refined protein structures. *Protein Sci* **4**, 2392–2404 (1995).
2. Ota, N. & Agard, D. A. Intramolecular signaling pathways revealed by modeling anisotropic thermal diffusion. *J Mol Biol* **351**, 345–354 (2005).
3. Das, A. *et al.* Protein Kinase A Catalytic Subunit Primed for Action: Time-Lapse Crystallography of Michaelis Complex Formation. *Structure* **23**, 2331–2340 (2015).
4. Reuleaux, F. *The Kinematics of Machinery: Outlines of a Theory of Machines*. (Macmillan and Company, 1876).

Chapter VI

Appendices

1 Code

1.1 delta_r

1.1.1 delta_r_analysis.py

A series of functions and a command line interface for comparing displacements between crystal structures as described in Chapter II.

```
#!/usr/bin/env python

from __future__ import print_function, with_statement, division, unicode_literals,
absolute_import

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by-sa/4.0/'

import os, argparse, glob, pickle
import itertools as it
import numpy as np
from Bio import PDB
from matplotlib import pyplot as plt

def load_pdb_list(ensemble, superimpose=True, supersele='all', ref=None, reflections_list=[],
dumpPDB=False, verbose=True, out_dir = None):
    """
    Creates an ensemble of Bio.PDB structures from one or more PDB files.
    Input:
        ensemble: A list of PDB files, a path to a PDB file, or a path to a directory containing
        multiple PDB files.
        superimpose: If true, output structures will be aligned to the first model of the first
        PDB file specified using Bio.PDB.Superimposer
        supersele: A string specifying which residues to use for superimposing. If '\all\', use
        all residues. Otherwise, provide comma-separated ranges x:x+n, where x and x+n are consistent
        with PDB residue indexing.
        ref: Reference model ID; if None, will use first model encountered.
        reflections_list: A list with number of unique (non-anomalous) reflections used to
        generate a corresponding structure in ensemble; if not provided, will try to find it in each
        header.
        dumpPDB: If true, will save PDB files after superposition.
        verbose: If true, say more.
    Output:
        ensemble_list: A list of one or more paths to pickles containing Bio.PDB structure
        objects.
        rms_list: If superimpose is True, will output RMS of each PDB relative to the reference
        after superposition.
    """
    def
get_atoms(biopdb_obj,supersele_list,model_id=0,refMode=False,atom_label=['N','CA','C','O']):
    """
    Get CA atoms consistent with selection list from a structure.
    """
    if not isinstance(biopdb_obj,PDB.Model.Model):
```

```

    try:
        model = biopdb_obj[model_id]
    except KeyError:
        print('Failed to get structure {} from {}'.format(model_id, biopdb_obj.id))
        raise
else:
    model = biopdb_obj
    if type(atom_label) is str:
        atom_label = [atom_label]
    atoms = []
    resis = []
    for chain in model:
        for res in chain:
            if supersele_list == ['all'] or any([res.id[1] in range(*supersele_list[i]) for i
in range(len(supersele_list))]):
                try:
                    [atoms.append(res[a]) for a in atom_label]
                    resis.append(res.id[1])
                except KeyError:
                    continue
    if refMode and atoms == []:
        raise ValueError('Cannot get atoms from {}! Check model indexing and selection
string.'.format(biopdb_obj))
    return(atoms, resis)

# Initialzie everything
print('\nLoading models...')
parser = PDB.PDBParser()
if superimpose:
    m=' and superimposing '
    if supersele == 'all':
        m += 'at all positions '
        supersele_list = ['all']
    else:
        supersele_list=[]
        for range_str in supersele.split(','):
            supersele_list.append([int(i) for i in range_str.split(':')])
            supersele_list[-1][1] += 1
        m += 'a subset of positions as specified by supersele_list {}
.'.format(supersele_list)
    else:
        m=' '
    if superimpose and ref is None:
        gotRef = False
    elif superimpose and ref is not None:
        if type(ref) is str and os.path.isfile(ref):
            print(' Will superimpose everything with first model of structure from
{}.'.format(ref))
            refStructure = parser.get_structure(os.path.basename(ref), ref)
        elif isinstance(ref, PDB.Structure.Structure):
            print(' Will superimpose everything with first model of structure from
{}.'.format(ref.id))
            refStructure = ref
        else:
            raise ValueError('Supplied reference {} is not valid!'.forma(ref))
        refModel = refStructure[0]
        refAtoms, refResis = get_atoms(refModel, supersele_list, refMode=True)
        gotRef = True
    if type(ensemble) is not list:
        if os.path.isfile(ensemble):
            ensemble = (ensemble,)
        elif os.path.isdir(ensemble):
            ensemble = glob.glob(os.path.join(ensemble, '*.pdb'))
        else:
            raise ValueError('Invalid PDB file or directory of files specified.')
    elif len(ensemble) == 1:
        if os.path.isdir(ensemble[0]):
            ensemble = glob.glob(os.path.join(ensemble[0], '*.pdb'))
    ensemble_list = []

```

```

n_reflections = None
for i,pdb in enumerate(ensemble):
    if reflections_list == [] or len(reflections_list) != len(ensemble):
        with open(pdb,'r') as f:
            for line in f:
                if line.rstrip().split(' ')[0] == 'ATOM':
                    break
                if 'NUMBER OF REFLECTIONS (NON-ANOMALOUS)' in line:
                    n_reflections = int(line.rstrip().split(' ')[-1])
                    break
            if n_reflections is None:
                raise ValueError('Could not find number of reflections in header of
{}'.format(pdb))
            else:
                n_reflections = reflections_list[i]
                pdbPath,pdbFile = os.path.split(pdb)
                pdbBase = os.path.splitext(pdbFile)[0]
                print(' Loading {}'.format(pdb))
                thisStructure = parser.get_structure(os.path.basename(pdb),pdb)
                thisStructure.xtra['n_reflections'] = n_reflections
                print(' Harvesting{}for models from from {}'.format(m,os.path.basename(pdb)))
                if superimpose and not gotRef: # If we're on the first model and a reference structure
was not supplied, create a CA atom list from it to use as reference
                    print(' Superposing everything with first model of structure from {}'.format(pdb))
                    refModel = thisStructure[0]
                    refAtoms, refResis = get_atoms(refModel, supersele_list, refMode=True)
                    gotRef = True
                if superimpose and gotRef: # If we're on a model other than the reference, align it to
the reference using PDB.Superimposer
                    rms_list = []
                    for thisModel in thisStructure:
                        theseAtoms, theseResis = get_atoms(thisModel, supersele_list)
                        if len(refAtoms) == len(theseAtoms):
                            super_imposer = PDB.Superimposer()
                            super_imposer.set_atoms(refAtoms,theseAtoms)
                            if verbose:
                                print(' Aligned model {} in {} to reference (RMS = {:.2f}
Angstroms)'.format(thisModel.id,os.path.basename(pdb),np.abs(super_imposer.rms)))
                                rms_list.append((pdb,super_imposer.rms))
                                super_imposer.apply(thisStructure[thisModel.id].get_atoms())
                        else:
                            raise ValueError('Number of CA atoms in model {} of current structure is {}
this is different from the number in the reference structure,
{}!\n\nREFERENCE:\n{}\n{}\n\nCURRENT\n{}\n{}\n\n'.format(thisModel.id, len(theseAtoms),
len(refAtoms), refModel.get_parent().id, refResis, thisStructure.id, theseResis))
                    thisStructure.xtra['super_rms'] = super_imposer.rms
                ensemble_list.append(thisStructure)
                if dumpPDB:
                    io=PDB.PDBIO()
                    io.set_structure(thisStructure)
                    pdb_out='{}_{}_'.format(os.path.splitext(pdb)[0],i)
                    if superimpose:
                        if supersele == 'all':
                            pdb_out += 'superposed-all.pdb'
                        else:
                            pdb_out += 'superposed-subset.pdb'
                    else:
                        pdb_out += 'raw.pdb'
                    if out_dir is not None:
                        try:
                            pdb_out = os.path.join(os.path.abspath(out_dir),os.path.basename(pdb_out))
                        except:
                            raise
                    print(' Saving output PDB to {}'.format(pdb_out))
                    io.save(pdb_out)
                return(ensemble_list)

def bfactor_analysis(structure_list, resi_sele='all', ignore_hydrogens=True, ignore_hetatms=True,
plot=False, text_out=False, out_dir=None):

```

```

'''
Pack isotropic B-factors from structures of structure_list into a dictionary, and calculate
occupancy-weighted mean and standard deviation for each residue. Assumes model ID is 0 in all
cases.
Input:
structure_list: A list of Bio.PDB structures.
resi_sele: Residue IDs to include in dictionary, for example, '300:315,336:348,352:390'
selects a discontinuous set of residues. Note that this does not currently support multiple
chains.
ignore_hydrogens: If True, hydrogen info will not be included in bfactor_dict.
ignore_hetatms: If True, only protein residues will be added to bfactor_dict.
plot: If True, will create plots.
text_out: If True, will create text file with per-residue mean/std of B-factors.
out_dir: A path to a directory; will put things here.
Output:
bfactor_dict: A dictionary keyed as follows:
    structure_key: A string specifying the PDB IDs for the comparison
    model_id: An integer corresponding to model index; typically 0.
    chain_id: A string corresponding to a chain in model specified by model_id;
for example, 'A'.
    resi_id: An integer corresponding to a residue in chain specified by
chain_id; for example, 344.
    atom_id: A string corresponding to an atom name in residue specified
by resi_id; for example, 'CA'. This keys a dictionary with a variety of properties related to the
calculation:
        altloc_id: A string corresponding to a given altloc; for atoms
with only one conformation, only 'A' will be present.
        'occ': Occupancy for altloc_id; if no other altlocs, will be
equal to 1.
        'b': B-factor for altloc_id.
        'weighted_avg': Average B-factor for residue resi_id.
        'weighted_std': Standard deviation of atomic B-factors for residue
resi_id.
'''
print('Analyzing isotropic B-factors across input ensemble...')
if len(structure_list) < 1:
    raise ValueError('At least one structure must be provided in structure_list!')
for i,structure in enumerate(structure_list):
    if not isinstance(structure,PDB.Structure.Structure):
        raise ValueError('Structure {}, {}, in structure_list is not a Bio.PDB
structure!'.format(i, structure))
    if type(resi_sele) is not list and resi_sele != 'all':
        resi_sele_list=[]
        for range_str in resi_sele.split(','):
            temp_range = [int(i) for i in range_str.split(':')]
            temp_range[1] += 1
            resi_sele_list.append(list(range(*temp_range)))
        resi_sele_list = [item for sublist in resi_sele_list for item in sublist]
    if type(resi_sele) is list and resi_sele[0] != 'all':
        resi_sele_list = resi_sele
    bfactor_dict = {}
    for structure in structure_list:
        if resi_sele != 'all':
            print(' Analyzing isotropic B-factors for all residues in
{}...'.format(structure.id))
        else:
            print(' Analyzing isotropic B-factors for residues {} in
{}...'.format(resi_sele,structure.id))
        bfactor_dict[structure.id] = {}
        for model in structure:
            bfactor_dict[structure.id][model.id] = {}
            for chain in model:
                bfactor_dict[structure.id][model.id][chain.id] = {}
                for resi in chain:
                    if resi.id[0] != ' ' and ignore_hetatms is True:
                        continue
                    bfactor_dict[structure.id][model.id][chain.id][resi.id[1]] = {}
                    b_factors = []
                    for atom in resi:

```

```

        if atom.element == 'H' and ignore_hydrogens is True:
            continue
        bfactor_dict[structure.id][model.id][chain.id][resi.id[1]][atom.id] = {}
        if atom.is_disordered() != 0:
            alts = [(a.altloc, a.occupancy, a.bfactor) for a in
atom.disordered_get_list()]

[bfactor_dict[structure.id][model.id][chain.id][resi.id[1]][atom.id].update({alt[0]:{'occ':alt[1]
,'b':alt[2]})} for alt in alts]
            total_occ = np.sum([alt[1] for alt in alts])
            if total_occ > 1.01 or total_occ < 0.99:
                print('    Warning: altloc occupancies for atom {} of resi {},
chain {}, structure {} sum to {}, not 1.00, skipping!'.format(atom.id, resi.id,
resi.get_parent().id, structure.id, total_occ))
                continue
            b_factors.append(np.sum([alt[1]*alt[2] for alt in alts]))
        else:

bfactor_dict[structure.id][model.id][chain.id][resi.id[1]][atom.id]['A'] =
{'occ':1.00, 'b':atom.bfactor}
            b_factors.append(atom.bfactor)
            bfactor_dict[structure.id][model.id][chain.id][resi.id[1]]['weighted_avg'] =
np.mean(b_factors)
            bfactor_dict[structure.id][model.id][chain.id][resi.id[1]]['weighted_std'] =
np.std(b_factors)
            if plot and len(bfactor_dict.keys()) > 0:
                print('    Creating figure...')
                reduced_bfactor_dict = {}
                n_plots = len(bfactor_dict.keys())
                fig, ax = plt.subplots(nrows=n_plots, ncols = 1, figsize=(20, 10*n_plots), squeeze=False)
                fig.suptitle('Per-residue isotropic B-factors', fontsize=30)
                for i, structure in enumerate(sorted(bfactor_dict.keys())):
                    chain_list = []
                    resi_idx_list = []
                    mean_b_list = []
                    std_b_list = []
                    for model in sorted(bfactor_dict[structure].keys()):
                        for chain in sorted(bfactor_dict[structure][model].keys()):
                            these_indices = list(bfactor_dict[structure][model][chain].keys())
                            resi_idx_list += these_indices
                            chain_list += len(these_indices)*chain

[mean_b_list.append(bfactor_dict[structure][model][chain][resi]['weighted_avg']) for resi in
bfactor_dict[structure][model][chain]]

[std_b_list.append(bfactor_dict[structure][model][chain][resi]['weighted_std']) for resi in
bfactor_dict[structure][model][chain]]
                temp_bfactor_list = list(zip(chain_list, resi_idx_list, mean_b_list, std_b_list))
                if resi_sele != 'all':
                    reduced_bfactor_dict[structure] = []
                    [reduced_bfactor_dict[structure].append(entry) for entry in temp_bfactor_list if
entry[1] in resi_sele_list]
                    resi_idx_list = [entry[1] for entry in reduced_bfactor_dict[structure]]
                    mean_b_list = [entry[2] for entry in reduced_bfactor_dict[structure]]
                    std_b_list = [entry[3] for entry in reduced_bfactor_dict[structure]]
                else:
                    reduced_bfactor_dict[structure] = temp_bfactor_list
                print('    Drawing plot for structure {}...'.format(structure))
                ax[i][0].bar(range(len(mean_b_list)), mean_b_list, yerr=std_b_list, error_kw={'ecolor':
'0.5'}, color='k', alpha=0.8, edgecolor='none')
                ax[i][0].plot([0, len(mean_b_list)+1], [31, 31], "r--")
                ax[i][0].plot([0, len(mean_b_list)+1], [16, 16], "b--")
                ax[i][0].set_xlim([0, len(mean_b_list)])
                #ax[i][0].set_ylim([0, max(mean_b_list)+0.2*max(mean_b_list)])
                ax[i][0].set_ylim([0, 100])
                labels = [resi_idx_list[m] for m in [int(n) for n in ax[i][0].get_xticks()[:-1]]]
                ax[i][0].set_xticklabels(labels, rotation=45)
                ax[i][0].set_title('{}'.format(structure), fontsize=15)
                ax[i][0].set_xlabel('Residue index', fontsize=15)

```

```

    ax[i][0].set_ylabel('Isotropic B-factor (\xc3^2)',fontsize=15)
if out_dir is None:
    plt.show()
elif out_dir is not None:
    if not os.path.isdir(out_dir):
        out_dir = os.path.curdir
    try:
        [plt.savefig(os.path.join(os.path.abspath(out_dir),'per-
resi_bfactor_plot.{}'.format(f)),transparent=True,format=f) for f in ('eps','png')]
    except:
        print('Failed to save B-factor figure.')
plt.close('all')
if text_out and len(bfactor_dict.keys()) > 0:
    if out_dir is None or os.path.isdir(out_dir) is False:
        out_dir = os.path.curdir
    print(' Saving per-residue B-factor data to text file...')
    try:
        with open(os.path.join(os.path.abspath(out_dir),'per-resi_bfactor_table.txt'), 'wb')
as f:
        f.write(str('#\tChain\tResidue index\tmean(isoB), Ang**2\tstd(isoB),
Ang**2\n').encode('UTF-8'))
        for structure in sorted(reduced_bfactor_dict.keys()):
            mean_mean = np.mean([entry[2] for entry in reduced_bfactor_dict[structure]])
            mean_std = np.mean([entry[3] for entry in reduced_bfactor_dict[structure]])
            f.write(str('#\t{:} mean isotropic B = {:.3f} +/- {:.3f}
Ang**2\n'.format(structure,mean_mean,mean_std)).encode('UTF-8'))
            [f.write(str('{ }\t{ }\t{:.3f}\t{:.3f}\n'.format(*entry)).encode('UTF-8'))
for entry in reduced_bfactor_dict[structure]]
    except:
        print(' Failed to save data to text file.')
print(' Done.')
return(bfactor_dict)

def calc_delta_r(structure_list, occ_weight=True, ignore_hydrogens=True, ignore_hetatms=True,
ref_structure_idx=0, verbose=True):
    """
    Calculates Delta r between two or more structures.
    Input:
        structure_list: A list of Bio.PDB structures.
        occ_weight: If True, weight atom properties by occupancy.
        ignore_hydrogens: If True, analysis will not be performed for hydrogen atoms in the
structure, and no entries for hydrogen atoms will be present in dr_dict.
        ignore_hetatms: If True, analysis will not be performed for hetatms, and no entries will
be included in dr_dict.
        ref_structure_idx: The integer-valued index of the structure in structure_list to be used
as the reference for creating the dr_dict hierarchy.
        verbose: If True, say more.
    Output:
        dr_dict: A dictionary, with the following k:v hierarchy:
            comp_key: A string specifying the PDB IDs for the comparison
            model_id: An integer corresponding to model index; typically 0.
            chain_id: A string corresponding to a chain in model specified by model_id;
for example, 'A'.
                resi_id: An integer corresponding to a residue in chain specified by
chain_id; for example, 344.
                    atom_id: An string corresponding to an atom name in residue specified
by resi_id; for example, 'CA'. This keys a dictionary with a variety of properties related to the
calculation:
                        (order * d)coord: A float64 numpy array with values equal to dx,
dy, and dz for atom_id between two structures.
                        (order * d)r: A float64 corresponding to the raw Delta r between
atom_id for two structures.
                        (order * d)bn: A float64 corresponding to the normalized
difference in B-factors between atom_id for two structures.
                        (order * d)rn: A float64 corresponding to Delta r corrected by
positional error between atom_id for two structures.
                        (order * d)coordn: A normalized float64 numpy array with values
equal to dx, dy, and dz for atom_id between two structures.

```

```

'''
def create_dr_dict(structure_list, order, ref_structure_idx=0):
    d_str = 'd' * order
    comp_key = '+'.join([s.id for s in structure_list])
    dr_dict = {comp_key: {}}
    for model in structure_list[ref_structure_idx]: # Build dictionary to store data; in the
future, eliminate nested for loops
        dr_dict[comp_key][model.id] = {}
        for chain in model:
            dr_dict[comp_key][model.id][chain.id] = {}
            for resi in chain:
                if ignore_hetatms and resi.id[0] != ' ': # Ignore hetatms if requested
                    continue
                dr_dict[comp_key][model.id][chain.id][resi.id[1]] = {}
                for atom in resi:
                    if ignore_hydrogens and atom.element == 'H': # Ignore hydrogens if
requested
                        continue
                    dr_dict[comp_key][model.id][chain.id][resi.id[1]][atom.id] =
{d_str+'coord':np.zeros(3,dtype='float64')*np.nan, d_str+'r':np.nan,
d_str+'coordn':np.zeros(3,dtype='float64')*np.nan, d_str+'bn':np.nan, 'altloc':False, 'sets':[],
'occ_unity':False, 'msg':[]}
            return(dr_dict)

def sf_pos_err(b_factor, n_atoms, n_reflections):
'''
    This function calculates the positional error from the Stroud-Fauman formula (Protein
Science, 1995, Vol 4, pp. 2392-2404), given the B-factor, number of atoms, and the number of
reflections.
'''
    def calc_epsilon(b_factor, ratio):
'''
        Values of k are those determined by Rama and [Rohit] by fitting published curves with
published formulas.
'''
        k = np.array([-0.7238, -3.317e-5, 3.6284, 0.66709, 0.0098103,
9.9735],dtype='float64')
        slope = k[0] + k[1] * np.exp(b_factor / k[2])
        intercept = k[3] + k[4] * np.exp(b_factor / k[5])
        epsilon = intercept + slope * np.exp(-2 * ratio)
        return(epsilon)

        ratio = n_atoms / n_reflections
        p_c = 10 * ((calc_epsilon(20,ratio)-calc_epsilon(10,ratio))/(calc_epsilon(30,ratio)-
calc_epsilon(20,ratio)))
        p_b = (calc_epsilon(20,ratio)-calc_epsilon(10,ratio))/(np.exp(20 / p_c) - np.exp(10 /
p_c))
        p_a = calc_epsilon(20,ratio) - p_b * np.exp(20 / p_c)
        pos_error = p_a + p_b * np.exp(b_factor / p_c)
        return(pos_error)

def recursive_difference(a):
'''
    Calculate recursive difference for a list with an even number of values.
'''
    if len(a) % 2 != 0:
        raise ValueError('Length of a must be a multiple of two!')
    a = [a[i] - a[i+1] for i in range(0,len(a),2)]
    if len(a) == 1:
        return(a[0])
    else:
        return(recursive_difference(a))

def cart2sph(x, y, z):
'''
    http://stackoverflow.com/questions/30084174/efficient-matlab-cart2sph-and-sph2cart-
functions-in-python
'''
    azimuth = np.arctan2(y,x)

```

```

elevation = np.arctan2(z,np.linalg.norm((x,y)))
r = np.linalg.norm((x, y, z))
return(azimuth, elevation, r)

def sph2cart(azimuth, elevation, r):
    """
    http://stackoverflow.com/questions/30084174/efficient-matlab-cart2sph-and-sph2cart-
    functions-in-python
    """
    x = r * np.cos(elevation) * np.cos(azimuth)
    y = r * np.cos(elevation) * np.sin(azimuth)
    z = r * np.sin(elevation)
    return(x, y, z)

def property_mean_std(d,k,strict=False):
    """
    Get mean/standard deviation of a property keyed by k in all sub-dictionaries of d.
    If strict, np.nan will be returned for mean and std.
    """
    if strict: # If we're being anal about residues being the same, we can just return np.nan
    on the first key error
        try:
            v = [d[i][k] for i in d.keys() if i != 'merge']
        except KeyError:
            return(np.nan,np.nan)
    elif not strict: # Otherwise, just take the mean of what is here
        v = []
        for i in d.keys():
            if i == 'merge':
                continue
            try:
                v.append(d[i][k])
            except KeyError:
                continue
    if len(v) > 0 and not np.isnan(v).all():
        return(np.nanmean(v,axis=0),np.nanstd(v,axis=0))
    else:
        return(np.nan,np.nan)

print('\nPerforming Delta r calculations...')
print('  Checking input...')
if len(structure_list) < 2:
    raise ValueError('At least two structures must be provided in structure_list!')
order = np.log2(len(structure_list))
if order.is_integer() and order > 0:
    order = int(order)
else:
    raise ValueError('Log base 2 of number of structures must be a positive integer!')
for i,structure in enumerate(structure_list):
    if not isinstance(structure,PDB.Structure.Structure):
        raise ValueError('Structure {}, {}, in structure_list is not a Bio.PDB
structure!'.format(i, structure))
    if False in ['n_reflections' in s.xtra.keys() for s in structure_list]:
        raise ValueError('Number of reflections must be specified as key/value pair in
structure.xtra dictionary!')

print('  Performing order {} calculations...'.format(order))
print('  Creating dictionary and getting structure info...')
dr_dict = create_dr_dict(structure_list, order, ref_structure_idx=ref_structure_idx)
comp_key = list(dr_dict.keys())[0]
n_atoms = {}
for structure in structure_list:
    if ignore_hydrogens:
        n_atoms[structure.id] = len([a for a in structure.get_atoms() if a.element != 'H'])
    else:
        n_atoms[structure.id] = len(list(structure.get_atoms()))
s_ids = [s.id for s in structure_list]

print('  Performing calculations over common atoms in {} and {}...'.format(', '.join(s.id

```

```

for s in structure_list[:-1]),structure_list[-1].id)
for ref_atom in structure_list[ref_structure_idx].get_atoms():
    if ref_atom.element == 'H' and ignore_hydrogens: # Skip hydrogens if requested
        continue
    if ref_atom.get_parent().id[0] != ' ' and ignore_hetatms: # Skip hetatms if requested
        continue
    atom_id = ref_atom.id
    resi_id = ref_atom.get_parent().id[1]
    chain_id = ref_atom.get_parent().get_parent().id
    model_id = ref_atom.get_parent().get_parent().get_parent().id
    missing = False
    for i,this_structure in enumerate(structure_list): # Make sure this atom exists in all
non-reference structures
        if i == ref_structure_idx:
            continue
        try:
            assert(this_structure[model_id][chain_id][resi_id][atom_id].id)
        except KeyError:
            missing = True
            msg = ' Warning: model {}, chain {}, resi {}, atom {} found in structure {} but
not structure {}, skipping this calculation!'.format(model_id, chain_id, resi_id, atom_id,
structure_list[ref_structure_idx].id, this_structure.id)
            if verbose:
                print(' '+msg)
            dr_dict[comp_key][model_id][chain_id][resi_id][atom_id]['msg'].append(msg)
            continue
    if missing: # Don't perform any calculations if any structures were missing an atom in
the reference.
        continue
    # Get coordinates, bfactors, errors...
    atom_dict = {}
    for strct in structure_list:
        atom_dict[strct.id] = {}
        this_atom = strct[model_id][chain_id][resi_id][atom_id]
        if this_atom.is_disordered() > 0 and occ_weight:
            dr_dict[comp_key][model_id][chain_id][resi_id][atom_id]['altloc'] = True
            occ_sum = np.sum([this_atom.child_dict[k].get_occupancy() for k in
this_atom.child_dict.keys()])
            if occ_sum < 0.999: # Check if occupancies sum to unity; if not, warn the user
                if verbose:
                    print(' Warning: altloc occupancies for atom {} of model {}, chain
{}, resi {}, structure {} do not sum to 1.0!'.format(atom_id,model_id, chain_id, resi_id,
strct.id))
            else:
                dr_dict[comp_key][model_id][chain_id][resi_id][atom_id]['occ_unity'] = True
                atom_data = np.array([np.hstack((v.get_coord(), v.get_bfactor(),
v.get_occupancy(), sf_pos_err(v.get_bfactor(), n_atoms[strct.id], strct.xtra['n_reflections'])))
for v in this_atom.child_dict.values()]) # Create an array with each row corresponding to (x, y,
z, b, occ) of a given altloc
                atom_dict[strct.id]['coords'] = np.zeros(3, dtype='float64')
                for i in range(3): # Calculate occupancy-weighted atomic coordinates
                    atom_dict[strct.id]['coords'][i] = np.sum(atom_data[:,i] * atom_data[:,4]) /
occ_sum
                atom_dict[strct.id]['bfactor'] = np.sum(atom_data[:,3] * atom_data[:,4]) /
occ_sum
                atom_dict[strct.id]['pos_err'] = np.sqrt(np.sum(np.square(atom_data[:,5]) *
atom_data[:,4])) / occ_sum
            else:
                if this_atom.get_occupancy() < 0.999 or this_atom.get_occupancy() > 1.001:
                    if verbose:
                        print(' Warning: occupancy for atom {} of model {}, chain {}, resi
{}, structure {} is not 1.0!'.format(atom_id,model_id, chain_id, resi_id, strct.id))
                else:
                    dr_dict[comp_key][model_id][chain_id][resi_id][atom_id]['occ_unity'] = True
                    atom_dict[strct.id]['coords'] = this_atom.get_coord()
                    atom_dict[strct.id]['bfactor'] = this_atom.get_bfactor()
                    atom_dict[strct.id]['pos_err'] = sf_pos_err(this_atom.get_bfactor(),
n_atoms[strct.id], strct.xtra['n_reflections'])
                    dr_dict[comp_key][model_id][chain_id][resi_id][atom_id]['sets'].append(strct.id)

```

```

# Calculate all values with info pulled from relevant structures
dcoord = recursive_difference([atom_dict[s]['coords'] for s in s_ids]) # Note that
structure_list, and thus the ids of those structures, must be in the correct order!
dr = np.linalg.norm(dcoord)
b = np.array([atom_dict[k]['bfactor'] for k in atom_dict.keys()])
db_num = b.copy()
db_num[1:] *= -1
db_num = np.sum(db_num)
db_denom = np.sum(b)
dbn = db_num / db_denom
drn = dr / np.linalg.norm(np.array([atom_dict[k]['pos_err'] for k in atom_dict.keys()]))
azimuth, elevation, r = cart2sph(dcoord[0],dcoord[1],dcoord[2])
dcoordn = np.array(sph2cart(azimuth, elevation, drn),dtype='float64')
# Update dr_dict
d_str = order*'d'
dr_dict[comp_key][model_id][chain_id][resi_id][atom_id][d_str+'coord'] = dcoord
dr_dict[comp_key][model_id][chain_id][resi_id][atom_id][d_str+'r'] = dr
dr_dict[comp_key][model_id][chain_id][resi_id][atom_id][d_str+'bn'] = dbn
dr_dict[comp_key][model_id][chain_id][resi_id][atom_id][d_str+'rn'] = drn
dr_dict[comp_key][model_id][chain_id][resi_id][atom_id][d_str+'coordn'] = dcoordn
print('    Calculating per-residue statistics...')
for model_id in dr_dict[comp_key].keys(): # Create a new atom called 'merge' with statistics
over all atoms in the residue
    for chain_id in dr_dict[comp_key][model_id].keys():
        for resi_id in dr_dict[comp_key][model_id][chain_id].keys():
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'] = {}
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'][d_str+'coord'] =
property_mean_std(dr_dict[comp_key][model_id][chain_id][resi_id], d_str+'coord')
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'][d_str+'r'] =
property_mean_std(dr_dict[comp_key][model_id][chain_id][resi_id], d_str+'r')
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'][d_str+'bn'] =
property_mean_std(dr_dict[comp_key][model_id][chain_id][resi_id], d_str+'bn')
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'][d_str+'rn'] =
property_mean_std(dr_dict[comp_key][model_id][chain_id][resi_id], d_str+'rn')
            dr_dict[comp_key][model_id][chain_id][resi_id]['merge'][d_str+'coordn'] =
property_mean_std(dr_dict[comp_key][model_id][chain_id][resi_id], d_str+'coordn')
            print('Delta r calculations for {} complete.'.format(comp_key))
            return(dr_dict)

def
drn_analysis(dr_dict,max_order,drn_ecdf_cutoff,drsele='all',plot=False,text_out=False,out_dir=None):
    """
    Calculate mean/std per residue for normalized delta r.
    """
    print('\nBuilding Delta r distributions and calculating ECDF-based cutoffs...')
    if drsele == 'all':
        drsele_list = ['all']
    else:
        print('    Cropping Delta r analysis to residues included by selection string
{}...'.format(drsele))
        drsele_list=[]
        for range_str in drsele.split(','):
            if ':' in range_str:
                temp_range = [int(i) for i in range_str.split(':')]
                temp_range[1] += 1
                drsele_list.append(list(range(*temp_range)))
            else:
                drsele_list.append([int(range_str)])
        drsele_list = [item for sublist in drsele_list for item in sublist]
    resi_idx = {}
    mean = {}
    std = {}
    mean_sort_drn = {}
    drn_ecdf = {}
    drn_cutoff = {}
    for o in range(1,max_order+1):
        dr_key = (o*'d')+'r'

```

```

dbn_key = (o*'d')+'bn'
drn_key = (o*'d')+'rn'
resi_idx[o] = {}
mean[o] = {}
std[o] = {}
mean_sort_drn[o] = {}
drn_ecdf[o] = {}
drn_cutoff[o] = {}
for i,k in enumerate(dr_dict[o].keys()):
    resi_idx[o].update({k:[]})
    mean[o].update({k:{dr_key:[], dbn_key:[], drn_key:[]}})
    std[o].update({k:{dr_key:[], dbn_key:[], drn_key:[]}})
    for model_id in sorted(list(dr_dict[o][k].keys())):
        for chain_id in sorted(list(dr_dict[o][k][model_id].keys())):
            resi_idx[o][k] += [chain_id+str(r) for r in
dr_dict[o][k][model_id][chain_id].keys() if ('all' in drsele_list or r in drsele_list)]
            for dtype in (dr_key, dbn_key, drn_key):
                mean[o][k][dtype] +=
[dr_dict[o][k][model_id][chain_id][r]['merge'][dtype][0] for r in
dr_dict[o][k][model_id][chain_id].keys() if ('all' in drsele_list or r in drsele_list)]
                std[o][k][dtype] +=
[dr_dict[o][k][model_id][chain_id][r]['merge'][dtype][1] for r in
dr_dict[o][k][model_id][chain_id].keys() if ('all' in drsele_list or r in drsele_list)]
            mean_sort_drn[o][k] = np.sort(mean[o][k][drn_key])
            drn_ecdf[o][k] = np.arange(len(mean_sort_drn[o][k]))/float(len(mean_sort_drn[o][k]))
            drn_cutoff_idx = (np.abs(drn_ecdf[o][k]-(1-drn_ecdf_cutoff))).argmin()
            drn_cutoff[o][k] =
(drn_ecdf[o][k][drn_cutoff_idx],mean_sort_drn[o][k][drn_cutoff_idx])
            print('    Order {}, {}: {:.2f} at
{:.2f}%'.format(o,k,drn_cutoff[o][k][1],drn_cutoff[o][k][0]*100))
            if plot:
                n_plots = []
                [n_plots.append([order,len(list(dr_dict[order].keys()))]) for order in
range(1,max_order+1)]
                print('\nCreating dr and dbn figures...')
                fig_dict_raw = {}
                for o,n in n_plots:
                    print('    Creating dr/dbn figure for order {}...'.format(o))
                    dr_key = (o*'d')+'r'
                    dbn_key = (o*'d')+'bn'
                    fig_dict_raw[o] = {}
                    fig_dict_raw[o]['fig'],fig_dict_raw[o]['ax'] = plt.subplots(nrows=n,ncols =
2,figsize=(40,10*n),squeeze=False)
                    fig_dict_raw[o]['fig'].suptitle('Order {}'.format(o),fontsize=30)
                    for i,k in enumerate(sorted(dr_dict[o].keys())):
                        print('        Plotting {}...'.format(k))
                        # Plot raw dr
                    fig_dict_raw[o]['ax'][i][0].bar(range(len(mean[o][k][dr_key])),mean[o][k][dr_key],yerr=std[o][k][
dr_key],error_kw={'ecolor': '0.5'},color='k',alpha=0.8,edgecolor='none') #
                    fig_dict_raw[o]['ax'][i][0].set_xlim([0,len(mean[o][k][dr_key])])
                    fig_dict_raw[o]['ax'][i][0].set_ylim([0,max(mean[o][k][dr_key])+0.2*max(mean[o][k][dr_key])])
                    labels = [resi_idx[o][k][m] for m in [int(n) for n in
fig_dict_raw[o]['ax'][i][0].get_xticks()[:-1]]]
                    fig_dict_raw[o]['ax'][i][0].set_xticklabels(labels,rotation=45)
                    fig_dict_raw[o]['ax'][i][0].set_title('{}: {} r'.format(k,'Delta
'*o),fontsize=15)
                    fig_dict_raw[o]['ax'][i][0].set_xlabel('Residue index',fontsize=15)
                    fig_dict_raw[o]['ax'][i][0].set_ylabel('Raw {} r (\xc3)'.format('Delta
'*o),fontsize=15)
                    # Plot dbn
                    fig_dict_raw[o]['ax'][i][1].bar(range(len(mean[o][k][dbn_key])),mean[o][k][dbn_key],yerr=std[o][k][
[dbn_key],error_kw={'ecolor': '0.5'},color='k',alpha=0.8,edgecolor='none')
                    fig_dict_raw[o]['ax'][i][1].set_xlim([0,len(mean[o][k][dbn_key])])
                    labels = [resi_idx[o][k][m] for m in [int(n) for n in fig_dict_raw[o]['ax'][i][1].get_xticks()[:-
1]]]
                    fig_dict_raw[o]['ax'][i][1].set_xticklabels(labels,rotation=45)

```

```

fig_dict_raw[o]['ax'][i][1].set_title('{}: {} r'.format(k, 'Delta
'*o), fontsize=15)
fig_dict_raw[o]['ax'][i][1].set_xlabel('Residue index', fontsize=15)
fig_dict_raw[o]['ax'][i][1].set_ylabel('{} bn'.format('Delta '*o), fontsize=15)
if out_dir is None:
    plt.show()
elif out_dir is not None:
    try:
        [plt.savefig(os.path.join(out_dir, 'per-resi_dr-dbn_order-
{}_plot.{}'.format(o, f)), transparent=True, format=f) for f in ('png', 'eps')]
    except:
        print('Failed to save dr/dbn figure {}-{}; moving on...'.format(o, i))
plt.close('all')
print('\nCreating drn figures...')
fig_dict_drn = {}
for o, n in n_plots:
    print('    Creating drn figure for order {}...'.format(o))
    drn_key = (o*'d')+'rn'
    fig_dict_drn[o] = {}
    fig_dict_drn[o]['fig'], fig_dict_drn[o]['ax'] = plt.subplots(nrows=n, ncols =
2, figsize=(40, 10*n), squeeze=False)
    fig_dict_drn[o]['fig'].suptitle('Order {}'.format(o), fontsize=30)
    for i, k in enumerate(sorted(dr_dict[o].keys())):
        print('    Plotting {}...'.format(k))
        # Plot ECDF
        fig_dict_drn[o]['ax'][i][0].plot(mean_sort_drn[o][k], drn_ecdf[o][k], '-
', color='0.5')
fig_dict_drn[o]['ax'][i][0].plot(mean_sort_drn[o][k], drn_ecdf[o][k], 'o', color='k')
fig_dict_drn[o]['ax'][i][0].plot([0, max(mean_sort_drn[o][k]),
[drn_cutoff[o][k][0], drn_cutoff[o][k][0]], "r--")
fig_dict_drn[o]['ax'][i][0].plot([drn_cutoff[o][k][1], drn_cutoff[o][k][1]], [0,
1], "r--")
fig_dict_drn[o]['ax'][i][0].set_xlim([0, max(mean_sort_drn[o][k])])
fig_dict_drn[o]['ax'][i][0].set_ylim([0, 1])
fig_dict_drn[o]['ax'][i][0].set_title('{}: {}r ECDF'.format(k, 'Delta
'*o), fontsize=15)
fig_dict_drn[o]['ax'][i][0].set_xlabel('{}r '.format('Delta '*o), fontsize=15)
fig_dict_drn[o]['ax'][i][0].set_ylabel('ECDF({}r)'.format('Delta
'*o), fontsize=15)
# Plot Delta r
fig_dict_drn[o]['ax'][i][1].bar(range(len(mean[o][k][drn_key])), mean[o][k][drn_key], color='k', alp
ha=0.8, edgecolor='none') #, yerr=std[o][k][drn_key]
fig_dict_drn[o]['ax'][i][1].plot([0, len(mean[o][k][drn_key]),
[drn_cutoff[o][k][1], drn_cutoff[o][k][1]], "r--")
fig_dict_drn[o]['ax'][i][1].set_xlim([0, len(mean[o][k][drn_key])])
fig_dict_drn[o]['ax'][i][1].set_ylim([0, max(mean[o][k][drn_key])+0.2*max(mean[o][k][drn_key])])
labels = [resi_idx[o][k][m] for m in [int(n) for n in
fig_dict_drn[o]['ax'][i][1].get_xticks()[:-1]]]
fig_dict_drn[o]['ax'][i][1].set_xticklabels(labels, rotation=45)
fig_dict_drn[o]['ax'][i][1].set_title('{}: {} r'.format(k, 'Delta
'*o), fontsize=15)
fig_dict_drn[o]['ax'][i][1].set_xlabel('Residue index', fontsize=15)
fig_dict_drn[o]['ax'][i][1].set_ylabel('{} r '.format('Delta '*o), fontsize=15)
if out_dir is None:
    plt.show()
elif out_dir is not None:
    try:
        [plt.savefig(os.path.join(out_dir, 'per-resi_drn_order-{}_plot.{}'.format(o,
f)), transparent=True, format=f) for f in ('eps', 'png')]
    except:
        print('Failed to save drn figure {}-{}; moving on...'.format(o, i))
plt.close('all')

if text_out:
    print('\nSaving drn data to text file(s)...')
    for order in mean.keys():

```

```

dr_key = (order*'d')+'r'
dbn_key = (order*'d')+'bn'
drn_key = (order*'d')+'rn'
with open(os.path.join(out_dir, 'per-resi_drn_order-{}_table.txt'.format(order)),
'wb') as f:
    f.write(str('#\tChain\tResidue
index\tmean(dr)\tstd(dr)\tmean(db)\tstd(db)\tmean(drn)\tstd(drn)\n').encode('UTF-8'))
    for comp in sorted(mean[order].keys()):
        f.write(str('#\t{}: drn cutoff {:.2f} at
{:.2f}%\n'.format(comp,drn_cutoff[order][comp][1],drn_cutoff[order][comp][0]*100)).encode('UTF-
8'))
        for i,this_resi in enumerate(resi_idx[order][comp]):
            f.write(str('{}\t{}\t{:.4f}\t{:.4f}\t{:.4f}\t{:.4f}\t{:.4f}\t{:.4f}\n'.format(this_resi[0],
this_resi[1:],mean[order][comp][dr_key][i],std[order][comp][dr_key][i],mean[order][comp][dbn_key]
[i],std[order][comp][dbn_key][i],mean[order][comp][drn_key][i],std[order][comp][drn_key][i])).enc
ode('UTF-8'))
    return(resi_idx, mean, std, mean_sort_drn, drn_ecdf, drn_cutoff)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='A set of functions for performing delta r
analysis on a set of two or more protein structures.')
    parser.add_argument('ensemble', nargs='*',type=str, default=os.getcwd(), help='A list of PDB
files or a directory containing PDB files; must be in proper order for Delta r calculation!')
    parser.add_argument('-s','--supersele',type=str, default='none',help='A string specifying
which Calphas to use in superposition (e.g., \'297:315,336:415\'). If \'all\', all Calphas will
be used; if none, alignment will not be performed.')
    parser.add_argument('-d','--drsele',type=str, default='none',help='A string specifying which
residues to consider for Delta r analysis after calculation (e.g., \'297:315,336:415\'). If
\'all\', all Calphas will be used.')
    parser.add_argument('-c','--drn_ecdf_cutoff',type=float,default=0.2,help='Top fraction to
consider significant using ECDF for drn.')
    parser.add_argument('-r','--reflections_list',nargs='*',type=int, default=[],help='A list of
integer-valued reflection counts in the same order as input PDB files.')
    parser.add_argument('-o','--out_dir',type=str, default=os.getcwd(), help='A directory into
which to dump pickles, plots, etc.')
    #parser.add_argument('-l','--orders',nargs='*',type=int,default=[],help='Order(s) to
calculate; defaults to all.')
    parser.add_argument('-p','--plot',action='store_true',help='Plot results.')
    parser.add_argument('-t','--text_out',action='store_true',help='Write results to text file.')
    args = parser.parse_args()

    print('\n== delta_r_analysis.py ==')

    # Check inputs
    if args.supersele != 'none':
        superimpose = True
    else:
        superimpose = False
    if args.out_dir.lower() == 'none':
        args.out_dir = None
    if args.drsele == 'none':
        args.drsele = 'all'

    # Generate the ensemble, dump it to pickle
    ensemble_list = load_pdb_list(args.ensemble, superimpose=superimpose,
supersele=args.supersele, ref=None, reflections_list=args.reflections_list, dumpPDB=True,
out_dir=args.out_dir)

    # Perform analysis of isotropic B-factors for each structure
    bfactor_dict = bfactor_analysis(ensemble_list, resi_sele=args.drsele, ignore_hydrogens=True,
ignore_hetatms=True, plot=True, text_out=True, out_dir=args.out_dir)

    # Perform all delta r calculations, dump output to pickle
    max_order = np.log2(len(ensemble_list))
    if max_order >= 1:
        max_order = int(max_order)
    else:
        raise ValueError('Must perform at least first order calculation!')

```

```

dr_dict = {}
for order in range(1,max_order+1):
    dr_dict[order] = {}
    [dr_dict[order].update(calc_delta_r(structures)) for structures in
it.combinations(ensemble_list,2**order)] # Parallelize this
    with open(os.path.join(args.out_dir, 'dr_dict.pkl'), 'wb') as p:
        pickle.dump(dr_dict, p)

    # Get drn from dr_dict and calculate per residue statistics... calculate cutoff based on
    ECDF, plot if requested, output text if requested
    resi_idx, mean, std, mean_sort_drn, drn_ecdf, cutoff_drn = drn_analysis(dr_dict, max_order,
args.drn_ecdf_cutoff, args.drsele, plot=args.plot, text_out=args.text_out, out_dir=args.out_dir)

    print('\nFinished!\n')

```

1.2 multiconf_tools

1.2.1 prep_pdb.py

These functions and command line interface are used to prepare single-conformer PDB files with a variety of different properties.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, argparse
from random import choice
from uuid import uuid4

try:
    import Bio.PDB
    from Bio.PDB.PDBParser import PDBParser
    from Bio.PDB import PDBIO
    from Bio.PDB import Select
except:
    print('Module biopython is required!')
    raise

try:
    from itertools import zip_longest
except:
    from itertools import izip_longest as zip_longest

def split_pdb_occupancy(pdbIn,mode,pdbOut=None,strict=True):
    """
    Takes a PDB file and creates a new one in which each residue has a single conformation with
    coordinates derived from the altloc with maximum or minimum occupancy.
    Altloc IDs are set to ' ' and all occupancies in the output file are set to 1.00.
    Input:
        pdbIn: Path to a PDB file with multiple conformations.
        pdbOut: Optional path and file name for output PDB file; if not specified, will default
        to pdbIn_mode.pdb in the same directory in which pdbIn is found.
        mode: Mode in which to operate; if 'max', will output PDB file where all residues were
        the ones with the highest occupancy in pdbIn. If 'min', pdbOut will contain all residues with
        minimum occupancy.
        strict: If keep_occ selector class encounters an error and strict is True, an error will
        be thrown. If strict is False, the atom will be unselected and execution will continue.
    """

```

```

Output:
... pdbOut: Path to PDB file written.
...

class keep_min_max_occ(Select):
    """
    A Bio.PDB selection class which keeps the alt. loc. with either minimum or maximum
    occupancy at a given position.
    """
    def __init__(self,mode='max',strict=True):
        self.mode=mode
        self.strict=strict
    def accept_atom(self, atom):
        try:
            if atom.is_disordered() > 0:
                parent = atom.get_parent()
                testAtom = parent[parent.child_list[0].id]
                altlocs=testAtom.disordered_get_id_list()
                occs = []
                for alt in altlocs:
                    occs.append((testAtom.disordered_get(alt).occupancy,alt))
                keep=sorted(occs)[-1][1]
                if atom.get_altloc() == keep:
                    if self.mode=='max':
                        return(1)
                    elif self.mode=='min':
                        return(0)
                    elif self.mode=='random':
                        return
                else:
                    if self.mode=='max':
                        return(0)
                    elif self.mode=='min':
                        return(1)
            else:
                return(1)
        except:
            print('Error parsing atom {} in residue {} of chain {}, check altloc IDs in input
PDB file.'.format(atom.id,atom.get_parent().id[1],atom.get_parent().get_parent().id),end=' ')
            if self.strict:
                print('Exiting.')
                raise
            else:
                print('Ignoring.')
                return(0)

# Function
path,file = os.path.split(pdbIn)
prefix=os.path.splitext(file)[0]
if pdbOut == None:
    pdbOut = os.path.join(path,prefix+'_'+mode+'occ.pdb')
print('Creating PDB file '+os.path.split(pdbOut)[1]+'', in which only altlocs with
'+mode+'imum occupancy are preserved...')
parser = PDBParser()
structure = parser.get_structure(prefix,pdbIn)
io=PDBIO()
io.set_structure(structure)
mode=mode.lower().strip()
if mode != 'min' and mode != 'max':
    print('Invalid mode specified, defaulting to \'max\'.')
    mode = 'max'
if mode=='max':
    io.save(pdbOut+'_temp', select=keep_min_max_occ(mode='max',strict=strict))
elif mode == 'min':
    io.save(pdbOut+'_temp', select=keep_min_max_occ(mode='min',strict=strict))
else:
    print('?')
with open(pdbIn,'r') as f:

```

```

with open(pdbOut+'_temp','r') as g:
    with open(pdbOut,'w') as h:
        for l in f:
            if ('REMARK' in l.split()[0] or 'CRYST' in l.split()[0] or 'SCALE' in
l.split()[0]):
                h.write(l)
            elif 'ATOM' in l.split()[0]:
                break
            for l in g:
                if 'ATOM' in l.split()[0] or 'HETATM' in l.split()[0]:
                    lmod = list(l)
                    lmod[16] = ' '
                    lmod[56:60] = '1.00'
                    lmod[72] = ' '
                    lmod=''.join(lmod)
                    h.write(lmod)
                else:
                    h.write(l)
os.remove(pdbOut+'_temp')
print('Done.\n')
return(pdbOut)

def
split_pdb_random_altconf(pdbIn,localize=True,pdbOut=None,strict=True,occupancy_bias=True,verbose=
False):
    """
    Takes a PDB file and creates a new one in which each residue has a randomly chosen
conformation.
    Input:
        pdbIn: Path to a PDB file with multiple conformations.
        pdbOut: Optional path and file name for output PDB file; if not specified, will default
to pdbIn_mode.pdb in the same directory in which pdbIn is found.
        mode: Mode in which to operate; if 'max', will output PDB file where all residues were
the ones with the highest occupancy in pdbIn. If 'min', pdbOut will contain all residues with
minimum occupancy.
        strict: If keep_occ selector class encounters an error and strict is True, an error will
be thrown. If strict is False, the atom will be unselected and execution will continue.
    Output:
        pdbOut: Path to PDB file written.
    """

class keep_alt_loc_posns(Select):
    def __init__(self,posnsAltlocList,strict=True):
        if not posnsAltlocList == [] and type(posnsAltlocList) == list:
            self.posnsAltlocList=posnsAltlocList
            self.altResiIDs = list(zip(*posnsAltlocList))[1]
            self.strict=strict
        else:
            raise ValueError('posnsAltlocList must be a list with at least one element! Does
input model have altlocs?')
    def accept_atom(self, atom):
        try:
            thisID = atom.get_parent().id[1]
            thisChain = atom.get_parent().get_parent().id
            if thisID in self.altResiIDs and atom.is_disordered() != 0: # First pass; are we
supposed to mess with it, is it disordered
                keepAltloc = [x for i,x in enumerate(self.posnsAltlocList) if
self.posnsAltlocList[i][1] == thisID and self.posnsAltlocList[i][0] == thisChain] # Get altloc ID
for residue that we're supposed to keep
                if len(keepAltloc) == 1:
                    keepAltloc = keepAltloc[0]
                elif len(keepAltloc) == 0:
                    raise ValueError('There should be at least one entry for {} in
posnsAltlocList, check and try again.'.format(thisID))
                else:
                    raise ValueError('Ambiguous residue/chain indexing found for residue ID
{} of chain {}: {}'.format(thisID, thisChain, keepAltloc))
                if atom.get_altloc() == keepAltloc[2]: # if the altloc is correct
                    return(1)

```

```

        else:
            return(0)
            elif thisID in self.altResiIDs and atom.is_disordered() == 0: # Get rid of any
spurious, non-disordered atoms in a residue expected to be ordered
                keepAltloc = [x for i,x in enumerate(self.posnsAltlocList) if
self.posnsAltlocList[i][1] == thisID and self.posnsAltlocList[i][0] == thisChain]
                if len(keepAltloc) > 0: # If there is an entry in keepAltloc for this
chain/residue but it is not disordered, do not return the atom
                    return(0)
                else: # If there is no entry for this residue/chain, it's probably a cross-
chain conflict and we want to keep this residue.
                    return(1)
            elif thisID not in self.altResiIDs and atom.is_disordered() != 0: # Get rid of
any spurious disordered atoms in otherwise ordered residues
                return(0)
            else:
                return(1)
    except:
        print('Error parsing atom {} in resi {} of chain {}, check altloc IDs in lookup
list against PDB
file.'.format(atom.id,atom.get_parent().id[1],atom.get_parent().get_parent().id),end=' ')
        if self.strict:
            print('Exiting.')
            raise
        else:
            print('Ignoring.')
            return(0)

    path,file = os.path.split(pdbIn)
    prefix=os.path.splitext(file)[0]
    if pdbOut is None:
        if localize:
            pdbOut = os.path.join(path,prefix+'_rand-local_'+str(uuid4()).split('-')[0]+'pdb')
            print('\nCreating PDB file {}, in which alt. conf. segments are randomly
chosen...\n'.format(os.path.split(pdbOut)[1]))
        else:
            pdbOut = os.path.join(path,prefix+'_rand-full_'+str(uuid4()).split('-')[0]+'pdb')
            print('\nCreating PDB file {}, in which all alt. confs. are randomly
chosen...\n'.format(os.path.split(pdbOut)[1]))
            elif os.path.splitext(pdbOut)[1] == '':
                if localize:
                    pdbOut = os.path.join(os.path.splitext(pdbOut)[0],prefix+'_rand-
local_'+str(uuid4()).split('-')[0]+'pdb')
                    print('\nCreating PDB file {}, in which alt. conf. segments are randomly
chosen...\n'.format(os.path.split(pdbOut)[1]))
                else:
                    pdbOut = os.path.join(os.path.splitext(pdbOut)[0],prefix+'_rand-
full_'+str(uuid4()).split('-')[0]+'pdb')
                    print('\nCreating PDB file {}, in which all alt. confs. are randomly
chosen...\n'.format(os.path.split(pdbOut)[1]))
            parser = PDBParser()
            structure = parser.get_structure(prefix,pdbIn)
            io=PDBIO()
            io.set_structure(structure)
            resiList = [] # Resilist is a list of [resiID, altlocID] entries that tell the Bio.PDB
selection class which altlocID to keep for resiID
            for chain in structure[0]:
                for resi in chain: # First, build random list of alt. confs. to keep at each position
                    if resi.is_disordered(): # Only consider populate list with disordered residues
                        dCount = 0
                        for a in resi: # make sure all atoms are disordered; usually this is only a
problem when a residue without alt-confs has a lingering hydrogen with alt conf labels.
                            if a.is_disordered():
                                dCount+=1
                        if dCount != len(resi.child_list):
                            if verbose:
                                print('Residue {} is not completely disordered as only {} of {} atoms
have altlocs; will not pick a conformation here.'.format(resi.id[1],dCount,len(resi.child_list)))
                            continue

```

```

altIDOccList = [] # Set will keep only unique IDs for this residue
for atom in resi:
    try:
        alts = atom.disordered_get_id_list()
        [altIDOccList.append((a,atom.disordered_get(a).occupancy)) for a in alts]
# Add alt IDs in atom to set
    except:
        if verbose:
            print('Failed to get alt. IDs at disordered atom {} in residue {},
skipping.'.format(atom.id,resi.id[1]))
        continue # If it fails, skip to next atom
altOccList_unique = list(set([i[1] for i in altIDOccList]))
altIDList_unique = list(set([i[0] for i in altIDOccList]))
if len(altOccList_unique) != len(altIDList_unique):
    nAltID = len(altIDList_unique)
    nAltOcc = len(altOccList_unique)
    if nAltID < nAltOcc:
        if verbose:
            print('Fewer unique IDs {} found relative to occupancies {} for
residue {}, skipping.'.format(altIDList_unique,altOccList_unique,resi.id[1]))
        continue # If it fails, skip to next resi
    else: # If there are more IDs than occupancies, make sure that some
combination of occupancies adds to one.
        sub = altIDOccList[:nAltID]
        sub_occ_sum = 0
        for s in sub:
            sub_occ_sum += s[1]
        if sub_occ_sum != 1.0:
            if verbose:
                print('Found more alt. IDs {} than occupancies {} for residue {}
and occupancies do not sum to unity,
skipping.'.format(altIDList_unique,altOccList_unique,resi.id[1]))
            continue # If it fails, skip to next resi

thisID = resi.id[1]
thisChain = chain.id
if resiliList != [] and localize==True:
    lastID = resiliList[-1][1]
    lastChain = resiliList[-1][0]
    lastAltloc = resiliList[-1][2]
    if lastChain == thisChain and lastID == thisID - 1 and lastAltloc in [i for i
in a.disordered_get_id_list() for a in resi]: # If the previous residue and this residue share an
altloc, we'll use the previously chosen altloc again here
        resiliList.append([chain.id,thisID,lastAltloc])
        if verbose:
            print('Found altloc(s) at residue {} of chain {}; continuing previous
chain with altloc ID {}'.format(resiliList[-1][1],resiliList[-1][0],resiliList[-1][2]))
        continue # we've got what we want, skip to next iteration

if occupancy_bias:
    altIDOcc_dict = {}
    for i,pair in enumerate(altIDOccList): # Check to make sure occupancies are
consistent with IDs... we expect all A confs to have one occupancy, all B confs. to have another.
        if pair[0] not in altIDOcc_dict.keys():
            altIDOcc_dict[pair[0]] = altIDOccList[i][1]
        elif altIDOcc_dict[pair[0]] != altIDOccList[i][1]:
            if verbose:
                print('Residue {} has inconsistent occupancies for alt. ID {});
will not pick a conformation here.'.format(resi.id[1],pair[0]))
            continue
    conf_list = [] # Now, use occupancies to create a 100 element list with
proportional representation for each conf.
    for k,v in altIDOcc_dict.items(): # Add proportional number of elements to
conf_list

        conf_list += [k] * int(100 * v)
    if len(conf_list) not in (99,100,101):
        if verbose:

```

```

        print('Occupancies {} for residue {} do not sum to 1,
skipping.'.format(altOccList_unique, resi.id[1]))
        continue # If it fails, skip to next atom
        resiList.append([chain.id, thisID, choice(conf_list)])
    else:
        resiList.append([chain.id, thisID, choice(altIDList_unique)]) # Randomly choose
one altloc ID from the list and return list of [residue number, altloc to keep] elements
        if verbose:
            print('Found altloc(s) at residue {} of chain {}; randomly chose to preserve
altloc ID {}'.format(resiList[-1][1], resiList[-1][0], resiList[-1][2]))
            #print('\n{}\n'.format(resiList))
        io.save(pdbOut+'_temp', select=keep_alt_loc_posns(posnsAltlocList=resiList, strict=strict))
        with open(pdbIn, 'r') as f:
            with open(pdbOut+'_temp', 'r') as g:
                with open(pdbOut, 'w') as h:
                    for l in f:
                        if ('REMARK' in l.split()[0] or 'CRYST' in l.split()[0] or 'SCALE' in
l.split()[0]):
                            h.write(l)
                        elif 'ATOM' in l.split()[0]:
                            break
                    for l in g:
                        if 'ATOM' in l.split()[0] or 'HETATM' in l.split()[0]:
                            lmod = list(l)
                            lmod[16] = ' '
                            lmod[56:60] = '1.00'
                            lmod[72] = ' '
                            lmod=''.join(lmod)
                            h.write(lmod)
                        else:
                            h.write(l)
        os.remove(pdbOut+'_temp')
        print('Done.\n')
        return(resiList, pdbOut)

if __name__ == "__main__":
    '''
    ...
    parser = argparse.ArgumentParser(description='a collection of tools for modifying PDB files,
with an emphasis on wrangling altlocs.')
    parser.add_argument('pdbIn', type=str, help='path for the PDB file to be processed.')
    parser.add_argument('mode', choices=['minocc', 'maxocc', 'random-full', 'random-
local'], type=str, help='mode describing the type of operation to perform on specified PDB file.
Option \'minocc\' returns a PDB file with all altlocs with minimum occupancy only, while
\'maxocc\' does the opposite. Option \'random\' returns a single chain with alt. confs. chosen at
random.')
    parser.add_argument('-n', '--nFiles', type=int, default=1, help='number of files to generate in
random modes.')
    parser.add_argument('-o', '--pdbOut', type=str, default=None, help='name of and path to an output
PDB file to be created. If multiple models are to be created in random mode, will output to
directory specified.')
    parser.add_argument('-v', '--verbose', action='store_true', default=False, help='talk a little
more.')
    args=parser.parse_args()
    try:
        fmt = '{:=^'+str(os.popen('stty size', 'r').read().split()[1])+'}'
    except:
        fmt = '{:=^30}'
    print('\n'+fmt.format(' PREP_PDB.PY ')+'\n')
    if os.path.exists(args.pdbIn):
        if args.mode == 'minocc':
            if args.nFiles != 1:
                print('Only one PDB file will be created in \'minocc\' mode.')
            split_pdb_occupancy(args.pdbIn, mode='min', pdbOut=args.pdbOut, strict=False)
        elif args.mode == 'maxocc':
            if args.nFiles != 1:
                print('Only one PDB file will be created in \'maxocc\' mode.')
            split_pdb_occupancy(args.pdbIn, mode='max', pdbOut=args.pdbOut, strict=False)

```

```

elif args.mode.split('-')[0] == 'random':
    if args.mode.split('-')[1] == 'full':
        [split_pdb_random_altconf(args.pdbIn, localize=False, pdbOut=args.pdbOut, strict=True, verbose=args.verbose) for i in range(args.nFiles)]
        elif args.mode.split('-')[1] == 'local':
            [split_pdb_random_altconf(args.pdbIn, localize=True, pdbOut=args.pdbOut, strict=True, verbose=args.verbose) for i in range(args.nFiles)]
    else:
        print('Specified input PDB file does not exist; exiting.')

```

1.2.2 batch_ensemble_refine.py

These functions and command line interface are used to run phenix.ensemble_refinement on many PDB files in parallel.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, argparse, subprocess, errno, glob, pickle, json
import itertools as it
from uuid import uuid4
from random import randint
from multiprocessing import cpu_count

def generate_phenix_ensemble_refinement_commands(phenixPath, pdbList, mtzList, pTLSList,
tOffsetList, tauXList, cifList=[], extra_restraints=None, harmSel='',
ensembleReductionList=False, removeAltlocList=True, randSeedList=[], queuing=None, runName=None,
nProcesses=None, rand_seed_bits=32, commands_to_text=True):
    """
    Generate commands given a grid of parameters constructed from a series of parameter lists.

    Input:
    phenixPath: Path to directory containing phenix.ensemble_refinement.
    pdbList: List of paths to PDB files to be refined.
    mtzList: List of paths to MTZ files for refinement.
    pTLSList: The "percentile of atoms with the poorest fitting ADPs [to exclude from
    successive rounds] of TLS parameter fitting"; a list of floats with at least one entry between 0
    and 1.0. If not specified, defaults to 0.8.
    tOffsetList: Temperature offset parameter in K which "controls the X-ray weight in a
    system independent manner whilst maintaining the target temperature"; a list of floats with at
    least one entry greater than or equal to zero. If not specified, defaults to 5.0 K.
    tauXList: Time constant in ps for the time-dependent memory fuction used to restrain
    Fmodel; a list of floats with at least one element greater than zero.
    cifList: Absolute path to at least one .cif file to be used during refinement.
    extra_restraints: Absolute path to a .eff file specifying any extra non-standard geometry
    restraints for refinement.
    harmSel: A selection string specifying atoms for which to apply harmonic restraints
    during refinement.
    ensembleReductionList: Whether to perform ensemble reduction; a list of booleans equal to
    `[True]`, `[False]`, or `[True, False]`.
    removeAltlocList: Whether to instruct phenix.ensemble_refinement to remove altlocs prior
    to refinement.
    randSeedList: A list of random seeds; if empty, a 32-bit seed will be generated
    automatically for phenix.ensemble_refinement.

    Output:
    """

```

paramList: a list of lists of parameter grid points used to generate commands.
 refineDirs: a list of paths to all refinement directories created over the grid.

```

'''
if type(pdbList) is not list:
    pdbList = list(pdbList)
if type(mtzList) is not list:
    mtzList = list(pdbList)
if type(pTLSList) is not list:
    pTLSList = list(pTLSList)
if type(tOffsetList) is not list:
    tOffsetList = list(tOffsetList)
if type(tauXList) is not list:
    tauXList = list(tauXList)
if type(ensembleReductionList) is not list:
    ensembleReductionList = [ensembleReductionList]
if type(removeAltlocList) is not list:
    removeAltlocList = [removeAltlocList]
if type(cifList) is not list:
    cifList = [cifList]
if len(cifList) >= 1:
    cifList = ' '.join([cif for cif in cifList])
if type(randSeedList) is not list:
    if type(randSeedList) in (float, int):
        randSeedList = list(int(randSeedList))
    elif type(randSeedList) is str and randSeedList.lower() != 'unique':
        raise ValueError('Invalid random seed mode string specified; valid options are
(unique, ).')
    elif type(randSeedList) is list and len(randSeedList) > 0:
        if type(randSeedList[0]) is str and randSeedList[0].lower() in ('unique',):
            randSeedList = randSeedList[0] # A unique random seed for each run will be generated
        else:
            randSeedList = [abs(int(v)) for v in randSeedList]
    elif type(randSeedList) is list and len(randSeedList) == 0:
        randSeedList = [randint(1,2**rand_seed_bits)] # A single random seed for all runs
    else:
        raise ValueError('Invalid random seed provided.')
refineDirs = []
commands = []
subcommands = []
pCount = 0
if runName is not None:
    baseDir = os.path.join(os.path.split(pdbList[0])[0], runName)
    try:
        os.mkdir(baseDir)
    except FileExistsError:
        pass
    if type(randSeedList) is list:
        paramsList = [list(tup) for tup in
it.product(*[pdbList,mtzList,pTLSList,tOffsetList,tauXList,ensembleReductionList,removeAltlocList
,randSeedList])] # Generate all combinations of input parameters
    elif type(randSeedList) is str and randSeedList.lower() in ('unique',):
        paramsList = [list(tup) for tup in
it.product(*[pdbList,mtzList,pTLSList,tOffsetList,tauXList,ensembleReductionList,removeAltlocList
])] # Generate all combinations of input parameters
    [paramsList[i].append(randint(1,2**rand_seed_bits)) for i in range(len(paramsList))]
    for i,params in enumerate(paramsList):
        baseDir,pdb = os.path.split(params[0])
        mtz = os.path.splitext(os.path.split(params[1])[1])[0]
        if runName is not None:
            refineDirs.append(os.path.join(baseDir,runName,'{}_{}_{}_{}_{}_{}_{}'.format(runName,os.path.s
plitext(pdb)[0],mtz,params[2],params[3],params[4],params[5],params[6],params[7])))
        else:
            refineDirs.append(os.path.join(baseDir,'{}_{}_{}_{}_{}_{}_{}'.format(os.path.splitext(pdb)[0],
mtz,params[2],params[3],params[4],params[5],params[6],params[7])))
    try:
        os.mkdir(refineDirs[-1])

```

```

except OSError as exception:
    if exception.errno != errno.EEXIST:
        raise
try:
    if runName is not None:
        paramFile = os.path.join(refineDirs[-1],runName+'_ensref_params.txt')
    else:
        paramFile = os.path.join(refineDirs[-1],'ber_params.txt')
    with open(paramFile,'w') as f:
        f.write('generate_phenix_ensemble_refinement_commands created a command string
for this directory with the following parameters:\n')
        for param in
zip(['PDB','MTZ','pTLS','tOffset','TauX','EnsembleReduction','RemoveAltConfsFromInputFile','Rando
m Seed'],params):
            f.write('{}: {}\n'.format(param[0],param[1]))
        if type(cifList) == str:
            f.write('CIF Files: {}\n'.format(cifList))
        if type(harmSel) == str and harmSel != '':
            f.write('Harmonic restraints selections: {}\n'.format(harmSel))
        if type(extra_restraints) == str and os.path.isfile(extra_restraints):
            f.write('Custom geometry restraints: {}\n'.format(extra_restraints))
except:
    raise
if queuing is None:
    subcommands.append((refineDirs[-
1],[os.path.join(phenixPath,'phenix.ensemble_refinement'),params[0],params[1],cifList,'ptls='+str
(params[2]),'wxray_coupled_tbath_offset='+str(params[3]),'tx='+str(params[4]),'ensemble_reduction
='+str(params[5]),'remove_alt_conf_from_input_pdb='+str(params[6]),'extra_restraints_file='+str(e
xtra_restraints),'ensemble_refinement.random_seed='+str(params[7])])) # Creates a tuple with
(dir_for_cwd, command_args) to pass to POpen below.
    elif queuing == 'slurm':
        subcommands.append((refineDirs[-1],['srun','--exclusive','--ntasks','1','--
nodes','1',os.path.join(phenixPath,'phenix.ensemble_refinement'),params[0],params[1],cifList,'ptl
s='+str(params[2]),'wxray_coupled_tbath_offset='+str(params[3]),'tx='+str(params[4]),'ensemble_re
duction='+str(params[5]),'remove_alt_conf_from_input_pdb='+str(params[6]),'extra_restraints_file=
'+str(extra_restraints),'ensemble_refinement.random_seed='+str(params[7])])) # Creates a tuple
with (dir_for_cwd, command_args) to pass to POpen below.
        if harmSel != '':
            subcommands[-1][1].append('harmonic_restraints.selections='\{\}\'.format(harmSel))

    for i,item in enumerate(subcommands[-1][1]):
        if item == '' or item == ' ' or item == []:
            subcommands[-1][1].pop(i)

    pCount += 1
    if pCount == nProcesses:
        commands.append(subcommands)
        subcommands = []
        pCount = 0
if pCount > 0:
    commands.append(subcommands)

if commands_to_text:
    with open(os.path.join(baseDir,'{}_commands.txt'.format(runName)), 'w') as f:
        for comm in (c for subcomms in commands for c in subcomms):
            f.write('{}\n'.format(comm))

return(paramsList,refineDirs,commands)

def run_phenix_ensemble_refinement_commands(commands,nProcesses):
    """
    Run everything in commands nProcesses at a time.
    Inputs:
        commands: A list of tuples, where each entry has a command list and a directory string
specifying the directory in which to execute the command.
        nProcesses: The maximum number of simultaneous processes to run at the same time.
    Outputs:
        groups:

```

```

'''
import subprocess
try:
    from itertools import zip_longest
except:
    try:
        print('\nFunction zip_longest not found in itertools, importing as izip_longest.')
        from itertools import izip_longest as zip_longest
    except:
        raise
groups = [(subprocess.Popen(cmd, cwd=rdir, shell=False, stdout=None) for rdir,cmd in
commands)] * nProcesses # Create an iterator object containing subprocesses
for processes in zip_longest(*groups): # For each iterator...
    for p in filter(None,processes): # For each process in iterator...
        p.wait() # Wait for the task to complete
return(groups)

def statistics_from_ensemble_refinement_log(refineDirs,runName=None,outDir=False):
if type(refineDirs) != list and os.path.isdir(refineDirs):
    tempDirList = []
    for d in os.path.listdir(refineDirs):
        if os.path.isdir(d):
            tempDirList.append(os.path.join(refineDirs,d))
    refineDirs = tempDirList
refineStatsDict = {}
for thisDir in refineDirs:
    print('Looking for log files in {}'.format(thisDir))
    refineStatsDict[thisDir] = {}
    for file in glob.glob(os.path.join(thisDir,'*.log')):
        print('Getting statistics from {}'.format(file))
        refineStatsDict[thisDir][file] = {}
        try:
            with open(os.path.join(thisDir,file),'r') as f:
                for line in f:
                    if 'FINAL Rwork' in line:
                        print('Found R-factors...')
                        l = line.split()
                        refineStatsDict[thisDir][file]['rwork']=l[3]
                        refineStatsDict[thisDir][file]['rfree']=l[6]
                        refineStatsDict[thisDir][file]['rf/rw']=l[9]
                    elif 'Final Twork' in line:
                        print('Found T-factors...')
                        l = line.split()
                        refineStatsDict[thisDir][file]['twork']=l[3]
                        refineStatsDict[thisDir][file]['tfree']=l[6]
                        refineStatsDict[thisDir][file]['tf/tw']=l[9]
                    break
        except:
            print('Could not extract statistics from {} in {}'.format(file,thisDir))
if outDir:
    _,prefix = os.path.split(outDir)
    print('\nDumping refinement statistics into {}_refine_stats.json.'.format(prefix))
    try:
        with open(os.path.join(outDir,runName+'_refine_stats.json'),'w') as outJSON:
            outJSON.write(json.dumps(refineStatsDict,sort_keys=True,indent=2))
        print('Success!')
    except:
        print('Failed to write to
{}!'.format(os.path.join(outDir,prefix+'_refine_stats.json')))
        raise
return(refineStatsDict)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='a wrapper for phenix.ensemble_refinement which
can be used to grid pTLS, Toffset, and tx params.')
    parser.add_argument('-p','--pdbList',type=str,nargs='*',required=True,help='at least one PDB
file to submit for ensemble refinement.')
    parser.add_argument('-m','--mtzList',type=str,nargs='*',required=True,help='at least one MTZ
file against which to refine PDB files in pdbList.')

```

```

parser.add_argument('--cif', type=str, nargs='*', default=[''], help='use the following .cif
files during ensemble refinement.')
parser.add_argument('--extraRestrains', type=str, default=None, help='.eff file specifying
extra non-standard geometry restraints for ensemble refinement.')
parser.add_argument('--harmSel', nargs='*', default=[''], help='PHENIX selection string
specifying atoms for which to use harmonic restraints during ensemble refinement.')
parser.add_argument('--runName', type=str, default='', help='a descriptive string to prepend to
directory names.')
parser.add_argument('--outDir', type=str, default='', help='directory in which to put ensemble
refinement subdirectories. If not specified, directories will be created in current location.')
parser.add_argument('--pTLSList', type=float, nargs='*', default=[0.8], help='the \'percentile of
atoms with the poorest fitting ADPs [to exclude from successive rounds] of TLS parameter fitting;
at least one float between 0 and 1.0.')
parser.add_argument('--tOffsetList', type=float, nargs='*', default=[5.0], help='temperature
offset parameter in K which \'controls the X-ray weight in a system independent manner whilst
maintaining the target temperature\'; at least one float greater than or equal to zero.')
parser.add_argument('--tauXList', nargs='*', default=[None], help='time constant in ps for the
time-dependent memory fuction used to restrain Fmodel; at least one float greater than zero.')
parser.add_argument('--removeAltConfs', choices=[True, False, 'both'], default=True, help='whether
or not to remove alternative conformations from the input model.')
parser.add_argument('--
reduceEnsemble', choices=[True, False, 'both'], default=False, help='whether or not to reduce the
number of models in the output ensemble.')
parser.add_argument('--randSeedList', nargs='*', default=[2679941], help='if randSeedList is an
integer, it will be passed to ER for refinement; if it is None, a 32-bit integer will be
generated. By default, passes the random seed that ER is hardcoded to use if no seed is specified
by the user.')
parser.add_argument('--phenixPath', type=str, default='', help='absolute path to the directory
containing the phenix.ensemble_refinement executable.')
parser.add_argument('--nProcesses', type=int, default=0, help='number of simultaneous refinement
jobs to run on a given CPU. If omitted, will be optimized automatically.')
parser.add_argument('--queuing', choices=[None, 'slurm'], default=None, help='if a supporting
queuing system is specified, use it to run commands.')
parser.add_argument('-v', '--verbose', action='store_true', default=False, help='open the
floodgates.')
args = parser.parse_args()

try:
    fmt = '{:=^'+str(os.popen('stty size', 'r').read().split()[1])+}'
except:
    fmt = '{:=^30}'
print('\n'+fmt.format(' BATCH_ENSEMBLE_REFINE.PY ')+'\n')

# Set everything up...
# Check for phenix.refine in path
try:
    phxInfo = subprocess.check_output([os.path.join(args.phenixPath, 'phenix.refine'), '--
version', '--dry-run']).split()
    if args.verbose:
        print('\nFound phenix.refine from build {}-{} in
path.'.format(phxInfo[9], phxInfo[12]))
except:
    print('\nInvalid path to PHENIX!')
    raise

# Make sure runName is unique
if args.runName == '': # If runName is blank and a single PDB file has been specified,
runName will be the same as the PDB file without prefix.
    if len(args.pdbList) == 1:
        args.runName = os.path.splitext(os.path.split(args.pdbList[0])[1])[0]
    else: # If many PDB files were given, runName will be run_XXXXXXXX, where the Xs are
random.
        args.runName = 'run_'+str(uuid4()).split('-')[0] # Generates a random character code
    if os.path.isdir(os.path.join(args.outDir, args.runName)): # If directory DIR already exists,
runName will be DIR_XXXXXXXX, where the Xs are random.
        args.runName = args.runName+'_'+str(uuid4()).split('-')[0]

# If you don't care where we put stuff, we'll just do it right here.
if not os.path.isdir(args.outDir) or args.outDir == '':

```

```

    args.outDir = os.getcwd()
    if args.verbose:
        print('Will generate output in current directory {}'.format(args.outDir))
os.chdir(args.outDir)

# Any PDB file(s) specified must exist.
for i,pdb in enumerate(args.pdbList):
    if not os.path.isfile(pdb):
        if args.verbose:
            print('Provided .pdb file {} not found, skipping.'.format(args.pdbList.pop(i)))
    if len(args.pdbList) == 0:
        raise OSError('Valid .pdb file(s) not found, exiting.')

# Reflections file must exist. Could add some additional checks here, but whatever
for i,mtz in enumerate(args.mtzList):
    if not os.path.isfile(mtz):
        if args.verbose:
            print('Provided .mtz file {} not found, skipping.'.format(args.mtzList.pop(i)))
    if len(args.mtzList) == 0:
        raise OSError('Valid .mtz file(s) not found, exiting.')

# If provided, custom restraints file must exist
if args.extraRestrains is not None:
    if not os.path.isfile(args.extraRestrains):
        raise OSError('Specified extra restraints file does not exist!')
    else:
        # In the future, add checks to make sure format is valid
        pass

# Processor stuff... just stick to one job/core for the moment
nCores = cpu_count()
if args.nProcesses == 0:
    args.nProcesses = nCores

# Time to process everything...
# Generate a list of phenix.ensemble_refinement commands to execute
if args.removeAltConfs == 'both':
    args.removeAltConfs = [True, False]
if args.reduceEnsemble == 'both':
    args.reduceEnsemble = [True, False]

if args.harmSel != [''] and len(args.harmSel) > 1:
    args.harmSel = ' '.join(args.harmSel)
elif args.harmSel != [''] and len(args.harmSel) == 1:
    args.harmSel = args.harmSel[0]
else:
    args.harmSel = ''

if args.verbose:
    print('\n{} input parameters:\npdbList {} \nmtzList {} \ntpTLSList {} \ntOffsetList
{} \ntauXList {} \nrandSeedList {} \nharmSel {}'.format(args.runName, args.pdbList, args.mtzList,
args.pTLSList, args.tOffsetList, args.tauXList, args.randSeedList, args.harmSel))
    paramList, refineDirs, commands =
generate_phenix_ensemble_refinement_commands(args.phenixPath, args.pdbList, args.mtzList, args.pTLSL
ist, args.tOffsetList, args.tauXList, harmSel=args.harmSel, cifList=args.cif, extra_restraints=args.ex
traRestrains, ensembleReductionList=args.reduceEnsemble, removeAltlocList=args.removeAltConfs, rand
SeedList=args.randSeedList, queuing=args.queuing, runName=args.runName, nProcesses=args.nProcesses)

# Execute the commands. If distributing over nodes...
if args.verbose:
    print('\nRunning a maximum of {} simultaneous refinement jobs with 1 core allocated per
job.'.format(args.nProcesses))
    for i,comm in enumerate(commands):
        if args.verbose:
            print('Starting phenix.ensemble_refinement commands batch {} of
{}...'.format(i, len(commands)))
            jobOutput = run_phenix_ensemble_refinement_commands(comm, args.nProcesses)
        if args.verbose:
            print('Job output:\n{}'.format(jobOutput)) # In the future, convert this to some sort of

```

```

diagnostic

# Pull R-factors from log files
if args.verbose:
    print('\nAggregating refinement statistics...')
    refineStatsDict =
statistics_from_ensemble_refinement_log(refineDirs,args.runName,outDir=args.outDir)

# Dump everything into a tasty pickle
print('\nDumping everything into {}_output.pkl.'.format(args.runName))
try:
    with open(os.path.join(args.outDir,args.runName,args.runName+'_output.pkl'),'wb') as
outPickle:
        pickle.dump([paramList,refineDirs,commands,refineStatsDict],outPickle)
        print('Success!')
    except:
        print('Failed to write to
{}!'.format(os.path.join(args.outDir,args.runName+'_output.pkl')))
        raise

print('\nFinished!')

```

1.2.3 submit_batch_ensemble_refine_slurm-frac.sh

A sample BASH script for running batch_ensemble_refine.py on the UTSW BioHPC cluster using the SLURM queuing system.

```

#!/bin/bash
#SBATCH --job-name=BER-3I4W-11
#SBATCH --partition=super
#SBATCH --nodes=4
#SBATCH --time=8-0:0:0
#SBATCH --mail-type=ALL
#SBATCH --output=slurm.%j.%N.out
#SBATCH --error=slurm.%j.%N.err
# Note that this currently assumes that there are many PDB files with only a few parameters to
scan each.
# Setup
source ~/software/phenix-dev-2026/phenix_env.sh
module add python/3.4.x-anaconda
# Args
NNODES=4
NCORES=32
RUNNAME="ER1"
OUTDIR=`pwd`
QUEING="slurm"
PDBLIST=( $(find `pwd` -name '*.pdb' ) )
MTZLIST=( $(find `pwd` -name '*.mtz' ) )
PTLS=(0.8)
TOFFSET=(1.0)
TXVAR=(None)
CIF="~/ensref/pdz3_cript/3I4w_apo_PDZ3/as_P21_job21/round_2/focus/models/SNN.cif"
HARMSEL="rename S04 and element S"
# Figure out how to distribute things
N=${#PDBLIST[@]}
if [ "$N" -lt "$NCORES" ]; then
    NCHUNKS=1
else
    NCHUNKS=$((N / NCORES))
fi
if [ "$N" -gt "$NCORES" ] && [ $(( $N % 32 )) != 0 ]; then
    NCHUNKS=$((NCHUNKS+1))
fi

```

```

echo -e "\nDividing $N PDB files into $NCHUNKS jobs over $NNODES nodes with $NCORES cores
each.\n"
# Execute the command for each set of PDB files
STARTIDX=0
for (( i=0; i <= $NCHUNKS-1; i++ )); do
    echo "Running chunk $i..."
    THISRUN=$RUNNAME\_i
    THISPDBARRAY=${PDBLIST[@]:$STARTIDX:$NCORES}
    python ~/python/multiconf_tools/batch_ensemble_refine.py -p ${THISPDBARRAY[@]} -m
    ${MTZLIST[@]} --cif $CIF --harmSel $HARMSEL --runName $THISRUN --outDir $OUTDIR --pTLSList
    ${PTLS[@]} --tOffsetList ${TOFFSET[@]} --tauXList ${TXVAR[@]} --queuing slurm --verbose
    sleep 2
    STARTIDX=$((STARTIDX + NCORES))
done
wait

```

1.2.4 MIxnyn.py

This file is a Python wrapper for the MIxnyn C executable distributed as part of the MILCA package (available at <https://www.ucl.ac.uk/ion/departments/sobell/Research/RLEmon/MILCA>).

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, subprocess
import numpy as np
from uuid import uuid4

def MIxnyn(x,y,k=6,path='',outdir=os.getcwd(),shuffle=False,tidy=True):
    """
    Calculate mutual information using the C program MIxnyn.
    Input:
        x: ndx columns (classes) with n data points (observations) per column.
        y: ndy columns (classes) with n data points (observations) per column.
        k: number of nearest neighbors for MI estimator
        path: path to MIxnyn executable
        outDir: directory to which to write data file used by MIxnyn for calculation of MI
        shuffle: if true, will randomize the order of rows in y prior to MI calculation.
        tidy: if true, data file written for MIxnyn will be deleted after calculation.
    Output:
        ... mi: the mutual information calculated by MIxnyn.
        ...
    if path != '':
        if 'MIxnyn' not in os.listdir(path):
            raise ValueError('Invalid path to MIxnyn executable!')
    x=x.copy()
    y=y.copy()
    if len(x.shape) == 1:
        nx = len(x)
        ndx = 1
    else:
        ndx,nx = x.shape
        if ndx > nx:
            x=x.T
            ndx,nx = x.shape
    if len(y.shape) == 1:

```

```

        ny = len(x)
        ndy = 1
    else:
        ndy,ny = y.shape
        if ndy > ny:
            y=y.T
            ndy,ny = y.shape
    if nx != ny: # Number of points must be the same for all channels of x and y
        raise ValueError('X ({} points) and Y ({} points) data vectors passed to MInyn must be
the same length!'.format(nx,ny))
    else:
        n=nx
    if shuffle:
        np.random.shuffle(y.T) # shuffle rows in place... can cause problems
        #[np.random.shuffle(yp[:,i]) for i in range(yp.shape[1])] # shuffle rows in each column
independently... non-physical
    datf = os.path.join(outDir, 'MI_{}_{}.txt'.format(shuffle,uuid4())) # Define output file with
UID
    with open(datf,'w+') as f:
        for i in range(n):
            f.write('{} {} \n'.format(' '.join([str(e) for e in x[:,i]]), ' '.join([str(e) for e in
y[:,i]]))) # Write appropriately formatted line to file; each line represents data point N over
some number of x channels and some number of y channels
        p =
subprocess.Popen([os.path.join(path, 'MInyn'), datf, str(ndx), str(ndy), str(n), str(k)], cwd=outDir, sh
ell=False, stdout=subprocess.PIPE)
        mi = p.communicate()
        try:
            mi = float(mi[0])
        except:
            print('MInyn output:\n{}'.format(mi))
            raise
        try:
            p.kill()
        except OSError:
            pass
        if tidy:
            os.remove(datf)
        del(x,y)
        return(mi)

```

1.2.5 cmap.py

This set of functions and command line interface can be used to calculate inter- or intramolecular contact graphs and distance matrices from PDB files.

```

#!/usr/bin/env python

# Imports for Python 2.X
from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, csv, argparse, pickle
import numpy as np
import itertools as it
from Bio import PDB
from scipy.cluster import hierarchy as hac
import matplotlib.pyplot as plt

```

```

from mpl_toolkits.axes_grid1 import make_axes_locatable

def symm_chain_fix(pdbIn,pdbOut,verbose=False):
    """
    Convert PyMOL symexp output with duplicate chain IDs to something less disastrous. Replaces
    character 21 of lines with ATOM with new chain ID following each TER entry.
    Input:
        pdbIn: Path to a PDB file to process.
        pdbOut: Path to and name of the PDB file to create.
    """
    print('Fixing chain IDs in '+os.path.split(pdbIn)[1]+'...')
    idList = list('ABCDEFGHIJKLMNQRSTUWXYZabcdefghijklmnopqrstuvwxy')
    chainIdx = 0
    with open(pdbIn,'r') as f:
        with open(pdbOut,'w') as g:
            for lin in f:
                if ("ATOM" in lin or "ANISOU" in lin or "HETATM" in lin) and "TER" not in lin:
                    lout = list(lin)
                    lout[21] = idList[chainIdx]
                    g.write(''.join(lout))
                    if verbose:
                        print(''.join(lout).rstrip())
                elif "TER" in lin:
                    chainIdx += 1
                    g.write(lin)
                    if verbose:
                        print(lin.rstrip())
                else:
                    g.write(lin)
                    if verbose:
                        print(lin.rstrip())
            if chainIdx == 51:
                print('Chain limit exceeded, exiting.')
                return

def strip_hetatm(pdbIn,pdbOut=None,verbose=False):
    path,file = os.path.split(pdbIn)
    prefix=os.path.splitext(file)[0]
    if pdbOut == None:
        pdbOut = os.path.join(path,prefix+'_stripped.pdb')
    with open(pdbIn,'r') as f:
        with open(pdbOut,'w') as g:
            for lin in f:
                if "HETATM" not in lin:
                    g.write(lin)
                else:
                    if verbose:
                        print('\tOmitting {}'.format(lin.rstrip()[:26]))
                    else:
                        continue
    return(pdbOut)

def get_chain_resi_idx(pdbIn,chain):
    path,file = os.path.split(pdbIn)
    prefix=os.path.splitext(file)[0]
    structure = PDB.PDBParser().get_structure(prefix,pdbIn)
    model = structure[0]
    resiIdxList = []
    try:
        for i,resi in enumerate(model[chain]):
            resiIdxList.append((i,resi.id[1]))
    except KeyError:
        print('\nInvalid chain ID specified!\n')
        raise
    return(resiIdxList)

def merge_chains(pdbIn,growChain,shrinkChain,outDir=None,verbose=False):
    """
    Combine chains with non-continuous indexing. This is useful for calculating the contact

```

```

matrix of a complex.
Input:
  pdbIn:
  growChain: Chain ID which consumes shrinkChain.
  shrinkChain: Chain ID to merge into growChain.
  outDir: Optional path to which to save model_chainMerge.pdb. If None, saves to current
path.
  verbose: Prints modified PDB file to stdout during reorganization.
Output:
  pdbOut: Path and complete name of output PDB file.
  resiIdxDict: A dictionary where each key is a chain ID in the output structure that
references a list of lists, where each sublist is [resiIndex, resiNoPreMerge, resiNoPostMerge]
I was extremely tired when I wrote this...
'''

path,file = os.path.split(pdbIn)
prefix=os.path.splitext(file)[0]
print('Merging chains in {}'.format(file))
# Pull header from PDB
header = []
with open(pdbIn,'r') as f:
    for lin in f:
        linStart = lin.split()[0]
        if ("REMARK" in linStart or "CRYST" in linStart or "SCALE" in linStart):
            header.append(lin)
        else:
            break

structure = PDB.PDBParser().get_structure(prefix,pdbIn)
model = structure[0]
try:
    growIdx = [resi.id[1] for resi in list(model[growChain].get_residues())]
except KeyError:
    print('Invalid chain ID specified for growChain!')
    raise
try:
    shrinkIdx = [resi.id[1] for resi in list(model[shrinkChain].get_residues())]
except KeyError:
    print('Invalid chain ID specified for shrinkChain!')
mergeIdxStart = list(model['A'].get_residues())[-1].id[1] + 1 # When we merge into the other
chain, start with numbering outside of used range
changeList = []
for i,resi in enumerate(model[shrinkChain].get_residues()):
    j = i+mergeIdxStart
    changeList.append((i,resi.id[1],j))
    resi.id = ('',j,'')
writer = PDB.PDBIO()
writer.set_structure(structure)
if outDir == None:
    pdbOutTemp = os.path.join(path,'~'+prefix+'_chainMerge.pdb')
    pdbOut = os.path.join(path,prefix+'_chainMerge.pdb')
else:
    pdbOutTemp = os.path.join(outDir,'~'+prefix+'_chainMerge.pdb')
    pdbOut = os.path.join(outDir,prefix+'_chainMerge.pdb')
writer.save(pdbOutTemp)
pdbOutTempList = []
with open(pdbOutTemp,'r') as f:
    for lin in f:
        pdbOutTempList.append(lin)
with open(pdbOutTemp,'w') as f:
    for lout in header:
        f.write(lout)
    for lout in pdbOutTempList:
        f.write(lout)
with open(pdbOutTemp,'r') as f:
    with open(pdbOut,'w') as g:
        pdbDict = {}
        chainList = []
        for lin in f:
            linStart = lin.split()[0]

```

```

        if ("REMARK" in linStart or "CRYST" in linStart or "SCALE" in linStart):
            g.write(lin)
        elif ("ATOM" in linStart or "ANISOU" in linStart or "HETATM" in linStart):
            lout = list(lin)
            if lout[21] == shrinkChain:
                lout[21] = growChain
            if lout[21] not in chainList:
                chainList.append(lout[21])
                pdbDict[lout[21]] = []
            pdbDict[lout[21]].append(lout)
        for chain in sorted(list(pdbDict.keys())):
            for entry in pdbDict[chain]:
                lout = ''.join(entry)
                if verbose:
                    print(lout.rstrip())
                g.write(lout)
            g.write("TER\n")
        g.write("END")
    os.remove(pdbOutTemp)
    structure = PDB.PDBParser().get_structure(prefix+'_merged',pdbOut)
    model = structure[0]
    resiIdxDict = {}
    for i, chain in enumerate(chainList):
        resiIdxDict[chain] = []
        resiIdxDict[chain] = [[j, resi.id[1], resi.id[1]] for j, resi in
            enumerate(list(model[chain].get_residues()))]
        for posn in resiIdxDict[chain]:
            for change in changeList:
                if posn[1] == change[2]:
                    posn[1] = change[1]
    return(pdbOut, resiIdxDict)

def calculate_cm(pdbFile, refChainID='A', resis = [], mode='distance-internal', specialResiDict =
None, vdWfrac = 0.2, ignore_alts = False, verbose = True):
    """
    Calculate contact matrices D_raw and D_bin from chain pdbChain of model in pdbFile.
    If latticeMode is True, calculate lattice contact matrix for chain pdbChain relative to other
    chains in pdbFile.
    Input:
        pdbFile: Path to PDB file from which to calculate contact matrix.
        refChainID: Chain in pdbFile for which to calculate contact matrix or to be used as
        reference chain in lattice mode.
        resis: List of residue indices.
        latticeMode: If true, calculate lattice contact matrix from refChainID to nearest
        symmetry mates in the unit cell.
        outDir: If a path is specified, save contact matrices to CSV files there.
        vdWfrac: Fractional space which, in addition to van der Waals radii of atoms i and j,
        determines contact threshold.
        verbose: If true, provide progress info.
    Output:
        D_raw: Contact matrix where each element is the distance is the closest distance in
        Angstroms between two atoms in residue i and residue j for refChainID (normal mode) or for
        refChainID relative to nearest chain (lattice mode) in pdbFile.
        D_bin: Binarized version of D_raw in which distances are converted to either 0 or 1,
        where 1 indicates a contact.
    """
    def strip_hetatom(structure):
        """
        This doesn't appear to work. Bug in Bio.PDB?
        """
        for model in structure:
            for chain in model:
                for residue in chain:
                    if residue.id[0] != ' ':
                        print('Removing {}'.format(residue.id))
                        chain.detach_child(residue.id)
                if len(chain) == 0:
                    model.detach_child(chain.id)
        return(structure)

```

```

def atom_dist(atomA,atomB):
    """
    Return distance between two atoms.
    """
    diff = atomA.coord - atomB.coord
    dist = np.sqrt(np.sum(diff**2))
    return(dist)

# Get operation mode:
mode = mode.split('-')
if 'distance' in mode:
    matrixMode = 'distance'
elif 'overlap' in mode:
    matrixMode = 'overlap'
    if 'minimum' in mode:
        overlapMode = 'minimum'
    elif 'weighted' in mode:
        overlapMode = 'weighted'

if 'internal' in mode:
    latticeMode = False
elif 'lattice' in mode:
    latticeMode = True
# Load the PDB file; for now, handles only one model object
pdbName = os.path.splitext(os.path.basename(pdbFile))[0]
structure = PDB.PDBParser().get_structure(pdbName,pdbFile)
#structure = strip_hetatom(structure) doesn't work...?
model = structure[0]
if len([c for c in model.child_list if c.id != 'S']) > 1 and 'internal' in mode:
    print('\tWarning: multiple chains present in internal calculation mode; consider merging
chains to calculate all distances in ASU.')
try:
    refChain = model[refChainID]
except KeyError:
    print('Specified refChainID "{}" not found.'.format(refChainID))
    raise
# Data for binarizing the contact matrix from Table I, column II of:
# Tsai, J., Taylor, R., Chothia, C. & Gerstein, M. The packing density in proteins: standard
radii and volumes. J Mol Biol 290, 253-266 (1999).
# Note that OXT distance is not explicitly provided in this paper.
atomRadii = {}
atomRadii['ALA']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88}

atomRadii['ARG']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.88,'
NE':1.64,'CZ':1.61,'NH1':1.64,'NH2':1.64}

atomRadii['ASN']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'OD1':1.42,
'ND2':1.64}

atomRadii['ASP']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'OD1':1.42,
'OD2':1.46}
    atomRadii['CYS']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'SG':1.77}

atomRadii['GLN']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.61,'
OE1':1.42,'NE2':1.64}

atomRadii['GLU']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.61,'
OE1':1.42,'OE2':1.46}
    atomRadii['GLY']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46}

atomRadii['HIS']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'ND1':1.88,
'CD2':1.64,'CE1':1.88,'NE2':1.64}

atomRadii['ILE']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG1':1.88,'CG2':1.88,
'CD1':1.88}

atomRadii['LEU']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD1':1.88,
'CD2':1.88}

```

```

atomRadii['LYS']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.88,'
CE':1.88,'NZ':1.64}

atomRadii['MET']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'SD':1.77,'
CE':1.88}

atomRadii['PHE']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'CD1':1.76,
'CD2':1.76,'CE1':1.76,'CE2':1.76,'CZ':1.76}

atomRadii['PRO']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.88}
atomRadii['SER']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'OG':1.46}

atomRadii['THR']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'OG1':1.46,'CG2':1.88
}

atomRadii['TRP']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'CD1':1.88,
'NE1':1.64,'CE2':1.61,'CD2':1.61,'CE3':1.76,'CZ3':1.76,'CH2':1.76,'CZ2':1.76}

atomRadii['TYR']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'CD1':1.76,
'CD2':1.76,'CE1':1.76,'CE2':1.76,'CZ':1.76,'OH':1.46}

atomRadii['VAL']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG1':1.88,'CG2':1.88
}

    if specialResiDict != None and type(specialResiDict) == dict:
        for resi in specialResiDict.keys():
            atomRadii[resi] = specialResiDict[resi]
        # Calculate the raw contact matrix
        nResis = len(list(refChain.get_residues()))
        D_raw = np.zeros((nResis,nResis),np.float)
        D_bin = np.zeros((nResis,nResis),np.int)
        if latticeMode: # If lattice mode, create a list of chains in lattice to iterate over,
            excluding reference chain
            print('\tCalculating inter-chain distances in lattice mode...')
            chainList = [model[childID] for childID in list(model.child_dict.keys()) if childID !=
            refChainID]
            if len(chainList) < 2:
                raise ValueError('Multiple chains needed to operate in lattice mode!')
            elif not latticeMode: # Or, stick to a single chain
                print('\tCalculating intra-chain distances in single-chain mode...')
                chainList = [refChain]
            resiIdxList = []
            for i, resi in enumerate(refChain):
                resiIdxList.append((i,resi.id[1]))
            for thisChain in chainList:
                for i, resiA in enumerate(refChain):
                    for j, resiB in enumerate(thisChain):
                        if i > nResis and j > i:
                            continue
                        minDist = 1e9
                        for atomA,atomB in it.product(resiA,resiB):
                            if atomA.id[0] == 'H' or atomB.id[0] == 'H':
                                continue
                            if verbose:
                                print('\t{:>2} {:>3} {:>3} <----> {:>2} {:>3} {:>3}'.format(refChain.id,
                                resiA.id[1], atomA.id, thisChain.id, resiB.id[1], atomB.id),end='\r')
                            if atomA.is_disordered() and atomB.is_disordered() and not ignore_alts:
                                for diso_atomA, diso_atomB in it.product(atomA.disordered_get_list(),
                                atomB.disordered_get_list()):
                                    thisDist = atom_dist(diso_atomA, diso_atomB)
                                    if thisDist < minDist:
                                        radii =
                                        (atomRadii[resiA.get_resname()][diso_atomA.id],atomRadii[resiB.get_resname()][diso_atomB.id])
                                        minDist = thisDist
                            elif atomA.is_disordered() and not atomB.is_disordered() and not ignore_alts:
                                for diso_atomA, atomB in it.product(atomA.disordered_get_list(),
                                [atomB]):
                                    thisDist = atom_dist(diso_atomA, atomB)
                                    if thisDist < minDist:

```

```

        radii =
(atomRadii[resiA.get_resname()][diso_atomA.id],atomRadii[resiB.get_resname()][atomB.id])
        minDist = thisDist
        elif not atomA.is_disordered() and atomB.is_disordered() and not ignore_alts:
            for atomA, diso_atomB in it.product([atomA],
atomB.disordered_get_list()):
                thisDist = atom_dist(atomA, diso_atomB)
                if thisDist < minDist:
                    radii =
(atomRadii[resiA.get_resname()][atomA.id],atomRadii[resiB.get_resname()][diso_atomB.id])
                    minDist = thisDist
            else:
                thisDist = atom_dist(atomA, atomB)
                if thisDist < minDist:
                    radii =
(atomRadii[resiA.get_resname()][atomA.id],atomRadii[resiB.get_resname()][atomB.id])
                    minDist = thisDist
                thisCutoff = radii[0]+radii[1]+(vdwFrac*(radii[0]+radii[1]))
                if D_raw[i,j] == 0: # If we haven't updated the matrix element yet, set it to
minDist
                    D_raw[i,j] = D_raw[j,i] = minDist
                    if minDist<thisCutoff:
                        D_bin[i,j] = D_bin[j,i] = 1
                    elif D_raw[i,j] != 0: # If it's not zero, it's been updated for a different
chain. Compare to previous value.
                        if minDist < D_raw[i,j]:
                            D_raw[i,j] = D_raw[j,i] = minDist
                            if minDist<thisCutoff:
                                D_bin[i,j] = D_bin[j,i] = 1
    if verbose:
        print('\r\tFinished distance calculations.  ')
    if type(resis) is list and len(resis) > 0:
        print('Reducing distance matrices to specified residues...')
        D_raw_sub = np.zeros(2*[len(resis)])
        D_bin_sub = np.zeros(2*[len(resis)])
        resi_id_list = [r[1] for r in resiIdxList]
        triu_idx = np.triu_indices(len(resis))
        for i,j in ((triu_idx[0][k],triu_idx[1][k]) for k in range(triu_idx[0].shape[0])):
            resi_i = resis[i]
            resi_j = resis[j]
            try:
                resi_i_idx = resi_id_list.index(resi_i)
                resi_j_idx = resi_id_list.index(resi_j)
            except ValueError:
                if verbose:
                    print('Residue pair ({}, {}) not present in reduced matrix,
skipping.'.format(resi_i,resi_j))
                continue
            D_raw_sub[i,j] = D_raw[resi_i_idx,resi_j_idx]
            D_bin_sub[i,j] = D_bin[resi_i_idx,resi_j_idx]
        D_raw_sub += np.triu(D_raw_sub,1).T
        D_bin_sub += np.triu(D_bin_sub,1).T
        return(resiIdxList, D_raw, D_bin, D_raw_sub, D_bin_sub)
    else:
        return(resiIdxList, D_raw, D_bin)

```

```

def walk_cm(cm, startPosn, thresh = 10, resis=[]):
    """

```

Generates a list of shells, where each list are positions in the contact graph which are within threshold of a position in the previous shell.

Input:

cm: A contact graph. Can be binary or weighted.
startPosn: Position in the contact graph from which to start walking. This is the column index in the contact graph, not the residue number.
thresh: Optional threshold in Angstroms below which to keep a contact.
resis: Optional list of tuples with (chain, position index) for residues in the protein, e.g., [('B',1),('A',384),('A',385),...]. Matrix index can also be substituted for chain. Generate such a list for a given chain: resis = [('A',i) for i in np.arange(297,416)]

Output:

```

    ... shellList: A list of lists, ordered by shell away from starting position.
    ...
    if type(resis) is list and len(resis) > 0 and startPosn in [r[1] for r in resis] and
len(resis) == len(cm):
        startPosn = [r[1] for r in resis].index(startPosn)+1
        if cm.dtype != 'bool' and len(np.unique(cm)) > 2: # If not a boolean contact matrix,
threshold by thresh
            cmt = cm.astype(int) < thresh*np.arange(1,len(cm)+1) # Threshold the contact matrix,
replacing values smaller than a certain cutoff with the row index of that position.
        else:
            cmt = cm.astype(int)*np.arange(1,len(cm)+1)
            cmt[np.tril_indices(len(cmt))]=cmt[np.triu_indices(len(cmt))]
            shellList=[[startPosn],[subPosn for subPosn in cmt[startPosn-1,:]] if (subPosn != startPosn
and subPosn != 0)]] # shellList starts with the starting position as the first item and the first
shell away from it as the second item
            consumed = [posn for sublist in shellList for posn in sublist] # Make a list of everything
we've added to a shell so far
            while shellList[-1] != []: # Keep going until we no longer add anything new
                tempShell = [] # Add everything in the next shell to this
                tempConsumed = []
                for posn in shellList[-1]: # Now, consider the residues in the previous shell
                    contacts = [subPosn for subPosn in cmt[posn-1,:]] if (subPosn != posn and subPosn != 0
and subPosn not in consumed and subPosn not in tempConsumed)
                    tempShell.append(contacts) # Find all positions in contact with posn and add them if
they arent the input position, are below the threshold, and have not been added to a previous
shell
                tempConsumed = tempConsumed + contacts
                shellList.append([posn for sublist in tempShell for posn in sublist]) # Append flattened
tempShell to the main list
                consumed = consumed + tempConsumed # Add to consumed for the next time around
            shellList.pop(-1) # Condition to exit loop is that last sublist = [], so get rid of it
            if type(resis) is list and len(resis) == len(cm): # If resi info is provided for the contact
graph, map chain and resi ID to shellList
                for i in range(len(shellList)): # To do: convert nested for loops to list comprehension
                    for j in range(len(shellList[i])):
                        shellList[i][j]=resis[shellList[i][j]-1]
            return(shellList)

def cm_to_contacts(D,resiIdxList=None,mode='simple'):
    """
    Create a list of contacts from a contact graph.
    """
    width = len(D[0])
    contacts = [[posn // width, posn % width] for posn, item in enumerate(col for row in
np.triu(D) for col in row) if item == 1] # Tuple with full list of contacts
    if mode == 'simple':
        contacts = np.unique([idx for pair in contacts for idx in pair])
    else:
        raise ValueError('Unsupported mode specified.')
    if resiIdxList != None:
        for i,contact in enumerate(contacts):
            for j,idx in enumerate(resiIdxList):
                if idx[0] == contact:
                    contacts[i] = idx[1]
    return(contacts)

def cluster_cm(D,method,plot=True,outDir=None,runName='run'):
    """
    """
    Z = hac.linkage(D,method=method)
    if plot:
        fig,ax=plt.subplots(nrows=1,ncols=1)
        fig.set_size_inches(20,6)
        R = hac.dendrogram(Z,truncate_mode='lastp',labels=[int(i)+1 for i in hac.leaves_list(Z)])
        ax.set_title('{:} hierarchical clustering of contact matrix, method
\{ }\'.format(runName,args.clusterMethod))
        ax.set_xlabel('Position')
        ax.set_ylabel('Height')
        if outDir is not None:

```

```

        plt.savefig(os.path.join(outDir, '{}_dendro-
{}.pdf'.format(runName,method)),transparent=True)
    else:
        plt.show()
        plt.close('all')
        return(Z,R)
else:
    return(Z)

def plot_cm(D_raw,D_bin,outDir=None,runName='run',resiList=[]):
    """
    """
    def onpick(event):
        """
        Event processor for picker
        """
        ind = event.ind
        names=list(refineStatsDict.keys())
        print('\n')
        for item in ind:
            print('Dij = {}'.format(item))

    fig,ax = plt.subplots(nrows=1,ncols=2,figsize=(30,10))
    fig.set_size_inches(20,6)
    a = ax[0].matshow(D_raw,cmap=plt.cm.afmhot_r, picker=True)
    ax[0].set_title('{}: raw contact matrix'.format(runName))
    ax[0].set_xlabel('resi i')
    ax[0].set_ylabel('resi j')
    if resiList != []: # Relabel matrix axes with proper residue indices
        j = [int(n) for n in ax[0].get_xticks()[:-1]]
        k = [resiList[n] for n in j]
        [ax[0].set_xticklabels(k) for i in (0,1,2)]
        [ax[0].set_yticklabels(k) for i in (0,1,2)]
    divA = make_axes_locatable(ax[0])
    caxA = divA.append_axes('right',size='5%',pad=0.05)
    cbarA = plt.colorbar(a,cax=caxA,label='Distance (Angstrom)')
    ax[1].matshow(D_bin,cmap=plt.cm.afmhot, picker=True)
    ax[1].set_title('{}: thresholded contact matrix'.format(runName))
    ax[1].set_xlabel('resi i')
    ax[1].set_ylabel('resi j')
    ax[1].yaxis.tick_right()
    if resiList != []: # Relabel matrix axes with proper residue indices
        j = [int(n) for n in ax[1].get_xticks()[:-1]]
        k = [resiList[n] for n in j]
        [ax[1].set_xticklabels(k) for i in (0,1,2)]
        [ax[1].set_yticklabels(k) for i in (0,1,2)]
    fig.canvas.mpl_connect('pick_event',onpick)
    if outDir is not None:
        plt.savefig(os.path.join(outDir, '{}_cmats.pdf'.format(runName)),transparent=True)
    else:
        plt.show()
        plt.close('all')

if __name__ == "__main__":
    """

```

When run from the command line, lattice_cm.py can be used to generate contact graphs either for atoms within a single chain or between multiple chains in a lattice. To generate an input PDB for lattice mode, use the symexp command in PyMOL and then save all models in a single file with the save command. Keep the number of chains to a minimum in order to minimize processing time.

Examples:

```
./cmap.py -P /input/directory/1bfe.pdb -C 'A' -O /output/directory # WRONG
```

This command will take a raw pdb file, fix it, and then calculate the contact graph with default clustering algorithm settings.

```
./cmap.py -P /input/directory/1bfe_symm.pdb -C 'A' -O /output/directory -lfp # WRONG
```

This command will take the raw output from PyMOL/symexp, fix it, calculate the lattice contact graph with default clustering algorithm settings, and then plot the results.

```

...

parser = argparse.ArgumentParser(description='Calculate a contact matrix for any chain in a
PDB file or between chains in a lattice. Multi-chain contact matrix calculation is currently
unsupported, but could be implemented easily if needed.')
parser.add_argument('pdb',type=str,help='A PDB file to process. Strip waters first, please---
there is an issue with hetatom removal in the Bio.PDB module.')
parser.add_argument('-c','--chain',type=str,default='A',help='Chain ID for which to calculate
contact matrix. In the standard mode, this is the chain or chains over which to calculate the
contact matrix. In lattice mode, this is the reference chain.')
parser.add_argument('-s','--selectionStr',type=str,default=None,help='Subset of residues to
use in creating reduced contact matrices; has format \'resiI:resiJ,resiX:resiY\'.')
parser.add_argument('-o','--outDir',type=str,default=os.getcwd(),help='Optional path where
contact matrices will be exported in CSV format.')
parser.add_argument('-m','--mode',choices=['distance-internal','distance-lattice','overlap-
minimum-internal','overlap-minimum-lattice','overlap-weighted-internal','overlap-weighted-
lattice'],default='distance-internal',help='Flag which specifies whether or not to execute in
lattice mode.')
parser.add_argument('-w','--walkResi',type=int,default=None,help='Starting residue for
contact shell analysis.')
parser.add_argument('-p','--plot',action='store_true',default=False,help='If true, will plot
raw and binarized contact matrices.')
parser.add_argument('-l','--clusterMatrix',choices=['raw','bin','both'],default=None,help='If
user specifies \'raw\' or \'bin\', will attempt to cluster contacts. Keyword \'raw\' clusters the
raw contact matrix, while \'bin\' clusters the binarized matrix.')
parser.add_argument('--fixPDB',action='store_true',default=False,help='If true, will fix
chain names in PDB file saved following application of symexp in PyMOL.')
parser.add_argument('--
clusterMethod',choices=['single','complete','average','weighted','centroid','median','ward'],defa
ult='ward',help='Clustering method to use for linkage matrix calculation.')
parser.add_argument('--vdwPercent',type=float,default=0.2,help='Contact gap fraction.')
parser.add_argument('-v','--verbose',action='store_true',default=False,help='Open the
floodgates.')
args=parser.parse_args()

print('Inspecting input PDB file...')
path,file = os.path.split(args.pdb)
prefix=os.path.splitext(file)[0]

# First, strip HETATM entries
print('Stripping HETATM entries in PDB file {}'.format(prefix))
pdb_strip = os.path.join(args.outDir,prefix+'_stripped.pdb')
strip_hetatm(args.pdb,pdb_strip,args.verbose)

# If requested, fix input PDB chains.
if args.fixPDB:
    pdb = os.path.join(args.outDir,prefix+'_stripped_fixed.pdb')
    try:
        symm_chain_fix(pdb_strip,pdb,args.verbose)
    except:
        print('Unable to fix PDB file, exiting.')
        raise
else:
    pdb = args.pdb

# Parse residue subset list
if args.selectionStr is not None and type(args.selectionStr) is str:
    print('Parsing selection string...')
    resi_sublist=[]
    for range_str in args.selectionStr.split(','):
        if ':' in range_str:
            temp_range = [int(i) for i in range_str.split(':')]
            temp_range[1] += 1
            resi_sublist.append(list(range(*temp_range)))
        else:
            resi_sublist.append([int(range_str)])
    resi_sublist = [item for sublist in resi_sublist for item in sublist]
else:
    resi_sublist = []

```

```

    # Calculate raw and binarized contact matrices
    print('Calculating contact matrices for {}'.format(os.path.split(pdb)[1]))
    if resi_sublist != []:
        resiIdxList, D_raw, D_bin, D_raw_sub, D_bin_sub = calculate_cm(pdb,
refChainID=args.chain, vdwFrac=args.vdwPercent, mode=args.mode, verbose=args.verbose, resis =
resi_sublist)

plot_cm(D_raw_sub,D_bin_sub,outDir=args.outDir,runName='{}_subset'.format(prefix),resiList=resi_s
ublist)
    outDict =
{'resiIdxList':resiIdxList,'D_raw_full':D_raw,'D_bin_full':D_bin,'D_raw_sub':D_raw_sub,'D_bin_sub
':D_bin_sub}
    if args.outDir is not None:
        try:

np.savetxt(os.path.join(args.outDir,'{}_Draw_sub.csv'.format(prefix)),D_raw_sub,delimiter=',',fmt
='%.6f')

np.savetxt(os.path.join(args.outDir,'{}_Dbin_sub.csv'.format(prefix)),D_bin_sub,delimiter=',',fmt
='%i')

        except:
            print('Failed to save reduced contact matrices to CSV files, moving on...')
        else:
            resiIdxList, D_raw, D_bin = calculate_cm(pdb, refChainID=args.chain,
vdwFrac=args.vdwPercent, mode=args.mode, verbose=args.verbose)
            outDict = {'resiIdxList':resiIdxList,'D_raw_full':D_raw,'D_bin_full':D_bin}
            if args.outDir is not None:
                try:

np.savetxt(os.path.join(args.outDir,'{}_Draw_full.csv'.format(prefix)),D_raw,delimiter=',',fmt='%
.6f')

np.savetxt(os.path.join(args.outDir,'{}_Dbin_full.csv'.format(prefix)),D_bin,delimiter=',',fmt='%
i')

                except:
                    print('Failed to save full contact matrices to CSV files, moving on...')
                    plot_cm(D_raw,D_bin,outDir=args.outDir,runName='{}_full'.format(prefix),resiList=[r[1] for r
in resiIdxList])

    # If in internal mode and starting residue provided, perform contact shell analysis... note
that this seems to have a bug where the first residue (index 0) is omitted from output?!
    if 'internal' in args.mode and type(args.walkResi) is int:
        print('Calculating contact shells...')
        shell_list = walk_cm(D_raw, args.walkResi, thresh = 10, resis=resiIdxList)
        outDict.update({'shell_list':shell_list})

    # If in lattice mode, get contact indices from D_bin
    print('Getting contact indices for {}'.format(os.path.split(pdb)[1]))
    if 'lattice' in args.mode:
        contacts = cm_to_contacts(D_bin, resiIdxList=resiIdxList)
        outDict.update({'contacts':contacts})
        print('Found the following contacts:\n{}'.format(contacts))
        if args.outDir is not None:
            try:

np.savetxt(os.path.join(args.outDir,'{}_contacts.csv'.format(prefix)),contacts,delimiter=',',fmt=
'%i')

            except:
                print('Failed to save contact CSV file, moving on.')

    # Perform hierarchical clustering. Available methods are 'single', 'complete', 'average',
'weighted', 'centroid', 'median', and 'ward'; see documentation for scipy.cluster.hierarchy for
more info.
    if args.clusterMatrix in ('raw', 'bin', 'both'):
        print('Clustering contact matrices calculated from from
{}'.format(os.path.split(pdb)[1]))
        if args.clusterMatrix in ('raw', 'both'):
            Z_raw_full, R_raw_full =

```

```

cluster_cm(D_raw,args.clusterMethod,outDir=args.outDir,runName='{}_raw_full'.format(prefix))
    outDict.update({'Z_raw_full':Z_raw_full,'R_raw_full':R_raw_full})
    if args.clusterMatrix in ('bin','both'):
        Z_bin_full, R_bin_full =
cluster_cm(D_bin,args.clusterMethod,outDir=args.outDir,runName='{}_bin_full'.format(prefix))
    outDict.update({'Z_bin_full':Z_bin_full,'R_bin_full':R_bin_full})

# Pickle everything
print('Pickling...')
if args.outDir is not None:
    try:
        with open(os.path.join(args.outDir,'{}_contacts.pkl'.format(prefix)),'wb') as f:
            pickle.dump(outDict,f)
    except:
        print('Failed to pickle data, skipping.')

print('Finished!')

```

1.2.6 analyze_superensemble.py

A series of functions and a command line interface for calculating a variety of things (e.g., coordinate and dihedral mutual information, network models) from super-ensembles generated using `batch_ensemble_refine.py`.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, argparse, subprocess, glob, pickle, random, time, prody, multiconf_utilities
import itertools as it
import multiprocessing as mp
import numpy as np
import entropy_estimators as ee
from sys import stdout
from Bio import PDB
from scipy.optimize import curve_fit
from sklearn.decomposition import PCA
from MIXnyn import MIXnyn
from matplotlib import pyplot as plt
from matplotlib import cm as mplcm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import colors as mplcolors
from mpl_toolkits.mplot3d import Axes3D

def average_super_ensemble_refinement_statistics(pdb_list, run_name='run', out_dir=os.getcwd(),
        verbose=False):
    """
    Rips statistics out of ensemble refinement PDB headers and averages. Writes info to text file
    and returns dictionary of statistics.
    Input:
        pdb_list: A list of strings specifying paths to PDB files.
        run_name: A descriptive title.
        out_dir: A path to which to write output.
        verbose: Say more.
    Output:
    """

```

```

stats: A dictionary with the following statistics by key:
    rwork
    rfree
    coord_err
    phase_err
    number_models
    mean_rmsd_bond
    mean_rmsd_angle
    mean_rmsd_chirality
    mean_rmsd_chirality
    mean_rmsd_dihedrals
...
stats = {'rwork':[], 'rfree':[], 'coord_err':[], 'phase_err':[], 'number_models':[],
'mean_rmsd_bond':[], 'mean_rmsd_angle':[], 'mean_rmsd_chirality':[], 'mean_rmsd_planarity':[],
'mean_rmsd_dihedral':[],}
for pdb in pdb_list:
    with open(pdb,'r') as fi:
        rmsd_table_reached = False
        found_count = 0
        for i,line in enumerate(fi):
            to_int = False
            if 'R VALUE (WORKING SET)' in line:
                stat_key = 'rwork'
                found_count += 1
            elif 'FREE R VALUE :' in line:
                stat_key = 'rfree'
                found_count += 1
            elif 'COORDINATE ERROR (MAXIMUM-LIKELIHOOD BASED) :' in line:
                stat_key = 'coord_err'
                found_count += 1
            elif 'PHASE ERROR (DEGREES, MAXIMUM-LIKELIHOOD BASED) :' in line:
                stat_key = 'phase_err'
                found_count += 1
            elif 'NUMBER STRUCTURES IN ENSEMBLE :' in line:
                stat_key = 'number_models'
                to_int = True
                found_count += 1
            elif 'RMSD (MEAN RMSD PER STRUCTURE)' in line:
                rmsd_table_reached = True
                continue
            elif 'REMARK 3 BOND :' in line and rmsd_table_reached:
                stat_key = 'mean_rmsd_bond'
                found_count += 1
            elif 'REMARK 3 ANGLE :' in line and rmsd_table_reached:
                stat_key = 'mean_rmsd_angle'
                found_count += 1
            elif 'REMARK 3 CHIRALITY :' in line and rmsd_table_reached:
                stat_key = 'mean_rmsd_chirality'
                found_count += 1
            elif 'REMARK 3 PLANARITY :' in line and rmsd_table_reached:
                stat_key = 'mean_rmsd_planarity'
                found_count += 1
            elif 'REMARK 3 DIHEDRAL :' in line and rmsd_table_reached:
                stat_key = 'mean_rmsd_dihedral'
                found_count += 1
            elif 'CRYST1' in line:
                break
            else:
                continue
        try:
            if to_int:
                stats[stat_key].append(int(line.rstrip().strip(' ').split(' ')[-1]))
            else:
                stats[stat_key].append(float(line.rstrip().strip(' ').split(' ')[-1]))
        except:
            print('Failed on line {} of {}:\n{}'.format(i, pdb, line))
            raise
    if verbose:
        print('{:}\t{:20}\t{:10}'.format(pdb, stat_key, stats[stat_key][-1]))

```

```

        if found_count != 10:
            raise ValueError('Couldn\'t find all statistics for {}'.format(pdb))
        stats_mean = {stat:np.mean(vals) for stat,vals in stats.items()}
        stats_std = {stat:np.std(vals) for stat,vals in stats.items()}
        stats_min = {stat:np.min(vals) for stat,vals in stats.items()}
        stats_max = {stat:np.max(vals) for stat,vals in stats.items()}
        with open(os.path.join(out_dir, '{}_average_refinement_statistics.txt'.format(run_name)), 'w')
as fo:
    fo.write('{}: super-ensemble refinement statistics\n'.format(run_name))
    fo.write('{:<20}\t{:>10}\t{:>10}\t{:>10}\t{:>10}\n'.format('statistic', 'mean', 'std',
'min', 'max'))
    for stat in sorted(stats.keys()):
        fo.write('{:<20}\t{:>10f}\t{:>10f}\t{:>10f}\t{:>10f}\n'.format(stat,
stats_mean[stat], stats_std[stat], stats_min[stat], stats_max[stat]))
    return(stats)

def build_prody_super_ensemble(pdb_list=[], subset=None, chain='A', selection_string=None,
run_name='run', iterpose=True):
    """
    Constructs a super-ensemble from multiple PDB files containing ensembles output by
    phenix.ensemble_refinement, iterposing if requested.
    Input:
        pdb_list: Either a PDB file, a directory, or a list of paths to PDB files from which to
        build a super-ensemble. If a directory is provided, all files matching *.pdb will be included in
        the super-ensemble.
        subset: A string which tells ProDy to parse a subset of atoms. Options are '\alpha',
        '\backbone', or None. If None, all atoms will be included.
        chain: The chain of the model(s) to include in the super-ensemble.
        selection_string: A ProDy atom selection string. If None, no selection will be performed.
    Note that '\protein' and '\heavy' are already selected by default (i.e., only non-hydrogen
    protein atoms). For formatting information, see
    http://prody.csb.pitt.edu/manual/reference/atomic/select.html.
        run_name: A descriptive title
        iterpose: If True, use ProDy iterpose method to align all models in the super-ensemble.
    Output:
        ref_pdb: A prody.Selection object, which serves as a reference object for the
        super_ensemble.
        super_ensemble: A prody.Ensemble object, which has all coordinates for all models in the
        super_ensemble.
        lookup_list: A list of lists, with one entry per PDB file used to create the super-
        ensemble. Elements in each sublist are ['filename_base', file_index, model_index_in-file,
        model_index_all-files], where filename_base is a string corresponding to the basename of a given
        PDB file and the various indices are integers.
    """
    print('Building super-ensemble...')
    if type(pdb_list) is str and os.path.isdir(pdb_list):
        print('Looking for PDB files in {}'.format(pdb_list))
        pdb_list = glob.glob(os.path.join(pdb_list, '*.pdb'))
    elif type(pdb_list) is str and os.path.isfile(pdb_list):
        print('Loading {}'.format(pdb_list))
        pdb_list = [pdb_list]
    if pdb_list == []:
        print('No PDB files or directory provided, looking in current directory.')
        pdb_list = glob.glob('./*.pdb')
        if len(pdb_list) == 0:
            raise ValueError('Failed to find any PDB files in current directory!')
    n_pdb = len(pdb_list)
    lookup_list = []
    pdb_dict = {}
    for i,p in enumerate(pdb_list):
        base = os.path.splitext(os.path.basename(p))[0]
        print('Parsing {}... ({} / {})'.format(base, i+1, n_pdb))
        #try:
        if selection_string is not None:
            pdb_dict.update({base:prody.parsePDB(p, subset=subset,
chain=chain).protein.heavy.select(selection_string)}) # Parse all PDB files
        else:
            pdb_dict.update({base:prody.parsePDB(p, subset=subset, chain=chain).protein.heavy}) #
Parse all PDB files

```

```

        [lookup_list.append([base,i,0,j]) for j in range(pdb_dict[base].numCoordsets())]
    #except:
    #    pdb_dict.update({base:None})
    for i in range(len(lookup_list)):
        lookup_list[i][2] = i
    print('Super-ensemble loaded, finding and setting reference...')
    super_ensemble = prody.Ensemble(run_name) # Initialize super ensemble object
    ref_pdb = None
    for pdb in pdb_dict.values(): # Get the first valid reference
        if pdb is not None:
            super_ensemble.setCoords(pdb.getCoords()) # Define coordinates
            found_ref = True
            ref_pdb = pdb
            print('Reference set to {}'.format(pdb))
            break
        else:
            print('\tSkipping {}, double-check file contents...'.format(pdb))
    if ref_pdb is None:
        raise ValueError('Could not find reference coordinates, double check PDB files!')
    print('Getting coordsets...')
    [super_ensemble.addCoordset(ens[1].getCoordsets()) for ens in sorted(pdb_dict.items()) if ens
is not None] # Add all coordinates to super ensemble
    if iterpose:
        print('Superposing...')
        super_ensemble.iterpose() # Superpose super ensemble
    return(ref_pdb,super_ensemble,lookup_list)

def calculate_super_ensemble_distance_contact_matrix(super_ensemble, reference, contacts=True,
vdw_frac=0.2, special_resi_dict=None, run_name='run', out_dir=None, verbose=False, ens_idx=None):
    """
    Calculates either a distance matrix or a contact matrix for all atoms in each model in the
    super-ensemble (i.e., with dimensions n_coordsets x n_atoms x n_atoms).
    Input:
        super_ensemble: A prody.Ensemble object.
        reference: A prody.Selection object which matches super_ensemble
        contacts: If False, returns raw distance matrices. If True, binarizes distance matrices
        to produce contact graph.
        vdw_frac: Percent of the sum of van der Waals radii for two atoms to add in declaring a
        contact. Default is 20%.
        special_resi_dict: A dictionary of dictionaries, where each key refers to a dictionary
        for a special residue name, which in turn keys the atoms in that residue and their associated the
        van der Waals radii of all atoms in that residue. For example,
        {'ALA':{'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88}, ...}.
        run_name: A descriptive title
        out_dir: If a string specifying a path to a directory is provided, things like plots will
        be saved there.
        verbose: If True, say more.
        ens_idx: Can be used to denote the integer-valued index of an ensemble in the super-
        ensemble if breaking things up for parallelism.
    Output:
        ens_idx (conditional): If an integer-valued index is provided in input, it will be
        returned.
        super_ensemble_distance_matrix: An np.ndarray. If contacts was set to True, this matrix
        will be a contact matrix with n_coordsets x n_residues x n_residues dimensions; if contacts was
        set to False, it will be a distance matrix with n_coordsets x n_atoms x n_atoms dimensions. If it
        is a contact matrix, [i, j, k] will be an integer corresponding to the number of atomic contacts
        between residues j and k in model i of the super ensemble. If it is a distance matrix, [i, j, k]
        will be the distance in Angstrom between atoms j and k in model i of the super-ensemble.
    """
    if verbose and type(ens_idx) is int:
        print('\tCalculating contact matrix for ensemble {}'.format(ens_idx))
    elif verbose:
        print('Calculating contact matrices over super-ensemble.')
    atom_names = reference.getNames()
    resi_names = reference.getResnames()
    resi_numbers = reference.getResnums()
    resi_indices = reference.getResindices()
    unique_resis = [resi_numbers[i] for i in sorted(np.unique(resi_numbers,
return_index=True)[1])] # Creates list of unique residues, preserving order.

```

```

n_resis = len(unique_resis)
n_atoms = reference.numAtoms()

if isinstance(super_ensemble, prody.ensemble.ensemble.Ensemble):
    coord_set_iterable = super_ensemble.iterCoordsets()
    n_coordsets = super_ensemble.numCoordsets()
elif isinstance(super_ensemble, prody.ensemble.conformation.Conformation):
    coord_set_iterable = [super_ensemble.getCoords()]
    n_coordsets = 1
elif isinstance(super_ensemble, np.ndarray):
    coord_set_iterable = [super_ensemble]
    n_coordsets = 1
elif isinstance(super_ensemble, list):
    if type(super_ensemble[0]) not in (prody.ensemble.conformation.Conformation, np.ndarray):
        raise ValueError('super_ensemble passed as a list with invalid elements of type
{}'.format(type(super_ensemble)))
    elif all([isinstance(ens, prody.ensemble.conformation.Conformation) for ens in
super_ensemble]):
        coord_set_iterable = (ens.getCoords() for ens in super_ensemble)
    elif all([isinstance(ens, np.ndarray) for ens in super_ensemble]):
        coord_set_iterable = super_ensemble
    n_coordsets = len(super_ensemble)
else:
    raise ValueError('{} is an invalid super_ensemble format!'.format(type(super_ensemble)))

if contacts: # Return contact counts per residue
    # Data for binarizing the contact matrix from Table I, column II of:
    # Tsai, J., Taylor, R., Chothia, C. & Gerstein, M. The packing density in proteins:
standard radii and volumes. J Mol Biol 290, 253-266 (1999).
    # Note that OXT distance is not explicitly provided in this paper.
    max_contact_dist = 1.88
    atom_radii = {}
    atom_rad_ii['ALA']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88}

atom_rad_ii['ARG']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'CD':1.88,
'NE':1.64, 'CZ':1.61, 'NH1':1.64, 'NH2':1.64}

atom_rad_ii['ASN']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.61, 'OD1':1.42
, 'ND2':1.64}

atom_rad_ii['ASP']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.61, 'OD1':1.42
, 'OD2':1.46}
    atom_rad_ii['CYS']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'SG':1.77}

atom_rad_ii['GLN']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'CD':1.61,
'OE1':1.42, 'NE2':1.64}

atom_rad_ii['GLU']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'CD':1.61,
'OE1':1.42, 'OE2':1.46}
    atom_rad_ii['GLY']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46}

atom_rad_ii['HIS']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.61, 'ND1':1.88
, 'CD2':1.64, 'CE1':1.88, 'NE2':1.64}

atom_rad_ii['ILE']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG1':1.88, 'CG2':1.8
8, 'CD1':1.88}

atom_rad_ii['LEU']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'CD1':1.88
, 'CD2':1.88}

atom_rad_ii['LYS']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'CD':1.88,
'CE':1.88, 'NZ':1.64}

atom_rad_ii['MET']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.88, 'SD':1.77,
'CE':1.88}

atom_rad_ii['PHE']={'N':1.64, 'CA':1.88, 'C':1.61, 'O':1.42, 'OXT':1.46, 'CB':1.88, 'CG':1.61, 'CD1':1.76
, 'CD2':1.76, 'CE1':1.76, 'CE2':1.76, 'CZ':1.76}

```

```

atom_radii['PRO']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.88,'CD':1.88}
atom_radii['SER']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'OG':1.46}

atom_radii['THR']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'OG1':1.46,'CG2':1.88}

atom_radii['TRP']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'CD1':1.88,'NE1':1.64,'CE2':1.61,'CD2':1.61,'CE3':1.76,'CZ3':1.76,'CH2':1.76,'CZ2':1.76}

atom_radii['TYR']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG':1.61,'CD1':1.76,'CD2':1.76,'CE1':1.76,'CE2':1.76,'CZ':1.76,'OH':1.46}

atom_radii['VAL']={'N':1.64,'CA':1.88,'C':1.61,'O':1.42,'OXT':1.46,'CB':1.88,'CG1':1.88,'CG2':1.88}

if special_resi_dict is not None and type(special_resi_dict) == dict:
    atom_radii.update(special_resi_dict)
    for this_resi in special_resi_dict.values():
        for this_atom_vdw in this_resi.values():
            if this_atom_vdw > max_contact_dist:
                max_contact_dist = this_atom_vdw
max_contact_dist = 2 * max_contact_dist + (2 * max_contact_dist * vdw_frac)
super_ensemble_contact_matrix = np.zeros((n_coordsets,n_resis,n_resis),dtype='uint8')
for cset_idx, coord_set in enumerate(coord_set_iterable):
    for idx_i, idx_j in np.nditer(np.triu_indices(n_atoms)):
        dist = np.linalg.norm(coord_set[idx_i,:]-coord_set[idx_j,:])
        if dist > max_contact_dist:
            continue
        resi_idx_i, resi_idx_j = resi_indices[idx_i], resi_indices[idx_j]
        atom_name_i, atom_name_j = atom_names[idx_i], atom_names[idx_j]
        resi_name_i, resi_name_j = resi_names[idx_i], resi_names[idx_j]
        if dist <=
(atom_radii[resi_name_i][atom_name_i]+atom_radii[resi_name_j][atom_name_j]) + vdw_frac *
(atom_radii[resi_name_i][atom_name_i]+atom_radii[resi_name_j][atom_name_j]):
            if verbose:
                print('Coordset {} contact: {}-{}-{} vs. {}-{}-{}, distance
{}'.format(cset_idx, resi_idx_i, resi_name_i, atom_name_i, resi_idx_j, resi_name_j, atom_name_j,
dist))
                super_ensemble_contact_matrix[cset_idx, resi_idx_i, resi_idx_j] += 1
            super_ensemble_contact_matrix[cset_idx, :, :] +=
np.triu(super_ensemble_contact_matrix[cset_idx, :, :], 1).T
            if out_dir is not None and os.path.isdir(out_dir):
                with open(os.path.join(out_dir, '{}_ensemble-cm.pkl'.format(run_name)), 'wb') as f:
pickle.dump({'super_ensemble_contact_matrix':super_ensemble_contact_matrix,'lookup':unique_resis}
,f)

if type(ens_idx) is not int:
    return(super_ensemble_contact_matrix)
else:
    return(ens_idx,super_ensemble_contact_matrix)
else: # Return pairwise atomic distance matrix.
    super_ensemble_distance_matrix = np.zeros((n_coordsets,n_atoms,n_atoms),dtype='float16')
    for cset_idx, coord_set in enumerate(coord_set_iterable):
        for idx_i, idx_j in np.nditer(np.triu_indices(n_atoms)):
            super_ensemble_distance_matrix[cset_idx, idx_i, idx_j] =
np.linalg.norm(coord_set[idx_i,:]-coord_set[idx_j,:])
            super_ensemble_distance_matrix[cset_idx, :, :] +=
np.triu(super_ensemble_distance_matrix[cset_idx, :, :], 1).T
        if type(ens_idx) is not int:
            return(super_ensemble_distance_matrix)
        else:
            return(ens_idx, super_ensemble_distance_matrix)

def reduce_super_ensemble_distance_contact_matrix(super_ensemble_contact_matrix, binarize=True,
mode='mean', run_name='run', plot=False, out_dir=None):
    """
    Converts an atomic contact matrix with dimensions n_coordsets x n_residues x n_residues
    dimensions to an average contact matrix with dimensions n_residues x n_residues dimensions.
    Input:
        super_ensemble_contact_matrix: A contact or distance matrix stored as an np.ndarray with

```

dimensions `n_coordsets x n_atoms x n_atoms`. Can also be a list of pickled dictionaries, each with a 'super_ensemble_contact_matrix' object.

`binarize`: If True, the individual contact matrices in `super_ensemble_contact_matrix` will be binarized prior to averaging, i.e., the output matrices will be a contact probability matrix and the standard deviation over binarized contacts. If False, elements of the output matrices will report the average or standard deviation of the number of atomic contacts between two residues.

`mode`: Must be 'mean'. In the future, more reduction modes may be added.

`run_name`: A descriptive title.

`plot`: If True, draw the matrices.

`out_dir`: If not None and a valid path to a directory, plots, etc. will be saved there.

Output:

`full_mat_mean`: Mean over all contact matrices in the super-ensemble, with `n_residues x n_residues` elements.

`full_mat_std`: Standard deviation over all contact matrices in the super-ensemble, with `n_residues x n_residues` elements.

```

'''
    if not isinstance(mode,str) and mode not in ('mean',):
        raise ValueError('Specified mode \'{ }\' is invalid!'.format(mode))
    if not isinstance(super_ensemble_contact_matrix, np.ndarray) and
type(super_ensemble_contact_matrix) in (str, list) and not
isinstance(super_ensemble_contact_matrix[0],np.ndarray):
        if isinstance(super_ensemble_contact_matrix, str) and
os.path.isdir(super_ensemble_contact_matrix):
            pkl_list =
glob.glob(os.path.join(super_ensemble_contact_matrix,'*contact_matrix_chunk*.pkl'))
            elif isinstance(super_ensemble_contact_matrix, list) and all([os.path.isfile(s) for s in
super_ensemble_contact_matrix]):
                pkl_list = [s for s in super_ensemble_contact_matrix if os.path.splitext(s)[1] ==
'.pkl']
            if len(pkl_list) == 0:
                raise ValueError('No contact matrix chunk .pkl files found in specified directory!')
            chunk_list = sorted([(int(i) for i in item)+[idx] for idx,item in
enumerate([os.path.splitext(os.path.basename(pkl))[0].split('-')[-2:] for pkl in pkl_list])) #
[[chunk_idx, total_chunks, list_idx], ...]
            if not all([chunk[1] == chunk_list[0][1] for chunk in chunk_list]):
                raise ValueError('Chunk numbering is not consistent; different numbers of chunks
detected across files.')
            n_chunks = chunk_list[0][1]
            chunk_idx_list = [chunk[0] for chunk in chunk_list]
            pkl_idx_list = [chunk[2] for chunk in chunk_list]
            chunk_start, chunk_end = min(chunk_idx_list), max(chunk_idx_list)
            if (chunk_start == 0 and chunk_idx_list != list(range(0,chunk_end))) or (chunk_start == 1
and chunk_idx_list != list(range(1,chunk_end+1))):
                raise ValueError('Chunk list is inconsistent with corresponding range:\nDetected:
{}'.format(list(range(chunk_end+1))))
            print('Found a series of {} pickles.'.format(n_chunks))
            full_mat = []
            for chunk_idx in chunk_idx_list:
                if chunk_idx_list[0] == 1:
                    chunk_idx -= 1
                this_pickle = pkl_list[pkl_idx_list[chunk_idx]]
                print('Loading data from {}'.format(os.path.split(this_pickle)[1]),end=' ')
                with open(this_pickle,'rb') as f:
                    this_chunk = pickle.load(f)['super_ensemble_contact_matrix']
                if not isinstance(this_chunk,np.ndarray):
                    raise ValueError('Object loaded from dictionary for chunk {} is not a numpy
array!'.format(chunk_idx))
                print('Found numpy ndarray with dimensions {}'.format(this_chunk.shape))
                if binarize:
                    full_mat.append(this_chunk.astype('bool'))
                else:
                    full_mat.append(this_chunk)
            full_mat = np.vstack(full_mat)
            print('Successfully reconstructed contact matrix from pickled data.')
            elif isinstance(super_ensemble_contact_matrix, list) and
isinstance(super_ensemble_contact_matrix[0],np.ndarray):
                full_mat = np.vstack(super_ensemble_contact_matrix)
                if binarize:
'''

```

```

        full_mat = full_mat.astype('bool')
    elif isinstance(super_ensemble_contact_matrix,np.ndarray):
        full_mat = super_ensemble_contact_matrix
        if binarize:
            full_mat = full_mat.astype('bool')
    else:
        raise ValueError('super_ensemble_contact_matrix format is invalid!')
mode = mode.lower()
if mode == 'mean':
    print('Taking mean of contact matrices...')
    full_mat_mean = full_mat.mean(axis=0)
    full_mat_std = full_mat.std(axis=0)
    if plot:
        if binarize:
            vmin = full_mat_mean.min()
            vmax = full_mat_mean.max()
            label='contact probability'
        else:
            vmin=np.triu(full_mat_mean,2).min()
            vmax=np.triu(full_mat_mean,2).max()
            label='num. contacts'
        figA,axA=plt.subplots(1,2,figsize=(25,15))
        p=axA[0].matshow(full_mat_mean, cmap=plt.cm.afmhot, vmin=vmin, vmax=vmax)
        divA = make_axes_locatable(axA[0])
        caxA = divA.append_axes('right',size='5%',pad=0.05)
        cbarA = plt.colorbar(p,cax=caxA,label='mean of {}'.format(label))
        axA[0].set_xlabel('residue i')
        axA[0].set_ylabel('residue j')
        axA[0].set_title('mean(cm)')
        q=axA[1].matshow(full_mat_std, cmap=plt.cm.afmhot, vmin=vmin, vmax=vmax)
        divB = make_axes_locatable(axA[1])
        caxB = divB.append_axes('right',size='5%',pad=0.05)
        cbarB = plt.colorbar(q,cax=caxB,label='Std. dev. of {}'.format(label))
        axA[1].set_xlabel('residue i')
        axA[1].set_ylabel('residue j')
        axA[1].set_title('std(cm)')
        if out_dir is not None:
            try:
plt.savefig(os.path.join(out_dir, '{}_secm_{}.eps'.format(run_name,mode)),transparent=True,format='eps')

plt.savefig(os.path.join(out_dir, '{}_secm_{}.png'.format(run_name,mode)),transparent=True,format='png')

                plt.close()
            except:
                print('Failed to save figure! Moving on.')
            else:
                plt.show()
                plt.close()
            if out_dir is not None and isinstance(out_dir,str) and os.path.isdir(out_dir):
                with open(os.path.join(out_dir, '{}_secm_bin-
{}_pk1'.format(run_name,str(binarize).lower()), 'wb') as f:
                    pickle.dump({'secm_mean':full_mat_mean,'secm_std':full_mat_std},f)
                return(full_mat_mean, full_mat_std)
            else:
                raise ValueError('Currently, only \'mean\' is supported as a contact matrix reduction
mode.')
```

def build_nm(pdb, selection_string=None, model='anm', n_modes=10, gamma_func_file=None, run_name='run', out_dir=None):

```

    """
    Uses ProDy to build either a Guassian or anharmonic network model for a model in a specified
    PDB file.
    Input:
        pdb: A path to a PDB file.
        selection_string: selection_string: A ProDy atom selection string. If None, no selection
        will be performed. Note that \'protein\' and \'heavy\' are already selected by default (i.e.,

```

```

only non-hydrogen protein atoms). For formatting information, see
http://prody.csb.pitt.edu/manual/reference/atomic/select.html.
    model: A string specifying the type of network model to calculate, either 'anm' for
anharmonic network model, or 'gnm' for Gaussian network model.
    n_modes: An integer specifying the number of modes to calculate for the network model.
    gamma_func_file: If not None and a valid path to a Python file with a proper custom
distance function, ProDy will use it to weight interactions based on distance.
    run_name: A descriptive title.
    out_dir: If not None and a path to a valid directory, will pickle and output NMD files
there.
'''
    if gamma_func_file is not None:
        if os.path.isfile(gamma_func_file):
            gamma_func_base = os.path.splitext(os.path.basename(gamma_func_file))[0]
            try:
                my_gamma = __import__(gamma_func_base)
            except:
                print('Failed to import custom gamma function! Make sure it\'s in your path.')
                raise
            if not hasattr(my_gamma, 'gammaDistanceDependent'):
                raise ValueError('Could not find gammaDistanceDependent in supplied
gamma_func_file!')
            if not hasattr(my_gamma, 'getCutoff'):
                raise ValueError('Could not find getCutoff in supplied gamma_func_file!')
        model = model.lower()
        if selection_string is not None and isinstance(selection_string, str):
            nm = {'prot':prody.parsePDB(pdb).protein.heavy.select(selection_string)}
        else:
            nm = {'prot':prody.parsePDB(pdb).protein.heavy}
        nm.update({'atom_names':nm['prot'].getNames(), 'resi_nums':nm['prot'].getResnums(),
'resi_indices':nm['prot'].getResindices()})
        print('Building {}'.format(model.upper()))
        if model == 'anm':
            nm.update({'atom_names':np.repeat(nm['prot'].getNames(),3),
'resi_nums':np.repeat(nm['prot'].getResnums(),3),
'resi_indices':np.repeat(nm['prot'].getResindices(),3)})
            nm['anm']=prody.ANM(run_name)
            if gamma_func_file is None:
                nm['anm'].buildHessian(nm['prot'])
            else:
                print('Using custom gamma function for distance-dependence of force constant...')
                nm['anm'].buildHessian(nm['prot'], gamma=my_gamma.gammaDistanceDependent,
cutoff=my_gamma.getCutoff())
        elif model == 'gnm':
            nm['gnm']=prody.GNM(run_name)
            if gamma_func_file is None:
                nm['gnm'].buildKirchhoff(nm['prot'])
            else:
                nm['gnm'].buildKirchhoff(nm['prot'], cutoff=my_gamma.getCutoff())
        else:
            raise ValueError('Supported network model modes are \'anm\' and \'gnm\'')
        nm[model].calcModes(n_modes=n_modes,zeros=False)
        nm['{}_cov'.format(model)] = nm[model].getCovariance()
        nm['{}_corr'.format(model)] = multiconf_utilities.cov_to_corr(nm['{}_cov'.format(model)])
        if model == 'anm':
            nm['{}_cov_rms'.format(model)] = multiconf_utilities.calc_rms(nm['{}_cov'.format(model)])
            nm['{}_corr_rms'.format(model)] =
multiconf_utilities.calc_rms(nm['{}_corr'.format(model)])
        if type(out_dir) == str and os.path.isdir(out_dir):
            try:
                prody.writeNMD(os.path.join(out_dir, '{}_{}_{}_modes.nmd'.format(run_name, model,
n_modes)), nm[model], nm['prot'])
            except:
                raise
            print('Failed to output NMD file!')
            with open(os.path.join(out_dir, '{}_{}.pkl'.format(run_name,model)), 'wb') as f:
                pickle.dump(nm, f)
    return(nm)

```

```

def compare_modes(mat_dict, n_modes=10, cc_threshold=0.4, start_diag=1, run_name='run',
plot=True, out_dir=None):
    """
    Compare modes between all two-dimensional matrices keyed in mat_dict and plot/pickle/write
    output to text. This is some ugly ass code. Do not learn from this example.
    First, computes/returns basic stats, performs eigendecomposition of all matrices. Second,
    compares upper triangle correlation/cosines between matrices. Third, performs all combinations of
    comparisons of modes between matrices.
    Input:
        mat_dict: A dictionary of two-dimensional np.ndarray objects, keyed by name. All matrices
        should have the same number of elements.
        n_modes: An integer specifying the number of top modes from the eigendecomposition of
        each matrix to compare.
        cc_threshold: A float; the correlation coefficient threshold above which reconstruction
        plots will be created.
        start_diag: An integer. When comparing upper triangles of matrices, start with this
        diagonal, where 0 indicates the primary diagonal. The default excludes the primary diagonal.
        plot: If True, make lots of nice plots.
        out_dir: A string specifying the location to which to save plots and output text. If
        None, nothing will be written.
    Output:
        evc_comp: A multi-level dictionary, with the following key/value pairs:
            'cc' or 'cos': Specifies the comparison type; either correlation coefficient or
            cosine
            'mat_dict_i-mat_dict_j': A key specifying the comparison in question, where
            mat_dict_i and mat_dict_j are a pair of keys from mat_dict.
            np.ndarray((n_modes,n_modes)): A matrix of correlations or cosines for every
            pairwise combination of the top n_modes eigenvectors from mat_dict_i and mat_dict_j.
    """
    if out_dir is not None:
        if not isinstance(out_dir, str) or not os.path.isdir(out_dir):
            raise ValueError('Invalid output directory specified.')
        fo = open(os.path.join(out_dir, '{}_mode-comparison.txt'.format(run_name)), 'w')
    if 'resi_nums' not in mat_dict.keys() or 'atom_names' not in mat_dict.keys():
        raise ValueError('Could not find resi_nums or atom_names in mat_dict.')
    elif len(mat_dict['resi_nums']) != len(mat_dict['atom_names']):
        raise ValueError('resi_nums and atom_names must have the same number of elements.')
    resi_atom_list = [{'{}-{}'.format(r,a) for r,a in zip(mat_dict['resi_nums'],
mat_dict['atom_names'])}]
    shapes = [v.shape for k,v in mat_dict.items() if 'corr' in k]
    if not all([i==j for i,j in it.combinations(shapes,2)]):
        print(shapes)
        raise ValueError('All matrices for comparison must be of the same size!')
    n_items = len([k for k in mat_dict.keys() if 'corr' in k])
    n_comps = len(list(it.combinations(range(n_items),2)))
    if plot:
        cmap=plt.cm.RdBu_r
        vmin,vmax = -1,1
        plt_stuff_a = {}
        figA,axA=plt.subplots(nrows=1,ncols=n_items,squeeze=False,figsize=(30,7.5))
    msg='Matrix triu stats:'
    if out_dir is not None:
        fo.write(msg+'\n')
    print(msg)
    triu_dict={}
    eig_dict={}
    i = -1
    for k,v in sorted(mat_dict.items()):
        if 'corr' not in k:
            continue
        else:
            i+=1
            v_triu = v[np.triu_indices_from(v,start_diag)].flatten()
            triu_dict.update({k:v_triu})
            evals, evecs = np.linalg.eigh(v)
            idx = evals.argsort()[::-1]
            eig_dict.update({k: {'evals':evals[idx], 'evecs':evecs[:,idx]}})
            msg = '\t{:} mean {:.2f} +/- {:.2f}, min: {:.2f}, max: {:.2f}'.format(k, np.mean(v_triu),
np.std(v_triu), np.min(v_triu), np.max(v_triu))

```

```

if out_dir is not None:
    fo.write(msg+'\n')
print(msg)
if plot:
    plt_stuff_a[k+'_plt']=axA[0][i].matshow(v, cmap=cmap, vmin=vmin, vmax=vmax)
    axA[0][i].set_title('{:} matrix\n'.format(run_name, k.split('_')[0]))
    axA[0][i].set_xlabel('atom i')
    axA[0][i].set_ylabel('atom j')
    try:
        j = [int(n) for n in axA[0][i].get_xticks()[:-1]]
        x = [resi_atom_list[n] for n in j]
    except:
        print('Matrix shape: {} \n resi_atom_list length: {}'.format(v.shape,
len(resi_atom_list)))
        raise
        axA[0][i].set_xticklabels(x)
        axA[0][i].set_yticklabels(x)
        plt_stuff_a[k+'_div'] = make_axes_locatable(axA[0][i])
        plt_stuff_a[k+'_cax'] = plt_stuff_a[k+'_div'].append_axes('right', size='5%', pad=0.05)
        plt_stuff_a[k+'_cbar'] =
plt.colorbar(plt_stuff_a[k+'_plt'], cax=plt_stuff_a[k+'_cax'])
if plot:
    if out_dir is not None:
        plt.savefig(os.path.join(out_dir, '{}_raw-mats.eps'.format(run_name)),
transparent=True, close=True, format='eps')
        plt.savefig(os.path.join(out_dir, '{}_raw-mats.png'.format(run_name)),
transparent=True, close=True, format='png')
    else:
        plt.show()

msg = 'Calculating overall triu matrix correlation coefficients...'
if out_dir is not None:
    fo.write(msg+'\n')
print(msg)
if plot:
    figC, axC=plt.subplots(nrows=1, ncols=n_comps, squeeze=False, figsize=(30,15))
    for i, keys in enumerate(it.combinations(triu_dict.keys(),2)):
        key_a, key_b = keys
        msg = '\t{:} vs. {:} cc {:.2f}, cos {:.2f}'.format(key_a, key_b,
np.corrcoef(triu_dict[key_a], triu_dict[key_b])[0,1],
multiconf_utilities.triu_cosine(triu_dict[key_a], triu_dict[key_b]))
        if out_dir is not None:
            fo.write(msg+'\n')
print(msg)
if plot:
        axC[0][i].plot(triu_dict[key_a], triu_dict[key_b], 'k.', alpha=0.2)
        axC[0][i].set_xlabel(key_a)
        axC[0][i].set_ylabel(key_b)
if plot:
    if out_dir is not None:
        plt.savefig(os.path.join(out_dir, '{}_triu-scatter.eps'.format(run_name)),
transparent=True, close=True, format='eps')
        plt.savefig(os.path.join(out_dir, '{}_triu-scatter.png'.format(run_name)),
transparent=True, close=True, format='png')
    else:
        plt.show()

msg = 'Calculating CC, cosine matrices for top 10 eigenmodes...'
if out_dir is not None:
    fo.write(msg+'\n')
print(msg)
if plot:
    plot_stuff_c = {}
    evec_comp = {'cc': {}, 'cos': {}}
    for kva, kvb in it.combinations(eig_dict.items(),2):
        key_a = kva[0]
        evecs_a = kva[1]['evecs']
        evals_a = kva[1]['evals']
        key_b = kvb[0]

```

```

    evecs_b = kvb[1]['evecs']
    evals_b = kvb[1]['evals']
    [evec_comp[comp_type].update({'{}-{}'.format(key_a, key_b): np.ndarray((n_modes, n_modes))})
for comp_type in ('cc', 'cos')]
    msg='\t{} vs. {}...'.format(key_a, key_b)
    if out_dir is not None:
        fo.write(msg+'\n')
    print(msg)
    for i,j in np.ndindex(n_modes, n_modes):
        evec_ai = np.zeros_like(evecs_a)
        evec_ai[:,i] = evecs_a[:,i]
        eval_ai = np.diag(np.zeros_like(evals_a))
        eval_ai[i,i] = evals_a[i]
        evec_bj = np.zeros_like(evecs_b)
        evec_bj[:,j] = evecs_b[:,j]
        eval_bj = np.diag(np.zeros_like(evals_b))
        eval_bj[j,j] = evals_b[j]
        evec_comp['cc']['{}-{}'.format(key_a, key_b)][i,j] = np.corrcoef(evec_ai[:,i],
evec_bj[:,j])[0,1]
        evec_comp['cos']['{}-{}'.format(key_a, key_b)][i,j] =
np.dot(evec_ai[:,i], evec_bj[:,j])/np.linalg.norm(evec_ai[:,i])/np.linalg.norm(evec_bj[:,j])
        msg = '\t\tMode {} vs. mode {}, cc {:.2f}, cos {:.2f}'.format(i,j,
evec_comp['cc']['{}-{}'.format(key_a, key_b)][i,j], evec_comp['cos']['{}-{}'.format(key_a,
key_b)][i,j])
        if out_dir is not None:
            fo.write(msg+'\n')
        print(msg)
        if plot and abs(evec_comp['cc']['{}-{}'.format(key_a, key_b)][i,j]) > cc_threshold:
            print('\t\t\tCC is greater than threshold of {}, plotting
reconstructions...'.format(cc_threshold))
            recon_a = np.dot(evec_ai, np.dot(eval_ai, evec_ai.T))
            recon_b = np.dot(evec_bj, np.dot(eval_bj, evec_bj.T))
            plot_stuff_c['fig_{}-{}-{}-{}'.format(key_a, key_b, i, j)], plot_stuff_c['ax_{}-{}-
{}-{}'.format(key_a, key_b, i, j)] = plt.subplots(nrows=1, ncols=3, squeeze=False, figsize=(30, 7.5))
            plot_stuff_c['scatter_{}-{}-{}-{}'.format(key_a, key_b, i, j)] =
plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][0].plot(evec_ai[:,i], evec_bj[:,j], 'k.', alpha=0.5)
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][0].annotate('r =
{:.2f}'.format(evec_comp['cc']['{}-{}'.format(key_a, key_b)][i,j]),
xy=(evec_ai[:,i].min()+(evec_ai[:,i].min()*0.10), evec_bj[:,j].max()-
(evec_bj[:,j].max()*0.10)), horizontalalignment='left')
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][0].set_xlabel(key_a)
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][0].set_ylabel(key_b)
            plot_stuff_c['recon_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][1].matshow(recon_a, cmap=cmap, vmin=vmin, vmax=vmax)
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][1].set_xlabel('atom i')
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][1].set_ylabel('atom j')
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][1].set_title('{} mode
{} reconstruction'.format(key_a, i))
            plot_stuff_c['div_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)] =
make_axes_locatable(plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][1])
            plot_stuff_c['cax_{}-{}-{}-{}-
{}'.format(key_a, i)].append_axes('right', size='5%', pad=0.05)
            plot_stuff_c['cbar_{}-{}-{}-{}-
{}'.format(key_a, i)], cax=plot_stuff_c['cax_{}-{}-{}-{}-
{}'.format(key_a, i)])
            plot_stuff_c['recon_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][2].matshow(recon_b, cmap=cmap, vmin=vmin, vmax=vmax)
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][2].set_xlabel('atom i')
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][2].set_ylabel('atom j')
            plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][2].set_title('{} mode
{} reconstruction'.format(key_b, j))
            plot_stuff_c['div_{}-{}-{}-{}-
{}'.format(key_b, j)] =
make_axes_locatable(plot_stuff_c['ax_{}-{}-{}-{}-
{}'.format(key_a, key_b, i, j)][0][2])
            plot_stuff_c['cax_{}-{}-{}-{}-
{}'.format(key_b, j)].append_axes('right', size='5%', pad=0.05)
            plot_stuff_c['cbar_{}-{}-{}-{}-
{}'.format(key_b, j)], cax=plot_stuff_c['cax_{}-{}-{}-{}-
{}'.format(key_b, j)])
        if out_dir is not None:
            plt.savefig(os.path.join(out_dir, '{}_{}_{}_{}_{}_mode-

```

```

plot.eps'.format(run_name, key_a, key_b, i, j)),transparent=True, close=True, format='eps')
    plt.savefig(os.path.join(out_dir,'{}_{}_{}-{}_{}_mode-
plot.png'.format(run_name, key_a, key_b, i, j)),transparent=True, close=True, format='png')
    else:
        plt.show()
    if plot:
        plt_stuff_b = {}
        figB,axB=plt.subplots(nrows=2,ncols=n_comps,squeeze=False,figsize=(30,15))
        for i,comp_type_item in enumerate(avec_comp.items()):
            comp_type_key, comp_type_dict = comp_type_item
            for j, comp_item in enumerate(comp_type_dict.items()):
                comp_key, comp_mat = comp_item
                plt_stuff_b['{}_-'
{}_plt'.format(i,j)]=axB[i][j].matshow(comp_mat,cmap=cmap,vmin=vmin,vmax=vmax)
                axB[i][j].set_xlabel(comp_key.split('-')[0]+' eigenvector index')
                axB[i][j].set_ylabel(comp_key.split('-')[1]+' eigenvector index')
                axB[i][j].set_title('{}: {}'.format(comp_type_key, comp_key))
                plt_stuff_b['{}_{}_div'.format(i,j)] = make_axes_locatable(axB[i][j])
                plt_stuff_b['{}_{}_cax'.format(i,j)] = plt_stuff_b['{}_-'
{}_div'.format(i,j)].append_axes('right',size='5%',pad=0.05)
                plt_stuff_b['{}_{}_cbar'.format(i,j)] = plt.colorbar(plt_stuff_b['{}_-'
{}_plt'.format(i,j)],cax=plt_stuff_b['{}_{}_cax'.format(i,j)])
                if out_dir is not None:
                    plt.savefig(os.path.join(out_dir,'{}_mode-comparison.eps'.format(run_name)),
transparent=True, close=True, format='eps')
                    plt.savefig(os.path.join(out_dir,'{}_mode-comparison.png'.format(run_name)),
transparent=True, close=True, format='png')
                    plt.close('all')
                else:
                    plt.show()
            if out_dir is not None:
                fo.close()
        return(avec_comp)

def cor_vs_mi(M, R, start_diag=1, run_name='run', plot=True, out_dir=None):
    """
    Compare a mutual information matrix to an equivalent correlation coefficient matrix, possibly
    plotting the results.
    Input:
        M: An np.ndarray object specifying a mutual information matrix.
        R: An np.ndarray object specifying a correlation coefficient matrix.
        start_diag: An integer specifying the diagonal to start with in the comparison.
    Specifying 1 excludes elements of the primary diagonal.
        run_name: A descriptive title
        plot: If True, draw the scatter plot for M and R with expectation line.
        out_dir: If a string specifying a valid directory, save the plot there.
    Output:
        .. None
        ...
    if M.shape != R.shape:
        raise ValueError('M and R must have an equal number of elements!')
    M_triu = M[np.triu_indices_from(M,start_diag)]
    R_triu = R[np.triu_indices_from(R,start_diag)]
    r = np.corrcoef(R_triu,M_triu)[0,1]
    if plot:
        x = np.linspace(0,1,1000)
        m = -0.5*np.log(1-x**2)
        figA,axA=plt.subplots(nrows=1,ncols=1,squeeze=False,figsize=(10,10))
        axA[0][0].plot(R_triu, M_triu, 'k.', alpha=0.3)
        axA[0][0].plot(x,m,'r',alpha=0.5)
        axA[0][0].set_xlim(0,1)
        axA[0][0].annotate('pearsonr =
{:.2f}'.format(r),xy=(0.05,M_triu.max()),horizontalalignment='left')
        axA[0][0].set_title('{}: R vs. I'.format(run_name))
        axA[0][0].set_ylabel('Mutual information (bits)')
        axA[0][0].set_xlabel('Correlation coefficient')
        if isinstance(out_dir, str) and os.path.isdir(out_dir):
            figA.savefig(os.path.join(out_dir,'{}_r-vs-
mi.eps'.format(run_name)),transparent=True,close=True,verbose=True,format='eps')

```

```

    else:
        plt.show()
    plt.close()
return

```

```

def super_ensemble_covariance_convergence(super_ensemble, lookup_list=None, subsample=100,
split=(False,3), plot=False, run_name='run', out_dir=None, verbose=True):
'''

```

Figure out just how much of a super-ensemble is needed to recapitulate half of the coordinate correlation coefficient matrix.

Input:

super_ensemble: A prody.Ensemble object.
lookup_list: A list of lists, with one entry per PDB file used to create the super-ensemble. Elements in each sublist are ['filename_base', file_index, model_index_in-file, model_index_all-files], where filename_base is a string corresponding to the basename of a given PDB file and the various indices are integers.

subsample: An integer greater than or equal to unity and less than the number of models in the super-ensemble if not operating in split mode or less than the number of models//2 if operating in split mode. If operating in split mode (i.e., split[0] is True), convergence calculations will be performed subsample times, evenly distributed over the trajectory. To sample at every point along the trajectory, set to None.

split: A tuple or list with two elements, (bool, int). If the first element is True, the trajectory order will be randomized and split into two chunks. Convergence will then be assessed based on comparison between first random chunk and the second random chunk, where the first chunk is sampled in increasingly large chunks and the second chunk is always full length. The second element of split specifies the number of times to repeat this procedure. If the first element of split is False, will assess convergence relative to the result obtained by including all models in the super-ensemble.

plot: If True, plot the trajectory convergence.

run_name: A string specifying a descriptive title.

out_dir: A string specifying a path to a directory in which things should be saved.

Output:

cov_fnorm_diff: An np.ndarray with dimensions subsample x split[1] (if split[0] is True) or n_coordsets (if split[0] is False). Each element is the Frobenius norm of the difference of the reference and subsampled covariance matrices.

corr_fnorm_diff: An np.ndarray with dimensions subsample x split[1] (if split[0] is True) or n_coordsets (if split[0] is False). Each element is the Frobenius norm of the difference of the reference and subsampled correlation coefficient matrices.

tau: A float returned if split[0] is True. A parameter derived from the Fisher z-transform which approaches unity in the case where sampling between first and second half of a randomized trajectory is independent.

```

def get_ens_bounds(lookup_list):
'''

```

Gets ensemble bounds from lookup_list if subsampling.

```

start = []
end = []
prev_entry = (0,-1)
for entry in lookup_list:
    if entry[3] == 0:
        start.append(entry[2])
        if entry[3] < prev_entry[1]:
            end.append(prev_entry[0])
        prev_entry = (entry[2],entry[3])
end.append(entry[2])
return(start,end)

```

```

def calculate_coordset_cov(coordset):
'''

```

Calculates a covariance matrix from a two-dimensional np.ndarray with n_models x (n_atoms x 3) elements.

Input:

coordset: An n_models x n_atoms x 3 np.ndarray.

Output:

cov: An (n_atoms x 3) x (n_atoms x 3) np.ndarray, where each element is the covariance between atom_i in dim_a vs atom_j in dim_b.

```

'''

```

```

if coordset.shape[0] < 3:

```

```

        raise ValueError('n_confs must be greater than three!')
    elif coordset.shape[1] < 3:
        raise ValueError('coordset must have more than 3 atoms!')
    cov = np.cov(coordset.reshape((coordset.shape[0], coordset.shape[1]*3)).T, bias=1)
    return(cov)

    if type(split) not in (list, tuple) or split[0] not in (True, False) or not
    isinstance(split[1],int) or split[1] < 1:
        raise ValueError('split must be a list or tuple with format (bool, int)!')
    n_coordsets = super_ensemble.numCoordsets()
    if split[0]:
        if subsample is not None and (not isinstance(subsample, int) or subsample < 1 or
        subsample > n_coordsets//2):
            raise ValueError('subsample must be an integer between 1 and n_coordsets//2.')
        cov_fnorm_diff = np.zeros((n_coordsets//2+1,split[1]))*np.nan
        corr_fnorm_diff = np.zeros((n_coordsets//2+1,split[1]))*np.nan
        ens_idx_list = list(range(n_coordsets))
        for j in range(split[1]): # Generate randomized lookup lists, split each into two groups,
        and store in dict.
            # Shuffle idx list and split into two halves
            ens_idx_list =
            random.sample(list(range(super_ensemble.numCoordsets())),super_ensemble.numCoordsets())
            ens_idx_list_this = ens_idx_list[:n_coordsets//2].copy()
            ens_idx_list_ref = ens_idx_list[n_coordsets//2:].copy()
            if verbose:
                print('Calculating reference matrices...')
            csets_ref = super_ensemble.getCoordsets(ens_idx_list_ref)
            cov_mat_ref = calculate_coordset_cov(csets_ref)
            corr_mat_ref = multiconf_utilities.cov_to_corr(cov_mat_ref)
            if verbose:
                print('Calculating matrices for full slice and comparing to reference...')
            csets_full = super_ensemble.getCoordsets(ens_idx_list_this)
            cov_mat_full = calculate_coordset_cov(csets_full)
            corr_mat_full = multiconf_utilities.cov_to_corr(cov_mat_full)
            cov_fnorm_diff[len(ens_idx_list_this),j] = np.linalg.norm(cov_mat_ref - cov_mat_full)
            corr_fnorm_diff[len(ens_idx_list_this),j] = np.linalg.norm(corr_mat_ref -
            corr_mat_full)
            if verbose:
                print('\tDifference covariance matrix norm = {:.2f}\n\tDifference correlation
                coefficient matrix norm = {:.2f}'.format(cov_fnorm_diff[-1,j],corr_fnorm_diff[-1,j]))
                if isinstance(subsample, int) and subsample > 0:
                    interval = int(len(ens_idx_list_this)/subsample)
                    if verbose:
                        print('Will evenly subsample convergence trajectory of length {} every {}
                        points.'.format(len(ens_idx_list_this),interval))
                    else:
                        interval = 1
                        if verbose:
                            print('Will sample convergence trajectory of length {} at every
                            point.'.format(len(ens_idx_list_this)))
                        for i in range(3,len(ens_idx_list_this),interval):
                            if verbose:
                                print('\tCalculating matrices for iteration {} slice [{}:] and comparing to
                                reference...'.format(j,i))
                                ens_idx_list_this_sub = ens_idx_list_this[:i]
                                csets_this = super_ensemble.getCoordsets(ens_idx_list_this_sub)
                                cov_mat_this = calculate_coordset_cov(csets_this)
                                corr_mat_this = multiconf_utilities.cov_to_corr(cov_mat_this)
                                cov_fnorm_diff[i,j] = np.linalg.norm(cov_mat_ref - cov_mat_this)
                                corr_fnorm_diff[i,j] = np.linalg.norm(corr_mat_ref - corr_mat_this)
                                if verbose:
                                    print('\t\tDifference covariance matrix norm = {:.2f}\n\t\tDifference
                                    correlation coefficient matrix norm = {:.2f}'.format(cov_fnorm_diff[i,j],corr_fnorm_diff[i,j]))
                                    tau = (corr_fnorm_diff[-1,:]**2 * len(ens_idx_list_ref)) / (3*corr_fnorm_diff[-1,:]**2 +
                                    2*(cov_mat_this.shape[0]-1)*cov_mat_this.shape[0])
                                if verbose:
                                    print('Value of tau for each repeat: {}'.format(tau))
                            if out_dir is not None:
                                if os.path.isdir(out_dir):

```

```

        pickle.dump({'cov_fnorm_diff':cov_fnorm_diff, 'corr_fnorm_diff':corr_fnorm_diff,
'tau':tau},open(os.path.join(out_dir,'{}_cov-corr-convergence.pkl'.format(run_name)), 'wb'))
    elif verbose:
        print('\nInvalid output directory specified, will not pickle data.')
    return(cov_fnorm_diff, corr_fnorm_diff, tau)
elif not split[0]:
    if verbose:
        print('Calculating reference covariance and correlation matrices over entire super-
ensemble...')
    # Initialize
    if lookup_list is None:
        raise ValueError('Valid lookup_list required for non-split mode.')
    if subsample is None:
        calc_range = range(3,n_coordsets)
    else:
        calc_range = get_ens_bounds(lookup_list)[1]
    cov_fnorm_diff = np.zeros(n_coordsets)*np.nan
    corr_fnorm_diff = np.zeros(n_coordsets)*np.nan
    # Get reference matrices
    csets_all = super_ensemble.getCoordsets()
    cov_mat_all = calculate_coordset_cov(csets_all)
    corr_mat_all = multiconf_utilities.cov_to_corr(cov_mat_all)
    for ens_idx in calc_range:
        if verbose:
            print('\n[{} / {}]'.format(ens_idx, n_coordsets))
        try:
            csets_slice = super_ensemble.getCoordsets(super_ensemble[:ens_idx])
        except:
            if verbose:
                print('Failed for slice [{}], skipping.'.format(ens_idx))
            cov_list.append(np.nan)
            continue
        cov_mat_slice = calculate_coordset_cov(csets_slice)
        corr_mat_slice = multiconf_utilities.cov_to_corr(cov_mat_slice)
        cov_fnorm_diff[ens_idx] = np.linalg.norm(cov_mat_all - cov_mat_slice)
        corr_fnorm_diff[ens_idx] = np.linalg.norm(corr_mat_all - corr_mat_slice)
        if verbose:
            print('Squared (covariance, correlation) difference matrix norm = ({} ,
{})\n'.format(cov_fnorm_diff[ens_idx], corr_fnorm_diff[ens_idx]))
        if plot:
            if verbose:
                print('Plotting...')
            fig = plt.figure()
            ax = fig.add_subplot(111)
            ax.plot(cov_fnorm_diff)
            ax.plot(corr_fnorm_diff)
    return(cov_fnorm_diff, corr_fnorm_diff)
else:
    raise ValueError('split[0] must be boolean!')

def plot_super_ensemble_covariance_convergence(traj_mat, data_label='data', title='Title',
tau_tuple=(None,None,None), out_dir=None, logy=True, logx=False):
    """
    Plots results from super_ensemble_covariance_convergence calculations.
    Input:
        traj_mat: An np.ndarray with dimensions subsample x repeats, where subsample is the
        number of samplings of a trajectory and repeats is the number of times the trajectory was
        randomly sampled.
        data_label: A string which will be included in various places to indicate the type of
        data stored in traj_mat.
        title: A string which will be used as a plot title.
        tau_tuple: A tuple (tau, d, n_r); the first element tau is a list of one or more floats,
        d is an integer number of data points (i.e., number of atoms * 3 dimensions), and n_r is an
        integer number of models in the reference slice.
        out_dir: If not None and a string specifying a valid directory, will save plots there.
        logy: If True, draw y-axis on log scale.
        logx: If True, draw x-axis on log scale.
    Output:
        reduced_mat: Like traj_mat, but with all np.nan elements removed.

```

```

...

def f_pred(tau, d, n_r, n_i):
    """
    Input:
        tau: Convergence target
        d: Number of observations (i.e., number of atoms * 3 dimensions)
        n_i: Number of models in first slice (independent var)
        n_r: Number of models in second slice (constant)
    Output:
        f: Predicted difference between matrices in first and second slice
    """
    f = np.sqrt(((d*(d-1))/((n_i/tau)-3))+((d*(d-1))/((n_r/tau)-3)))
    return(f)

# Tests
if len(tau_tuple) != 3:
    raise ValueError('tau_tuple must contain 3 elements only.')
if not isinstance(traj_mat, np.ndarray):
    raise ValueError('traj_mat must be a numpy array.')
# Get rid of NaNs
reduced = []
idx = []
for i in range(traj_mat.shape[0]):
    if any(np.isnan(traj_mat[i,:])):
        continue
    reduced.append((i, traj_mat[i,:]))
reduced_mat = np.vstack([np.hstack(m) for m in reduced])
reduced_mat[:,0] += 1
# Fit curve to average of data
xfull = np.linspace(0, reduced_mat[:,0].max(), 10000)
cf_opt, cf_cov =
curve_fit(multiconf_utilities.decay_func, reduced_mat[1:,0], reduced_mat[1:,1:].mean(axis=1))
cf_err = np.sqrt(np.diag(cf_cov))
cf_fit = [multiconf_utilities.decay_func(xi,*cf_opt) for xi in xfull]
# Plot
fig,ax=plt.subplots(nrows=1,ncols=1)
for i in range(1, reduced_mat.shape[1]):
    #ax.plot(reduced_mat[:,0], reduced_mat[:,i], 'k-', alpha=.3)
    ax.plot(reduced_mat[:,0], reduced_mat[:,i], 'k.')
    if None in tau_tuple or not isinstance(tau_tuple[0], list):
        ax.plot(xfull, multiconf_utilities.decay_func(xfull, cf_opt[0], cf_opt[1]), 'r-', label='')
    else:
        ax.plot(list(range(3, tau_tuple[2], 100)), [f_pred(tau_tuple[0], tau_tuple[1], tau_tuple[2],
n_i+1) for n_i in range(3, tau_tuple[2], 100)], 'r-', alpha=0.3)
        ax.annotate('<tau> = {:.2f}'.format(tau_tuple[0].mean()), xy=(reduced_mat[-
20,0], reduced_mat[3,:].mean()), horizontalalignment='left')
        if logx:
            ax.set_xscale('log')
        if logy:
            ax.set_yscale('log')
        ax.set_xlim((-1000, max(reduced_mat[:,0]+1000)))
        ax.set_xlabel('Models included in slice')
        ax.set_ylabel('|| {}_full - {}_slice ||_frob'.format(data_label))
        ax.set_title('{}'.format(title))
        ax.grid()
        if out_dir is not None:
            fig.savefig(os.path.join(out_dir, '{}_convergence.eps'.format(data_label)), transparent=True, close=
True, verbose=True, format='eps')
            fig.savefig(os.path.join(out_dir, '{}_convergence.png'.format(data_label)), transparent=True, close=
True, verbose=True, format='png')
        else:
            plt.show()
    return(reduced_mat)

def super_ensemble_pca(super_ensemble, ref_pdb, lookup_list, project=True, run_name='run',

```

```

max_var=0.70, plot=False, out_dir=None):
    """
    Perform principal components analysis (PCA) on super-ensemble using ProDy.
    Input:
        ref_pdb: A prody.Selection object, which serves as a reference object for the
        super_ensemble.
        super_ensemble: A prody.Ensemble object, which has all coordinates for all models in the
        super_ensemble.
        lookup_list: A list of lists, with one entry per PDB file used to create the super-
        ensemble. Elements in each sublist are ['filename_base', file_index, model_index_in-file,
        model_index_all-files], where filename_base is a string corresponding to the basename of a given
        PDB file and the various indices are integers.
        project: If True, calculate the projection of models along different modes using ProDy.
        run_name: A descriptive string.
        max_var: If cumulative variance of some number of modes exceeds this threshold, include
        no more modes.
        plot: If True, draw some plots.
        out_dir: If not None and a string describing the path to a valid directory, an NMD file
        will be written.
    Output:
        pca_dict: A dictionary with a variety of objects related to the PCA calculations having
        the following keys:
            'pca': The ProDy PCA object.
            'pca_cov': An np.ndarray covariance matrix with dimensions (3 x n_atoms) x (3 x
            n_atoms).
            'pca_cov_rms': An np.ndarray covariance matrix with dimensions n_atoms x n_atoms.
            'pca_corr': An np.ndarray correlation coefficient matrix with dimensions (3 x
            n_atoms) x (3 x n_atoms).
            'pca_corr_rms': An np.ndarray correlation coefficient matrix with dimensions n_atoms
            x n_atoms.
            'proj_cov': If project is True,
            'proj_corr': If project is True,
            'proj_corr_suppress': If project is True,
            ...
    """
    print('Performing PCA on super-ensemble...')
    pca = prody.PCA(run_name)
    pca.performSVD(super_ensemble)
    cum_var = 0
    for mode_idx, mode in enumerate(pca):
        var = prody.calcFractVariance(mode)
        if cum_var + var > max_var:
            mode_idx -= 1
            break
        else:
            cum_var += var
    print('Top 3 modes account for {0:.0%} of variance.'.format(sum(prody.calcFractVariance(mode)
    for mode in pca[0:3])))
    print('{0:.0%} of variance explained by top {1:d} modes.\nCalculating covariance and
    correlation matrices...'.format(cum_var, mode_idx))
    pca_cov = pca.getCovariance()
    pca_cov_rms = multiconf_utilities.calc_rms(pca_cov)
    pca_corr = multiconf_utilities.cov_to_corr(pca_cov)
    pca_corr_rms = multiconf_utilities.calc_rms(pca_corr)
    pca.calcModes()
    pca_dict = pca_dict = {'pca':pca, 'pca_cov':pca_cov, 'pca_cov_rms':pca_cov_rms,
    'pca_corr':pca_corr, 'pca_corr_rms':pca_corr_rms}
    if out_dir is not None:
        try:
            print('Writing NMD file...')
            prody.writeNMD(os.path.join(out_dir, '{}_pca_{}-
            modes.nmd'.format(run_name, pca.numModes())), pca, ref_pdb)
        except:
            print('Failed to output NMD file!')
    if project:
        print('Calculating projection of models along covariance and correlation modes...')
        proj_cov=prody.calcProjection(super_ensemble, pca)
        pca_corr_evals, pca_corr_evecs = np.linalg.eigh(pca_corr)
        pca_corr_evals_abs=np.abs(pca_corr_evals)
        idx = pca_corr_evals_abs.argsort()[::-1] # Sort from greatest to least

```

```

pca_corr_evals_abs_sort=pca_corr_evals_abs[idx]
pca_corr_evals_sort = pca_corr_evals[idx]
pca_corr_evecs_sort = pca_corr_evecs[:,idx]
deviations = super_ensemble.getDeviations()
if deviations.ndim == 3:
    deviations = deviations.reshape((deviations.shape[0], deviations.shape[1] * 3))
elif deviations.ndim == 2:
    deviations = deviations.reshape((1, deviations.shape[0] * 3))
else:
    deviations = deviations.reshape((1, deviations.shape[0]))
proj_corr = np.dot(deviations,pca_corr_evecs_sort)
proj_corr = (1 / (super_ensemble.numAtoms() ** 0.5)) * proj_corr
proj_corr_suppress = np.dot(deviations/np.sqrt(np.diag(pca_cov)),pca_corr_evecs_sort)
proj_corr_suppress = (1 / (super_ensemble.numAtoms() ** 0.5)) * proj_corr_suppress
pca_dict.update({'proj_cov':proj_cov, 'proj_corr':proj_corr,
'proj_corr_suppress':proj_corr_suppress})
# #MI = -1/2 log det (I - Cji Cii^-1 Cij Cjj^-1),
# L = ...
# M = -0.5 * np.log10(np.linalg.det(np.eye(L.shape[0])-L))
return(pca_dict)

def visualize_projection(proj, lookup_list, out_dir, run_name, percentiles=(5,95), n_models=10,
n_modes=20):
    """
    """
    def scatter3d(x, y, z, cs, cm='jet'):
        """
        http://stackoverflow.com/questions/8891994/matplotlib-3d-scatter-plot-with-color-gradient
        """
        cm = plt.get_cmap(cm)
        cNorm = mplcolors.Normalize(vmin=min(cs), vmax=max(cs))
        scalarMap = mplcm.ScalarMappable(norm=cNorm, cmap=cm)
        fig = plt.figure()
        ax = fig.add_subplot(111,projection='3d')
        ax.scatter(x, y, z, c=scalarMap.to_rgba(cs), alpha=0.1)
        ax.set_title('Super-ensemble PCA: First 3 modes')
        scalarMap.set_array(cs)
        fig.colorbar(scalarMap)
        plt.show()
        return(fig,ax)

    if isinstance(out_dir,str):
        if not os.path.exists(out_dir):
            try:
                os.mkdir(out_dir)
            except:
                print('Failed to find or create {}'.format(out_dir))
                raise
        with open(os.path.join(out_dir,'{}_pca_mode-info.txt'.format(run_name)),'w') as f:
            f.write('# percentiles = ({}, {}), n_models = {}, n_modes = {}\n'.format(percentiles[0],
            percentiles[1], n_models, n_modes))
            f.write('Mode\tMagnitude\tEnsemble\tModel\n')
            for i in range(n_modes):
                print('\rWriting data for mode {}/{}'.format(i,n_modes-1),end='')
                if not os.path.isdir(os.path.join(out_dir,'{}_pymol_commands'.format(run_name))):
                    os.mkdir(os.path.join(out_dir,'{}_pymol_commands'.format(run_name)))
                with open(os.path.join(out_dir,'{}_pymol_commands'.format(run_name),'{}_pca_mode-
                {}.pml'.format(run_name,i)),'w') as g:
                    if (type(percentiles[0]) in (float, int) and type(percentiles[1]) in (float,int)
                    and all([p > 0 for p in percentiles] and all([p <= 100 for p in percentiles]))):
                        if percentiles[0] > percentiles[1]:
                            percentiles = (percentiles[1], percentiles[0])
                        idx_full = np.argsort(proj[:,i])
                        idx_min = idx_full[proj[idx_full,i] >
                        np.percentile(proj[:,i],percentiles[0])]
                        idx_max = idx_full[proj[idx_full,i] <
                        np.percentile(proj[:,i],percentiles[1])]
                        idx = [j for j in idx_min if j in idx_max]

```

```

else:
    idx = np.argsort(proj[:,i])
    idx_sub = list(idx[::int(proj.shape[0]/n_models)])
    if idx[-1] != idx_sub[-1]:
        idx_sub.pop(-1)
        idx_sub.append(idx[-1])
    for k,j in enumerate(idx_sub):
        # Write to basic text file

f.write('{}\t{:.6}\t{}\t{}\n'.format(i,proj[j,i], '{}.pdb'.format(lookup_list[j][0]),lookup_list[j][3]+1))

        # Write to PyMOL command file
        g.write('load {}.pdb\n'.format(lookup_list[j][0]))
        g.write('split_states {0}, {1}, {1}, prefix=conf-{}-
{3}_\n'.format(lookup_list[j][0],lookup_list[j][3]+1,i,k))
        g.write('alter conf-{}-{}_{:04d},
b={:.4}\n'.format(i,k,lookup_list[j][3]+1,proj[j,i]))
        g.write('delete {}\n'.format(lookup_list[j][0]))
        g.write('unset specular\n')
        g.write('hide everything\n')
        g.write('remove solvent\n')
        g.write('remove hydrogens\n')
        g.write('show ribbon\n')
        g.write('set ribbon_radius, 1.0\n')
        g.write('show lines, name C+CA+CB+O+N\n')
        g.write('set line_width, 1.0\n')
        g.write('spectrum b, blue_grey_red, minimum = {:.4}, maximum =
{:.4}\n'.format(proj[idx,i].min(),proj[idx,i].max()))
        #g.write('spectrum b, blue_grey_red, minimum = {:.4}, maximum =
{:.4}\n'.format(*multiconf_utilities.cmap_bounds(proj[:,i])))
        print('\rFiles written for {} modes.          '.format(n_modes))
        print('Plotting...')
    try:
        fig,ax=scatter3d(proj[:,0],proj[:,1],proj[:,2],[sub[1] for sub in lookup_list])
    except:
        fig,ax=scatter3d(proj[:,0],proj[:,1],proj[:,2],range(proj.shape[0]))
    fig.savefig(os.path.join(out_dir,'{}_top-3-
modes_scatter.eps'.format(run_name)),transparent=True,close=True,verbose=True)
    return

def reduce_sort_correlations_by_residue(atom_corr_mat, resi_numbers, atom_names, xyz=True,
start_diagonal=0, take_abs=True, run_name='run', header=None, out_dir=None, plot=False):
    """
    Return a list of most correlated residues.
    Input:
        atom_corr_mat: A two-dimensional, symmetric np.ndarray with dimensions equal to n_atoms x
n_atoms or (3 x n_atoms) x (3 x n_atoms).
        resi_numbers: A list of residue indices per element in atom_corr_mat, obtained from a
ProDy PDB object (e.g., pdb_object.getResnums())
        atom_names: A list of atom names per element in atom_corr_mat, obtained from a ProDy PDB
object (e.g., a_pdb_object.getNames())
        xyz: If True, assume that there is a correlation value in each dimension present in the
matrix, and take only the largest one (i.e., take the largest correlation in x, y, or z).
        start_diagonal: Integer specifying diagonal index at which to begin sorting; specify
something around 10 to get out of atoms for a single residue
        take_abs: If True, absolute value will be taken and all correlations will be positive.
        run_name: A string specifying a descriptive title
        header: A string specifying information to add at beginning of text file output.
        out_dir: If not None and if a string specifying a valid directory, plots/text file will
be written there.
        plot: If True, plot the input matrix if xyz = False or the reduced matrix if xyz = True.
    Output:
        atom_corr_mat_reduced: An np.ndarray with atomic correlation coefficients reduced from
xyz to a single value.
        atom_corr_mat_reduced_flat: A list of sublists [idx_i, idx_j, corr_coef], where each
sublist describes the correlation float corr_coef between two atoms with integer-valued indices
i and j.
        resi_atom_list: A list of strings, 'resi_atom', for each element of
atom_corr_mat_reduced.

```

```

    resi_corr_mat_reduced: An np.ndarray with n_resis x n_resis elements, where each element
is the mean correlation of all atomic correlations for that residue from atom_corr_mat_reduced.
    resi_corr_mat_reduced_flat: A list of sublists [idx_i, idx_j, corr_coef], where each
sublist describes the correlation float corr_coef between two residues with integer-valued
indices i and j.
    unique_resis: A list of integers corresponding to unique residue indices for
resi_corr_mat_reduced and resi_corr_mat_reduced_flat
'''
def to_flat_mat(mat, start_diagonal):
    triu_idx = np.triu_indices_from(mat, start_diagonal)
    n_ele = len(triu_idx[0])
    flat = np.zeros((n_ele, 3))
    for i in range(n_ele):
        j, k = triu_idx[0][i], triu_idx[1][i]
        flat[i, :] = [j, k, mat[j, k]]
    flat = flat[np.argsort(flat[:, 2])[:, :-1]]
    return(flat)

# Validate input data
if not isinstance(atom_corr_mat, np.ndarray):
    raise ValueError('atom_corr_mat must be a numpy array, not type
{}'.format(type(atom_corr_mat)))
if atom_corr_mat.ndim != 2 or atom_corr_mat.shape[0] != atom_corr_mat.shape[1]:
    raise ValueError('atom_corr_mat must be a square matrix with 2 dimensions!')
if resi_numbers.size != atom_names.size:
    raise ValueError('resi_numbers and atom_names must be the same size!')
if resi_numbers.size != atom_corr_mat.shape[0] and resi_numbers.size*3 !=
atom_corr_mat.shape[0]:
    raise ValueError('resi_numbers and atom_names size must be equal or a one third of the
size of atom_corr_mat!')
if xyz:
    if resi_numbers.size == atom_corr_mat.shape[0]:
        resi_numbers = resi_numbers[range(0, resi_numbers.size, 3)]
        atom_names = atom_names[range(0, atom_names.size, 3)]
    if atom_corr_mat.shape[0] % 3 == 0:
        n_atoms = atom_corr_mat.shape[0] // 3
    else:
        raise ValueError('atom_corr_mat shape is not a multiple of three; fix input or set
xyz=False if there is only a single correlation value per atom.')
else:
    n_atoms = atom_corr_mat.shape[0]
    if len(resi_numbers) != n_atoms:
        raise ValueError('Length of resi_numbers ( {}) does not match the number of atoms ( {}) in
atom_corr_mat!'.format(len(resi_numbers), n_atoms))
    if len(atom_names) != n_atoms:
        raise ValueError('Length of atom_names ( {}) does not match the number of atoms ( {}) in
atom_corr_mat!'.format(len(atom_names), n_atoms))
    # If correlations are provided in each dimension, reduce atom_corr_mat to shape = [n_atoms,
n_atoms] with each value corresponding to the biggest correlation in any dimension.
    if xyz:
        atom_corr_mat_reduced = np.zeros((n_atoms, n_atoms))
        for i, j in np.nditer(np.triu_indices(n_atoms)):
            this_block = atom_corr_mat[i*3:i*3+3, j*3:j*3+3]
            if not take_abs:
                atom_corr_mat_reduced[i, j] =
this_block[np.unravel_index(np.abs(this_block).argmax(), this_block.shape)]
            elif take_abs:
                #atom_corr_mat_reduced[i, j] = this_block.max()
                atom_corr_mat_reduced[i, j] = np.abs(this_block).max()
                #atom_corr_mat_reduced[i, j] = np.abs(this_block).mean()
            atom_corr_mat_reduced += np.triu(atom_corr_mat_reduced, 1).T
    else:
        atom_corr_mat_reduced = atom_corr_mat.copy()
# Create reduced matrix pixel labels.
resi_atom_list = ['{}_{}'.format(resi, atom) for resi, atom in zip(resi_numbers, atom_names)]
unique_resis = [resi_numbers[i] for i in sorted(np.unique(resi_numbers,
return_index=True)[1])] # Creates list of unique residues, preserving order.
n_resis = len(unique_resis)
resi_corr_mat_reduced = np.zeros((n_resis, n_resis))

```

```

for i,j in np.nditer(np.triu_indices(n_resis)):
    resi_i = unique_resis[i]
    resi_j = unique_resis[j]
    slice_ij = multiconf_utilities.get_resi_coord_slice_for_prody_ensemble(resi_numbers,
[resi_i,resi_j], split_by_resi=True)
    if not take_abs:
        resi_corr_mat_reduced[i,j] = atom_corr_mat_reduced[slice_ij[0][0]:slice_ij[0][-
1]+1,slice_ij[1][0]:slice_ij[1][-1]+1].mean()
    if take_abs:
        resi_corr_mat_reduced[i,j] =
np.abs(atom_corr_mat_reduced[slice_ij[0][0]:slice_ij[0][-1]+1,slice_ij[1][0]:slice_ij[1][-
1]+1]).mean()
    resi_corr_mat_reduced += np.triu(resi_corr_mat_reduced,1).T
    # If requested, plot.
    if plot:
        if take_abs:
            cmap=plt.cm.afmhot
            vmin = 0
            vmax = 1
            ncols=2
        elif not take_abs:
            cmap=plt.cm.RdBu_r
            vmin,vmax = multiconf_utilities.cmap_bounds(resi_corr_mat_reduced)
            ncols=3
        figA,axA=plt.subplots(nrows=1,ncols=ncols,squeeze=False,figsize=(30,15))
        p=axA[0][0].matshow(atom_corr_mat_reduced,cmap=cmap,vmin=vmin,vmax=vmax)
        axA[0][0].set_title('{}: Reduced atomic correlation matrix'.format(run_name))
        axA[0][0].set_xlabel('atom i')
        axA[0][0].set_ylabel('atom j')
        j = [int(n) for n in axA[0][0].get_xticks()[::-1]]
        x = [resi_atom_list[n] for n in j]
        axA[0][0].set_xticklabels(x)
        axA[0][0].set_yticklabels(x)
        divA_p = make_axes_locatable(axA[0][0])
        caxA_p = divA_p.append_axes('right',size='5%',pad=0.05)
        cbarA_p = plt.colorbar(p,cax=caxA_p)
        q = axA[0][1].matshow(resi_corr_mat_reduced,cmap=cmap,vmin=vmin,vmax=vmax)
        axA[0][1].set_title('{}: Reduced residue correlation matrix'.format(run_name))
        axA[0][1].set_xlabel('atom i')
        axA[0][1].set_ylabel('atom j')
        j = [int(n) for n in axA[0][1].get_xticks()[::-1]]
        x = [unique_resis[n] for n in j]
        axA[0][1].set_xticklabels(x)
        axA[0][1].set_yticklabels(x)
        divA_q = make_axes_locatable(axA[0][1])
        caxA_q = divA_q.append_axes('right',size='5%',pad=0.05)
        cbarA_q = plt.colorbar(q,cax=caxA_q)
        if not take_abs:
            r = axA[0][2].matshow(np.abs(resi_corr_mat_reduced),cmap=plt.cm.afmhot,vmin=0,vmax=1)
            axA[0][2].set_title('{}: Reduced residue correlation matrix, absolute value
taken'.format(run_name))
            axA[0][2].set_xlabel('atom i')
            axA[0][2].set_ylabel('atom j')
            j = [int(n) for n in axA[0][2].get_xticks()[::-1]]
            x = [unique_resis[n] for n in j]
            axA[0][2].set_xticklabels(x)
            axA[0][2].set_yticklabels(x)
            divA_r = make_axes_locatable(axA[0][2])
            caxA_r = divA_r.append_axes('right',size='5%',pad=0.05)
            cbarA_r = plt.colorbar(r,cax=caxA_r)
        if out_dir is not None and isinstance(out_dir,str) and os.path.isdir(out_dir):
            try:
                plt.savefig(os.path.join(out_dir,'{}_reduced-corr-
mat.eps'.format(run_name)),transparent=True,format='eps')
            except:
                print('Failed to save figure {}; will carry on anyways.'.format(thisFig))
        else:
            plt.show()
    # Write text file if out_dir is provided.

```

```

    if out_dir is not None and isinstance(out_dir, str) and os.path.isdir(out_dir):
        full_path = [os.path.join(out_dir, '{}_sorted-atomic-correlations.txt'.format(run_name)),
os.path.join(out_dir, '{}_sorted-resi-correlations.txt'.format(run_name))]
        multiconf_utilities.mat_to_txt(atom_corr_mat_reduced, resi_atom_list, full_path[0],
header='{}, pairwise atomic correlations'.format(header), start_diagonal=start_diagonal)
        multiconf_utilities.mat_to_txt(resi_corr_mat_reduced, unique_resis, full_path[1],
header='{}, pairwise residue correlations'.format(header), start_diagonal=start_diagonal)
        # Created sorted lists.
        atom_corr_mat_reduced_flat = to_flat_mat(atom_corr_mat_reduced, start_diagonal)
        resi_corr_mat_reduced_flat = to_flat_mat(resi_corr_mat_reduced, start_diagonal)

    return(atom_corr_mat_reduced, atom_corr_mat_reduced_flat, resi_atom_list,
resi_corr_mat_reduced, resi_corr_mat_reduced_flat, unique_resis)

def atom_coord_joint_probability(super_ensemble, ref_pdb, resi_atom_i, resi_atom_j,
mean_subtract=True, rotate_basis=True, calc_stats=True, xy_minmax_percentiles=((0,100),(0,100)),
run_name='run', out_dir=None, mixnyn_path=''):
    """
    Create jointplot for coordinates of atoms from two residues i and j.
    analyze_superensemble.atom_coord_joint_probability(ens, ref, (7,'C'),
(392,'CD1'),run_name='descriptivetitle', out_dir='/path/to/dir')
    Input:
        super_ensemble: A prody.Ensemble object, which has all coordinates for all models in the
super_ensemble.
        ref_pdb: A prody.Selection object, which serves as a reference object for the
super_ensemble.
        resi_atom_i: A tuple, (resi_idx_i, atom_name_i), where resi_idx_i is the integer-valued
residue index of residue i and atom_name_i is a string specifying the atom of interest, e.g.,
'CA'.
        resi_atom_j: A tuple, (resi_idx_j, atom_name_j), where resi_idx_j is the integer-valued
residue index of residue j and atom_name_j is a string specifying the atom of interest, e.g.,
'CB'.
        mean_subtract: If True, coordinates of each atom will be mean-centered about zero prior
to analysis.
        rotate_basis: If True, principal components analysis will be applied to the coordinates
of each atom to ensure that projections are along directions of maximum variance.
        calc_stats: If True, create a dictionary with a variety of statistics regarding the two
atoms.
        xy_minmax_percentiles: A tuple containing two sub-tuples, where the first specifies the
percentile bounds for data along the x-axis of the joint plot and the second specifies the
percentile bounds for data along the y-axis of the joint plot.
        run_name: A string specifying a descriptive title.
        out_dir: If not None and a path to a valid directory, pickle and save plots there.
        mixnyn_path: Path to MIXnyn executable; necessary if calc_stats is True.
    Output:
        stats_dict: If calc_stats is True, stats_dict will be returned. It has the following
key/value pairs, where i refers to the first residue in the comparison, j the second, a the
coordinate in question for atom i, and b the coordinate in question for atom j:
        'resi_atom_i_index-resi_atom_i_name-coord_a-resi_atom_j_index-resi_atom_j_name-
coord_b':
            stat_tuple: A tuple, (r, m), where r is the correlation coefficient and m is the
mutual information in bits.
    """
    import seaborn as sns
    resi_numbers = ref_pdb.getResnums()
    atom_names = ref_pdb.getNames()
    slices = multiconf_utilities.get_resi_coord_slice_for_prody_ensemble(resi_numbers,
[resi_atom_i[0],resi_atom_j[0]], atom_names=atom_names,
atom_selection=[[resi_atom_i[1]], [resi_atom_j[1]]], split_by_resi=False)
    if len(slices) != 2:
        raise ValueError('Invalid resi/atom pairs specified, check selections!!')
    print('Getting relevant coordinates from super-ensemble...')
    coords = super_ensemble.getCoordsets()[ :, slices, :].copy()
    if mean_subtract:
        print('Mean-subtracting data...')
        coords[:,0,:] -= coords[:,0,:].mean(axis=0)
        coords[:,1,:] -= coords[:,1,:].mean(axis=0)
    if rotate_basis:
        print('Performing PCA in Cartesian space...')

```


and reference.

chain: If None, chain 'A' will be analyzed. If a string corresponding to a chain in super_ensemble and reference, that chain will be considered instead.

mode: A string specifying the dihedral calculation mode. If 'all', all dihedrals will be calculated. If 'bb', only backbone dihedrals (omega, phi, psi) will be calculated. If 'sc', only chi angles will be calculated.

filter_nan: If True, columns of NaN values corresponding to dihedrals which could not be calculated will be removed.

verbose: If True, more information will be provided.

Output:

dihedral_dict: A dictionary keyed by residue numbers with values equal to an np.ndarray with dimensions n_models x n_dihedrals, where n_models is the number of models in the super-ensemble and n_dihedrals is the number of dihedral angles for a given residue.

```
'''
# Check super_ensemble, reference
if not isinstance(super_ensemble, prody.Ensemble):
    raise ValueError('super_ensemble must be a ProDy ensemble object!')
if not isinstance(reference, prody.Selection):
    raise ValueError('reference must be a ProDy selection object!')
# Check residue_numbers input format
if type(residue_numbers) not in (list, int, np.int64):
    if isinstance(residue_numbers, str):
        residue_numbers = np.array([int(residue_numbers)])
    else:
        raise('Invalid residue_numbers format.')
if type(residue_numbers) in (int, np.int64):
    residue_numbers=np.array([residue_numbers])
elif isinstance(residue_numbers, list):
    if not (all([isinstance(r,int) for r in residue_numbers]) or all([isinstance(r,np.int64)
for r in residue_numbers])):
        if all([isinstance(r,str) for r in residue_numbers]):
            residue_numbers = np.array([int(r) for r in residue_numbers])
        else:
            raise ValueError('Elements in residue_numbers must be integers')
elif isinstance(residue_numbers, np.array) and 'int' not in
str(residue_numbers.dtype).lower():
    raise ValueError('If residue_numbers is passed as np.array, it must be an array of
integers.')
if not set(residue_numbers).issubset(reference.getResnums()):
    raise ValueError('Resi number {} not found in reference!'.format)
# Make sure mode is correct
mode=mode.lower()
if mode not in ('all', 'bb', 'sc'):
    raise ValueError('Invalid dihedral calculation mode.')
# Make sure chain selection is not ambiguous
if chain is None and np.unique(reference.getChids()).size > 1:
    raise ValueError('Chain must be specified.')
elif chain is None and np.unique(reference.getChids()).size == 1:
    chain = reference.getChids()[0]
if verbose:
    print('Calculating dihedrals for chain {} the following residues in \'{ }\''
mode:\n{}'.format(chain, mode, residue_numbers))

# Create dihedral lookup dictionary
chi_dict = dict(
    chi1=dict(
        ARG=['N', 'CA', 'CB', 'CG'],
        ASN=['N', 'CA', 'CB', 'CG'],
        ASP=['N', 'CA', 'CB', 'CG'],
        CYS=['N', 'CA', 'CB', 'SG'],
        GLN=['N', 'CA', 'CB', 'CG'],
        GLU=['N', 'CA', 'CB', 'CG'],
        HIS=['N', 'CA', 'CB', 'CG'],
        ILE=['N', 'CA', 'CB', 'CG1'],
        LEU=['N', 'CA', 'CB', 'CG'],
        LYS=['N', 'CA', 'CB', 'CG'],
        MET=['N', 'CA', 'CB', 'CG'],
        PHE=['N', 'CA', 'CB', 'CG'],
        PRO=['N', 'CA', 'CB', 'CG'],
```

```

        SER=['N', 'CA', 'CB', 'OG'],
        THR=['N', 'CA', 'CB', 'OG1'],
        TRP=['N', 'CA', 'CB', 'CG'],
        TYR=['N', 'CA', 'CB', 'CG'],
        VAL=['N', 'CA', 'CB', 'CG1'],
    ),
    altchi1=dict(
        VAL=['N', 'CA', 'CB', 'CG2'],
    ),
    chi2=dict(
        ARG=['CA', 'CB', 'CG', 'CD'],
        ASN=['CA', 'CB', 'CG', 'OD1'],
        ASP=['CA', 'CB', 'CG', 'OD1'],
        GLN=['CA', 'CB', 'CG', 'CD'],
        GLU=['CA', 'CB', 'CG', 'CD'],
        HIS=['CA', 'CB', 'CG', 'ND1'],
        ILE=['CA', 'CB', 'CG1', 'CD1'],
        LEU=['CA', 'CB', 'CG', 'CD1'],
        LYS=['CA', 'CB', 'CG', 'CD'],
        MET=['CA', 'CB', 'CG', 'SD'],
        PHE=['CA', 'CB', 'CG', 'CD1'],
        PRO=['CA', 'CB', 'CG', 'CD'],
        TRP=['CA', 'CB', 'CG', 'CD1'],
        TYR=['CA', 'CB', 'CG', 'CD1'],
    ),
    altchi2=dict(
        ASP=['CA', 'CB', 'CG', 'OD2'],
        LEU=['CA', 'CB', 'CG', 'CD2'],
        PHE=['CA', 'CB', 'CG', 'CD2'],
        TYR=['CA', 'CB', 'CG', 'CD2'],
    ),
    chi3=dict(
        ARG=['CB', 'CG', 'CD', 'NE'],
        GLN=['CB', 'CG', 'CD', 'OE1'],
        GLU=['CB', 'CG', 'CD', 'OE1'],
        LYS=['CB', 'CG', 'CD', 'CE'],
        MET=['CB', 'CG', 'SD', 'CE'],
    ),
    chi4=dict(
        ARG=['CG', 'CD', 'NE', 'CZ'],
        LYS=['CG', 'CD', 'CE', 'NZ'],
    ),
    chi5=dict(
        ARG=['CD', 'NE', 'CZ', 'NH1'],
    ),
)

# Create data objects for dihedrals
unique_resinfo = [[reference.getResnums()[i], reference.getResnames()[i]] for i in
sorted(np.unique(reference.getResnums(), return_index=True)[1])]
resi_chi_lookup = [r+[] for r in unique_resinfo if r[0] in residue_numbers]
for entry in resi_chi_lookup:
    for chi in chi_dict.keys():
        if entry[1] in chi_dict[chi]:
            entry[2].update({chi:chi_dict[chi][entry[1]]})
            entry.append(len(entry[2].keys()))
n_coordsets = super_ensemble.numCoordsets()
n_resis = len(residue_numbers)
dihedral_dict = {entry[0]:np.ndarray([n_coordsets,3+entry[3]])*np.nan for entry in
resi_chi_lookup}

# Calculate all requested dihedrals
for i,coords in enumerate(super_ensemble.iterCoordsets()):
    if verbose:
        print('Calculating dihedrals over (super-)ensemble
({:.0%})...'.format(i/n_coordsets),end='\r')
    reference.setCoords(coords)
    assert np.all(reference.getCoords() == coords)
    hv = reference.getHierView()

```

```

for entry in resi_chi_lookup:
    resi = hv[chain][entry[0]]
    if mode in ('all', 'bb'):
        try:
            dihedral_dict[entry[0]][i,0] = prody.measure.calcPhi(resi)
        except:
            dihedral_dict[entry[0]][i,0] = np.nan
        try:
            dihedral_dict[entry[0]][i,1] = prody.measure.calcPsi(resi)
        except:
            dihedral_dict[entry[0]][i,1] = np.nan
        try:
            dihedral_dict[entry[0]][i,2] = prody.measure.calcOmega(resi)
        except:
            dihedral_dict[entry[0]][i,2] = np.nan
    else:
        dihedral_dict[entry[0]][i,:3] = 3*[np.nan]
    if mode in ('all', 'sc'):
        for j,chi_atoms in enumerate(entry[2].values()):
            try:
                dihedral_dict[entry[0]][i,3+j] = prody.measure.calcDihedral(*[resi[atom]
for atom in chi_atoms])
            except:
                dihedral_dict[entry[0]][i,3+j] = np.nan
        else:
            dihedral_dict[entry[0]][i,3:] = entry[3]*[np.nan]
    if filter_nan:
        if verbose:
            print('\nStripping NaNs...')
        for k,v in dihedral_dict.items():
            v=v[:,~np.isnan(v).all(axis=0)]
            if verbose and np.isnan(v).any():
                print('Warning, NaN still present in dihedral_dict for resi {} after
filtering!'.format(k))
        return(dihedral_dict)

def calculate_pairwise_dihedral_mutual_information(dihedrals_i, dihedrals_j, k=6, mixnyn_path='',
item_indices=(None, None), out_dir=os.getcwd(), shuffle=False, tidy=True, verbose=True,
use_mixnyn=True):
    """
    Calculates the mutual information in dihedrals between two residues in the super-ensemble.
    Input:
        dihedrals_i: An np.ndarray with dimensions n_models x n_dihedrals, where n_models is the
number of models in the super-ensemble and n_dihedrals is the number of dihedral angles for
residue i.
        dihedrals_j: An np.ndarray with dimensions n_models x n_dihedrals, where n_models is the
number of models in the super-ensemble and n_dihedrals is the number of dihedral angles for
residue j.
        k: Integer-valued number of nearest neighbors for MI estimator
        mixnyn_path: A string specifying the path to the MIXnyn executable.
        item_indices: A tuple used to help organize multiprocessing results. If not equal to
(None, None) and each element is an integer, the first value will be returned as the index for
residue i and the second as the index for residue j.
        out_dir: A string specifying the location to write temporary files for MIXnyn
calculations.
        shuffle: If True, dihedrals will be shuffled between models prior to MI calculation.
        tidy: If True, MIXnyn input text files will be deleted after calculations are complete.
        verbose: If True, say more.
        use_mixnyn: If True, use MIXnyn.py to run the MIXnyn executable. If False, use the Non-
Parametric Entropy Estimators Toolbox (NPEET; entropy_estimators.py). In the case of using NPEET,
arguments mixnyn_path, shuffle, tidy are ignored.
    Output:
        mi: A float specifying the mutual information between distributions of dihedrals for
residue i and residue j.
    """
    if not isinstance(dihedrals_i, np.ndarray) or not isinstance(dihedrals_j, np.ndarray):
        raise ValueError('Input data must be provided as a pair of matrices containing
dihedrals.')
    if max(dihedrals_i.shape) != max(dihedrals_j.shape):

```

```

        raise ValueError('Number of observations must be consistent between two supplied
matrices.')
    else:
        n_obs = max(dihedrals_i.shape)
        if verbose and None not in item_indices:
            if item_indices[0] == item_indices[1]:
                print('Calculating self MI for residue {}'.format(item_indices[0]))
            else:
                print('Calculating MI between dihedrals in residues {} and
{}'.format(*item_indices))
        elif verbose:
            print('Calculating MI between input dihedrals matrices...')
        if np.isnan(dihedrals_i.any() or np.isnan(dihedrals_j).any()):
            if verbose:
                print('Warning, NaN encountered in input, skipping calculation!')
            mi = np.nan
        else:
            if use_mixnyn:
                mi = Mixnyn(dihedrals_i, dihedrals_j, k=k, path=mixnyn_path, outDir=out_dir,
shuffle=shuffle, tidy=tidy)
            else:
                mi = ee.mi(dihedrals_i, dihedrals_j, k=k)
        del(dihedrals_i, dihedrals_j)
        if None not in item_indices:
            return(item_indices[0], item_indices[1], mi)
        else:
            return(mi)

```

```

def calculate_pairwise_coord_mutual_information(super_ensemble, item_i, item_j,
analysis_level='residue', item_indices=(None, None), resi_numbers=None, atom_names=None, k=6,
mixnyn_path='', out_dir=os.getcwd(), shuffle=False, tidy=True, verbose=True, use_mixnyn=True):
    """

```

Calculates the mutual information in Cartesian coordinates between two or more atoms in the super-ensemble.

Input:

super_ensemble: A prody.Ensemble object, which has all coordinates for all models in the super_ensemble.
item_i: An np.ndarray with dimensions n_models x n_atoms, where n_models is the number of models in the super-ensemble and n_atoms is the number of atom classes in item i.
item_j: An np.ndarray with dimensions n_models x n_atoms, where n_models is the number of models in the super-ensemble and n_atoms is the number of atom classes in item j.
k: Integer-valued number of nearest neighbors for MI estimator
mixnyn_path: A string specifying the path to the Mixnyn executable.
item_indices: A tuple used to help organize multiprocessing results. If not equal to (None, None) and each element is an integer, the first value will be returned as the index for item i and the second as the index for item j.
out_dir: A string specifying the location to write temporary files for Mixnyn calculations.
shuffle: If True, dihedrals will be shuffled between models prior to MI calculation.
tidy: If True, Mixnyn input text files will be deleted after calculations are complete.
verbose: If True, say more.
use_mixnyn: If True, use Mixnyn.py to run the Mixnyn executable. If False, use the Non-Parametric Entropy Estimators Toolbox (NPEET; entropy_estimators.py). In the case of using NPEET, arguments mixnyn_path, shuffle, tidy are ignored.

Output:

mi: A float specifying the mutual information between distributions of atoms for item i and item j.
"""

```

    if analysis_level not in ('atom', 'residue'):
        raise ValueError('Analysis level must be either \'atom\' or \'residue\', not
{}'.format(analysis_level))
    if analysis_level == 'residue' and not isinstance(resi_numbers, np.ndarray):
        raise ValueError('resi_numbers must be a numpy array!')
    if not isinstance(item_i, int) or not isinstance(item_j, int):
        raise ValueError('Incorrect slice indices ({} , {}) provided with type ({} , {}); these
must be specified as integers consistent with residue or atom numbering in
ref_pdb!'.format(item_i, item_j, type(item_i), type(item_j)))
    if analysis_level == 'residue' and (item_i not in resi_numbers or item_j not in

```

```

resi_numbers):
    raise ValueError('Residue i or j not in resi_numbers!')
    if analysis_level == 'residue':
        slices = multiconf_utilities.get_resi_coord_slice_for_prody_ensemble(resi_numbers,
[item_i, item_j], atom_names=atom_names, atom_selection='all', split_by_resi=True)
    elif analysis_level == 'atom':
        slices = (np.array([item_i]), np.array([item_j]))
    if not isinstance(super_ensemble, np.ndarray):
        n_obs = super_ensemble.numCoordsets()
        coords = super_ensemble.getCoordsets()
    else:
        coords = super_ensemble
        n_obs = super_ensemble.shape[0]
    if item_indices != (None, None):
        if len(item_indices) != 2:
            raise ValueError('item_indices must be a tuple or list of length 2!')
        if not all([isinstance(idx, int) for idx in item_indices]):
            raise ValueError('item_indices must be a pair of integers!')
        if not all([idx >= 0 for idx in item_indices]):
            raise ValueError('item_indices must be greater than or equal to zero!')
        coord_i = coords[:, slices[0],:].reshape(n_obs, slices[0].size*3)
        coord_j = coords[:, slices[1],:].reshape(n_obs, slices[1].size*3)
        if use_mixnyn:
            coord_i = coord_i.T
            coord_j = coord_j.T
        if verbose:
            if item_indices == (None, None):
                idx_a, idx_b = item_i, item_j
            else:
                idx_a, idx_b = item_indices
            if idx_a == idx_b:
                print('Calculating self MI for {}...'.format(idx_a), end='\r')
            else:
                print('Calculating MI between {} and {}...'.format(idx_a, idx_b), end='\r')
        del(super_ensemble, coords)
        if use_mixnyn:
            mi = MIxyn(coord_i, coord_j, k=k, path=mixnyn_path, outDir=out_dir, shuffle=shuffle,
tidy=tidy)
        else:
            mi = ee.mi(coord_i, coord_j, k=k)
        del(coord_i, coord_j)
        if verbose:
            print('MI between {0:} {1:} and {2:} {0:} is {3:.3} bits.'.format(analysis_level, idx_a,
idx_b, mi))
        if item_indices != (None, None):
            return([item_indices[0], item_indices[1], mi])
        else:
            return([item_i, item_j, mi])

```

```

def analyze_atomic_mi(M, atom_names, resi_numbers, skip_covalent=True,
skip_adjacent_residues=None, start_diagonal=1, plot_dat_type='mean', run_name='run',
out_dir=None, plot=False):
    """

```

Explore the distribution of mutual information in an atomic mutual information matrix.
Input:

M: An np.ndarray. The pairwise atomic mutual information matrix.
atom_names: A list of atom names corresponding to indices of M.
resi_numbers: A list of residue numbers corresponding to indices of M.
skip_covalent: If True, mutual information between atoms which are covalently linked will not be included at any stage of the analysis.
skip_adjacent_residues: If not None and an integer, mutual information between atoms in residues closer in sequence space than this value will not be included at any stage of the analysis.
start_diagonal: An integer specifying the diagonal at which to start including mutual information in the analysis. By default, this is set to unity and self-MI along the 0th diagonal is excluded from all steps of the analysis.
plot_dat_type: A string specifying the type of data to plot. If equal to 'mean', a histogram with the mean MI per atom interaction class will be drawn. If equal to 'sum', a histogram with the sum of MI per atom interaction class will be drawn.

```

    run_name: A string specifying a descriptive title for the run.
    out_dir: If not None and a path to a valid directory, plots will be saved here.
Output:
    pairs: A dictionary with keyed by unique strings (atom_class_i)_(atom_class_j) and values
as lists of mutual information for that interaction type. For example, pairs['CA_0'] is a list of
all observed MI values between CA and 0 atoms in M, subject to the two skip arguments and the
start_diagonal argument.
    pair_stats: A dictionary with keyed by unique strings (atom_class_i)_(atom_class_j) with
values as sub-dictionaries. Each sub-dictionary has keys 'min', 'max', 'sum', 'mean', 'std' and
values corresponding to those statistics for the equivalent list in the pairs dictionary.
'''
# First, make histogram for interaction type
if M.ndim != 2 or M.shape[0] != M.shape[1]:
    raise ValueError('Input matrix must be two dimensional and square.')
if start_diagonal > M.shape[0] or start_diagonal < 0:
    raise ValueError('Start diagonal must be between 0 and the total number of atoms less
one.')
unique_atom_names = np.unique(atom_names)
pairs = {}
for pair in it.chain(it.combinations(unique_atom_names,2),((name, name) for name in
unique_atom_names)):
    pairs['{}_{}'.format(pair[0], pair[1])] = []
    for i,j in np.nditer(np.triu_indices_from(M, start_diagonal)):
        if skip_covalent:
            if resi_numbers[i] == resi_numbers[j]: # Skip MI within residues
                continue
            elif resi_numbers[i] == resi_numbers[j] - 1 and atom_names[i] == 'C' and
atom_names[j] == 'N':
                continue
            if isinstance(skip_adjacent_residues,int) and abs(resi_numbers[i] - resi_numbers[j]) <=
abs(skip_adjacent_residues):
                continue
            atom_i, atom_j = atom_names[i], atom_names[j]
            try:
                pairs['{}_{}'.format(atom_i, atom_j)].append(M[i,j])
                if pairs['{}_{}'.format(atom_i, atom_j)][-1] == 0.0 or pairs['{}_{}'.format(atom_i,
atom_j)][-1] < 0:
                    pairs['{}_{}'.format(atom_i, atom_j)][-1] = np.nan
            except KeyError:
                try:
                    pairs['{}_{}'.format(atom_j, atom_i)].append(M[j,i])
                    if pairs['{}_{}'.format(atom_j, atom_i)][-1] == 0.0 or
pairs['{}_{}'.format(atom_j, atom_i)][-1] < 0:
                        pairs['{}_{}'.format(atom_j, atom_i)][-1] = np.nan
                except KeyError:
                    raise
        pair_stats = {}
        for pair in pairs.keys():
            if len(pairs[pair]) > 0:
                pair_stats.update({pair:{'n_obs':len([p for p in pairs[pair] if p not in (0.0,
np.nan)]), 'min':np.nanmin(pairs[pair]), 'max':np.nanmax(pairs[pair]),
'sum':np.nansum(pairs[pair]), 'mean':np.nanmean(pairs[pair]), 'std':np.nanstd(pairs[pair])})})
            bb_atoms = ('N', 'C', 'CA', 'O', 'OXT')
            bb_sc_stats = {'bb_only_pairs':0, 'bb_only_mi':0., 'bb_partial_pairs':0, 'bb_partial_mi':0.,
'sc_only_mi':0., 'sc_only_pairs':0, 'total_pairs':0, 'total_mi':0.}
            for pair in pairs.keys():
                atom_i, atom_j = pair.split('_')
                if atom_i in bb_atoms and atom_j in bb_atoms:
                    try:
                        bb_sc_stats['bb_only_mi'] += pair_stats[pair]['sum']
                        bb_sc_stats['bb_only_pairs'] += pair_stats[pair]['n_obs']
                    except KeyError:
                        print('Couldn\t get pair_stats data for {}, skipping...'.format(pair))
                        continue
                elif atom_i in bb_atoms or atom_j in bb_atoms:
                    try:
                        bb_sc_stats['bb_partial_mi'] += pair_stats[pair]['sum']
                        bb_sc_stats['bb_partial_pairs'] += pair_stats[pair]['n_obs']
                    except:

```

```

        print('Couldn\'t get pair_stats data for {}, skipping...'.format(pair))
        continue
    else:
        try:
            bbsc_stats['sc_only_mi'] += pair_stats[pair]['sum']
            bbsc_stats['sc_only_pairs'] += pair_stats[pair]['n_obs']
        except:
            print('Couldn\'t get pair_stats data for {}, skipping...'.format(pair))
            continue
        bbsc_stats['total_pairs'] += pair_stats[pair]['n_obs']
        bbsc_stats['total_mi'] += pair_stats[pair]['sum']
        bbsc_stats.update({'bb_only_frac_mi':bbsc_stats['bb_only_mi']/bbsc_stats['total_mi'],
'bb_partial_frac_mi':bbsc_stats['bb_partial_mi']/bbsc_stats['total_mi'],
'sc_only_frac_mi':bbsc_stats['sc_only_mi']/bbsc_stats['total_mi']})

bbsc_stats.update({'bb_only_frac_pairs':bbsc_stats['bb_only_pairs']/bbsc_stats['total_pairs'],
'bb_partial_frac_pairs':bbsc_stats['bb_partial_pairs']/bbsc_stats['total_pairs'],
'sc_only_frac_pairs':bbsc_stats['sc_only_pairs']/bbsc_stats['total_pairs']})

frac_sum = 0
print('Backbone/sidechain interaction statistics:')
for k,v in sorted(bbsc_stats.items()):
    if 'frac' in k:
        print('\t{}: {:.2%}'.format(k,v))
        frac_sum += v
    elif 'pairs' in k:
        print('\t{}: {} instances'.format(k,v))
    elif 'mi' in k:
        print('\t{}: {:.2f} bits'.format(k,v))

if plot:
    # First, make bar plot with avg MI per interaction
    n_pairs = len(pair_stats.keys())
    names = []
    dat = np.ndarray(n_pairs)
    err = np.ndarray(n_pairs)
    for i,item in enumerate(pair_stats.items()):
        names.append(item[0])
        try:
            dat[i] = item[1][plot_dat_type]
        except KeyError:
            print('Invalid data type {} requested for plot, valid choices
are:\n{}'.format(plot_dat_type, item[1].keys()))
            err[i] = item[1]['std']
    indices = np.argsort(dat)[::-1]
    figA,axA=plt.subplots(nrows=1,ncols=2,squeeze=False,figsize=(30,10))
    x_idx = np.arange(dat.size)
    width = 1
    for i in range(2):
        if plot_dat_type == 'mean':
            if i == 0:
                axA[0][i].errorbar(x_idx+width, dat[indices], yerr=err[indices], fmt='.',
color='k', ecolor='grey')
            elif i == 1:
                axA[0][i].bar(x_idx[:10]+width, dat[indices][:10], width,
yerr=err[indices][:10], color='k', edgecolor='w', error_kw={'ecolor':'grey'})
                axA[0][i].set_ylabel('Mean mutual information (bits)')
        elif plot_dat_type == 'sum':
            axA[0][i].bar(x_idx+width, dat[indices]/bbsc_stats['total_mi'], width, color='k',
edgecolor='w')
            axA[0][i].set_ylabel('Fractional contribution to total mutual information')
            #axA[0][0].set_yscale('log')
    else:
        raise ValueError('Invalid data type requested.')
    axA[0][i].set_title('Atom pair contributions to MI, {}, skip covalent pairs = {},
start diagonal = {}'.format(plot_dat_type, skip_covalent,start_diagonal))
    axA[0][i].set_xlabel('Atomic pair')
    axA[0][i].set_xticks(x_idx+width)
    axA[0][i].set_xticklabels(np.array(names)[indices], rotation=-90)

```

```

    if i == 0:
        axA[0][i].set_xlim(-5,x_idx[-1]+(5*width))
        axA[0][i].set_ylim((0,dat[indices][0]+err[indices][0]+0.2))
    elif i == 1:
        axA[0][i].set_xlim(x_idx[0]+1,x_idx[10]+1)
        axA[0][i].set_ylim((dat[indices][9]-err[indices][9]-
0.2,dat[indices][0]+err[indices][0]+0.2))
    if out_dir is not None and isinstance(out_dir, str) and os.path.isdir(out_dir):
        plt.savefig(os.path.join(out_dir,'{}_atom-pair-
histograms.eps'.format(run_name)),transparent=True,close=True,format='eps')
    return(pairs, pair_stats)

def distance_vs_atomic_mi(M, D, out_dir, force_constant_max=10, run_name='run', plot=False,
verbose=False):
    """
    Examine distance dependence of atomic mutual information, and create a gamma function for
    ProDy ANM. The getCutoff function can also be used for ProDy GNM.
    Input:
        M: An np.ndarray. The pairwise atomic mutual information matrix.
        D: An np.ndarray. The pairwise atomic distance matrix.
        out_dir: A string specifying a path to which to save plots and write the Python file with
the gammaDistanceDependent and getCutoff functions for ProDy.
        force_constant_max: A positive integer specifying the upper-bounds on the force constant
for the gammaDistanceDependent function; The function essentially scales from this value at the
closest distances to a smaller constant (ideally 0) at very large distances.
        run_name: A string specifying a descriptive title.
        plot: If True, will draw a two-dimensional histogram of distance versus average MI with
the 0th diagonal excluded.
        verbose: If True, information about gammaDistanceDependent bins will be printed.
    Output:
        bin_stats: A list of lists, where each sublist has the form [k, m, s], where k is the
distance bin, starting at zero, m is the mean mutual information in that bin, and s is the
standard deviation of mutual information in that bin.
    """

    if not isinstance(out_dir, str) or not os.path.isdir(out_dir):
        raise ValueError('Invalid output directory specified.')
    M_flat = M[np.triu_indices_from(M,1)].flatten()
    D_flat = M[np.triu_indices_from(M,1)].flatten()
    M_flat_nozeros = M_flat[M_flat != 0.]
    D_flat_nozeros = D_flat[M_flat != 0.]
    bin_width = 1
    bins = np.arange(0,np.ceil(D_flat_nozeros.max()),bin_width)
    bin_lists = {bin_idx:[] for bin_idx in bins}
    for bin_idx in bins:
        print('Binning... ( {:.0%} )'.format(bin_idx/bins[-1]),end='\r')
        for i,j in np.nditer(np.triu_indices_from(obj['m_mean'],1)):
            if D[i,j] >= bin_idx and D[i,j] < bin_idx + bin_width:
                bin_lists[bin_idx].append(M[i,j])
    bin_stats = [[k, np.mean(v), np.std(v)] for k,v in sorted(bin_lists.items())]
    if verbose:
        print('Bin start: mean +/- std')
        [print(' {:.2f}: {:.2f} +/- {:.2f}'.format(x,y,e)) for x,y,e in bin_stats]
    # Normalize to get force constant for ANM
    min_mi = min([y for x,y,e in bin_stats if ~np.isnan(y)])
    max_mi = max([y for x,y,e in bin_stats if ~np.isnan(y)])
    for i,stats in enumerate(bin_stats):
        bin_stats[i].append((stats[0]+1)**2)
        bin_stats[i].append((stats[1]-min_mi)/(max_mi-min_mi)*force_constant_max)
    with open(os.path.join(out_dir,'{}_gamma-function.py'.format(run_name)),'w') as fo:
        fo.write('#!/usr/bin/env python\n\n')
        fo.write('def gammaDistanceDependent(dist_sq, *args):\n')
        start=False
        for i,stats in enumerate(bin_stats):
            if np.isnan(stats[4]):
                continue
            else:
                if not start:
                    fo.write('\tif dist_sq <= {}: return({:.2f})\n'.format(stats[3], stats[4]))

```

```

        start=True
    else:
        fo.write('\telif dist_sq <= {}: return({:.2f})\n'.format(stats[3], stats[4]))
        fo.write('\telse: return(0)\n\n')
        fo.write('def getCutoff(): return({:.2f})'.format(bin_stats[-1][0]+bin_width))
# Plot
if plot:
    plt.figure(figsize=(20,15))
    h,d_bins,mi_bins,im=plt.hist2d(D_flat_nozeros, M_flat_nozeros, cmap=plt.cm.afmhot,
bins=(np.ceil(D_flat_nozeros.max()),np.ceil(M_flat_nozeros.max()*10), normed=False)
    plt.xlim(np.percentile(D_flat_nozeros, 1), np.percentile(D_flat_nozeros, 99))
    plt.ylim(np.percentile(M_flat_nozeros, 1), np.percentile(M_flat_nozeros, 99))
    plt.title('PDZ3-CRIPT, distance vs. average MI, 0th diagonal excluded')
    plt.xlabel('Distance (Angstrom)')
    plt.ylabel('Mutual information (bits)')
    plt.colorbar(label='Counts')
    plt.savefig(os.path.join(out_dir, '{}_pdz3-cript_2dhist-dist-vs-
mi.eps'.format(run_name)),transparent=True,close=True,format='eps')

    plt.bar([x for x,y,e in bin_stats], [y for x,y,e in bin_stats], bin_width, color='k',
edgecolor='w', linewidth=1, yerr=[e for x,y,e in bin_stats], error_kw={'ecolor':'grey'})
    plt.xlim((min([x for x,y,e in bin_stats]), max([x for x,y,e in bin_stats])+bin_width))
    plt.ylim((min([y-e for x,y,e in bin_stats if ~np.isnan(y)])-0.1, max([y+e for x,y,e in
bin_stats if ~np.isnan(y)])+0.1))
    plt.xlabel('Distance (Angstrom)')
    plt.ylabel('Mutual information (bits)')
    plt.savefig(os.path.join(out_dir, '{}_pdz3-cript_mean-dist-vs-
mi.eps'.format(run_name)),transparent=True,close=True,format='eps')
    return(bin_stats)

if __name__ == '__main__':
    start_time = time.time()
    # Primary parser
    parser = argparse.ArgumentParser()
    parser.add_argument('-e', '--ensemble_list', nargs='*', default=[os.getcwd()], help='a
directory containing ensembles or a list of ensemble files.')
    parser.add_argument('-k', '--prody_pkl', type=str, default=None, help='a pickle with pre-
created super_ensemble, reference, and lookup objects.')
    parser.add_argument('-m', '--multiprocess', action='store_true', default=False,
help='parallelize some things.')
    parser.add_argument('-n', '--n_processors', type=int, default=mp.cpu_count(), help='number of
processors to use for any tasks for which multiprocessing is enabled; defaults to number of
available cores.')
    parser.add_argument('-p', '--plot', action='store_true', default=False, help='plot where
optional.')
    parser.add_argument('-d', '--out_dir', type=str, default=os.getcwd(), help='save output
here.')
    parser.add_argument('-r', '--run_name', type=str, default='run', help='name to prefix output
stuff with.')
    parser.add_argument('-f', '--rworkfilter', action='store_true', help='if specified, will remove
up to 80% of super-ensemble based on deviation of Rwork from normality.')
    subparsers = parser.add_subparsers(title='opmodes', help='operation
modes',dest='subparser_name')
    # prody_pkl subparser
    prody_pkl_parser = subparsers.add_parser('prody_pkl')
    prody_pkl_parser.add_argument('--pklselection', type=str, default=None, help='selection for
ProDy.')
    # pairwisemi subparser
    pairwisemi_parser = subparsers.add_parser('pairwisemi')
    pairwisemi_parser.add_argument('--mixnyn_path', type=str, default='', help='path to MIXnyn
executable; if in path, leave alone.')
    pairwisemi_parser.add_argument('--mi_observable', choices=['coordinates','dihedrals'],
type=str, default='coordinates', help='observable for which to calculate MI; must be
\'coordinates\' or \'dihedrals\'')
    pairwisemi_parser.add_argument('--coord_analysis_level', choices=['atom','residue'],
type=str, default='residue', help='mode in which to calculate coordinate MI; must be \'atom\' or
\'residue\'')
    pairwisemi_parser.add_argument('--dihedral_analysis_level', choices=['bb','sc', 'all'],
type=str, default='all', help='mode in which to calculate coordinate MI; must be \'atom\' or

```

```

\'residue\'.')
    pairwisemi_parser.add_argument('--shuffle', action='store_true', help='if specified, will
shuffle data across models; only works if using MIXnyn.')
    pairwisemi_parser.add_argument('--usenpeet', action='store_false', help='if specified, will
calculate mutual information with the Non-Parametric Entropy Estimation Toolbox (NPEET) to
calculate mutual information.')
    pairwisemi_parser.add_argument('--neighbors', type=int, default=6, help='integer-valued
number of nearest neighbors for MIXnyn or NPEET MI estimators.')
    # contactprob subparser
    contactprob_parser = subparsers.add_parser('contactprob')
    contactprob_parser.add_argument('--vdw_frac', type=float, default=0.2, help='a contact is
declared if the distance between two atoms is less than the sum of their van der Waals radii plus
this percent.')
    contactprob_parser.add_argument('--chunk', type=int, nargs=2, default=[1,1], help='two
integers; the first is the number of chunks to break the super ensemble into, the second is the
particular chunk to process; if (1,1), will calculate over the entire super ensemble, possibly
divided over cores.')
    # pairwisecc subparser
    pairwisecc_parser = subparsers.add_parser('pairwisecc')
    # buildnm subparser
    buildnm_parser = subparsers.add_parser('buildnm')
    buildnm_parser.add_argument('--nmpdb', type=str, help='PDB file from which to calculate NM
model.')
    buildnm_parser.add_argument('--nmselection', type=str, default=None, help='selection to
include in calculation of ANM.')
    buildnm_parser.add_argument('--nmmodel', type=str, choices=['anm','gnm'], default='anm',
help='type of network model to calculate.')
    buildnm_parser.add_argument('--nmcompare', choices=['corr','cov',None], default=None,
help='if provided, compare network model correlation matrix to ensemble correlation or covariance
matrix calculated with PCA.')
    buildnm_parser.add_argument('--nmnmodes', type=int, default=10, help='number of network model
modes to calculate.')
    buildnm_parser.add_argument('--nmgamma', type=str, default=None, help='path to a Python file
with function gammaDistanceDependent() that specifies distance dependent force constant model for
NM calculation.')
    # Covariance and correlation matrix convergence subparser
    covconv_parser = subparsers.add_parser('covconverge')
    covconv_parser.add_argument('--subset', choices=[None,'sc','bb'], default=None, help='Subset
of residues to include in super-ensemble; can be everything (None), back bone atoms (bb) or side
chain atoms (sc).')
    covconv_parser.add_argument('--chain', type=str, default='A', help='chain to preserve over
super-ensemble.')
    covconv_parser.add_argument('--selection_string', type=str, default=None, help='Parts of
model to include in super-ensemble. See ProDy selection syntax guidelines for more info. By
default, take all non-hydrogen protein atoms.')
    covconv_parser.add_argument('--ccrepeats', type=int, default=3, help='number of times to
repeat convergence calculations.')
    covconv_parser.add_argument('--ccsample', type=int, default=100, help='number of points
along trajectory to analyze.')
    # Projection analysis parser
    projection_parser = subparsers.add_parser('projection')
    projection_parser.add_argument('--percentiles', nargs=2, default=(5,95), help='top and bottom
principal component magnitudes to consider.')
    projection_parser.add_argument('--n_modes', type=int, default=20, help='number of top PCA
modes for which to output data.')
    projection_parser.add_argument('--n_models', type=int, default=10, help='number of models in
each PyMOL session created.')
    args = parser.parse_args()

    # Check for subparser
    if args.subparser_name is None:
        raise ValueError('subparser_name must be specified!')

    # Use appropriate number of processors
    if args.multiprocess:
        if args.n_processors < 1 or args.n_processors > mp.cpu_count():
            args.n_processors = mp.cpu_count()
    else:
        args.n_processors = 1

```

```

# Load stuff, or make sure ensembleList is a list of bona fide ensembles and get paths if it
is a directory
if args.prody_pkl is not None and os.path.isfile(args.prody_pkl):
    if args.rworkfilter:
        raise ValueError('Cannot filter super-ensemble by Rwork if preloading ProDy object.')
    preload = True
    with open(args.prody_pkl, 'rb') as f:
        prody_dict = pickle.load(f)
    if len(prody_dict.items()) > 3:
        raise ValueError('Specified prody_dict contains too many items!')
    print('Loading previously created ProDy objects in {}'.format(args.prody_pkl))
    found = [False, False, False]
    for value in prody_dict.values():
        if isinstance(value, prody.atomic.selection.Selection) and not found[0]:
            print('\tFound reference.')
            reference = value
            found[0] = True
        if isinstance(value, prody.ensemble.ensemble.Ensemble) and not found[1]:
            print('\tFound super-ensemble.')
            super_ensemble = value
            found[1] = True
        if isinstance(value, list) and not found[2]:
            print('\tFound lookup.')
            lookup = value
            found[2] = True
    if not all(found):
        raise ValueError('Did not find all required items in prody_dict!')
elif args.subparser_name == 'buildnm' and args.nmcompare is None:
    ...
else:
    preload = False
    ensemble_list_distilled = []
    for ens in args.ensemble_list:
        if os.path.isdir(ens):
            ensemble_list_distilled += glob.glob(os.path.join(ens, '*.pdb'))
        elif os.path.isfile(ens) and os.path.splitext(ens)[1] == '.pdb':
            ensemble_list_distilled += [ens]
        else:
            raise ValueError('{} is neither a directory or a file!'.format(ens))
    if len(ensemble_list_distilled) == 0:
        raise ValueError('No valid ensembles specified!')
    else:
        print('Found {} valid ensembles in specified
ensemble_list.'.format(len(ensemble_list_distilled)))
        if args.rworkfilter:
            ensemble_list_distilled, _ =
multiconf_utilities.filter_pdb_list_factor(ensemble_list_distilled, factor='work',
crop_percentile=('auto', 0.8), runName=args.run_name, outDir=args.out_dir)
        if not os.path.exists(args.out_dir):
            os.mkdir(args.out_dir)

    if args.subparser_name == 'prody_pkl':
        if preload == True:
            raise ValueError('This mode creates a new pickle with relevant prody objects, specify
raw data instead.')
        reference, super_ensemble, lookup = build_prody_super_ensemble(ensemble_list_distilled,
subset=None, chain='A', selection_string=args.ppkselection, run_name=args.run_name)
        with open(os.path.join(args.out_dir, '{}_prody-objects.pkl'.format(args.run_name)), 'wb')
as f:
            pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup}, f)

    if args.subparser_name == 'contactprob':
        if args.vdw_frac < 0 or args.vdw_frac > 1:
            raise ValueError('vdw_frac must be a float greater than or equal to 0 and less than
or equal to 1!')
        if args.chunk != [1,1]:
            if len(args.chunk) != 2:

```

```

        raise ValueError('chunk must be a list with exactly two elements.')
    if args.chunk[0] * args.chunk[1] <= 0 or sum(args.chunk) < 0:
        raise ValueError('chunk parameters ({} ) must be positive-valued
integers.'.format(args.chunk))
    if args.chunk[0] < args.chunk[1]:
        args.chunk = sorted(args.chunk)[::-1]
    if args.chunk[1]-1 not in list(range(args.chunk[0])):
        raise ValueError('Specified chunk index ({} ) must be an integer in
list(range({}))!'.format(args.chunk[1],args.chunk[0]))
    chunk_str = '_chunk-{}-{}_'.format(args.chunk[1],args.chunk[0])
    else:
        chunk_str = ''

    if not preload:
        reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=None, chain='A',
selection_string=None, run_name=args.run_name)
        with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
            pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup},f)
        resi_numbers = reference.getResnums()
        unique_resi_numbers = [resi_numbers[i] for i in sorted(np.unique(resi_numbers,
return_index=True)[1])]
        n_coordsets = super_ensemble.numCoordsets()

    if chunk_str != '':
        if args.chunk[0] > n_coordsets:
            raise ValueError('Number of chunks ({} ) cannot be greater than the number of
coordsets in super-ensemble ({} )!'.format(args.chunk[0],n_coordsets))
            idx_list = list(range(n_coordsets))
            idx_chunk_list = list(zip(*[iter(idx_list)]*(n_coordsets//args.chunk[0])))
            idx_chunk = list(idx_chunk_list[args.chunk[1]-1])
            if n_coordsets % args.chunk[0] != 0 and args.chunk[1] == args.chunk[0]: # add any
trailing indices on to the last chunk
                idx_chunk += list(range(idx_chunk_list[-1][-1]+1,n_coordsets))
            print('Analyzing super-ensemble chunk {} of {} (indices
{}:{}'.format(args.chunk[1],args.chunk[0], idx_chunk[0], idx_chunk[-1]))
            else:
                idx_chunk = list(range(n_coordsets))
                n_chunks = len(idx_chunk)

        print('Calculating pariwise residue contact matrices',end=' ')
        if args.multiprocess:
            if super_ensemble.numCoordsets() < n_chunks:
                args.n_processors = n_chunks
                print('in parallel over as many as {} cores...'.format(args.n_processors))
                pool = mp.Pool(processes=args.n_processors,maxtasksperchild=10)
                time.sleep(1)
                try:
                    command_gen = ((ensemble, reference, True, args.vdw_frac, None,
'run'+chunk_str[:-1], None, False, ens_idx) for ens_idx,ensemble in
enumerate(super_ensemble[idx_chunk]))
                    results = []
                    se_cm = []
                    start_i = 0
                    for i,command in enumerate(command_gen):

results.append(pool.apply_async(calculate_super_ensemble_distance_contact_matrix, args=command))
                    del(command)
                    if i > 0 and ((i + 1) % args.n_processors) == 0:
                        stdout.write('\x1b[2K\rCalculating super-ensemble contact matrices
({:.0%})...'.format(i/n_chunks))
                        stdout.flush()
                        se_cm += [r.get() for r in results]
                        results = []
                        start_i = i+1
                finally:
                    stdout.write('\x1b[2K\rClosing pool... ')

```

```

        stdout.flush()
        time.sleep(1)
        pool.terminate()
        pool.join()
        del(pool)
        time.sleep(1)
    se_cm = [cm[1] for cm in sorted(se_cm)]
    se_cm = np.vstack(se_cm)
    print('Finished calculations.')
else:
    print('serially.')
    se_cm = calculate_super_ensemble_distance_contact_matrix(super_ensemble, reference,
distance=False, vdw_frac=args.vdw_frac, special_resi_dict=None, run_name=args.run_name,
out_dir=args.out_dir, verbose=False)
    if chunk_str == '':
        se_cm_mean, se_cm_std = reduce_super_ensemble_distance_contact_matrix(se_cm,
binarize=True, mode='mean', run_name=args.run_name, plot=True, out_dir=args.out_dir)
    else:
        print('Pickling...\nDon\'t forget, you will need to stitch multiple se_cm matrices
together with reduce_super_ensemble_distance_contact_matrix!')
        with open(os.path.join(args.out_dir, '{}_super-ensemble-contact-
matrix{}.pkl'.format(args.run_name, chunk_str[:-1])), 'wb') as f:
            pickle.dump({'super_ensemble_contact_matrix':se_cm,
'lookup':unique_resi_numbers}, f)
        print('Done.')

    if args.subparser_name == 'pairwisemi':
        if not isinstance(args.neighbors, int) or args.neighbors < 1 or args.neighbors > 10:
            raise ValueError('neighbors must be an integer between 1 and 10, inclusive; instead,
got {} with value {}'.format(type(args.neighbors), args.neighbors))
        if not preload:
            reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=None, chain='A',
selection_string=None, run_name=args.run_name)
            with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
                pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup}, f)
            atom_names = reference.getNames()
            n_atoms = atom_names.size
            resi_numbers = reference.getResnums()
            unique_resi_numbers = [resi_numbers[i] for i in sorted(np.unique(resi_numbers,
return_index=True)[1])]
            n_resis = len(unique_resi_numbers)
            if args.mi_observable == 'coordinates':
                coords = super_ensemble.getCoordsets()
                del(super_ensemble, reference, lookup)
                print('\nPerforming pairwise coordinate {} mutual information
calculations'.format(args.coord_analysis_level), end=' ')
                if args.multiprocess:
                    if args.coord_analysis_level == 'residue':
                        n_pairs = int(((n_resis**2 - n_resis)/2) + n_resis)
                    elif args.coord_analysis_level == 'atom':
                        n_pairs = int(((n_atoms**2 - n_atoms)/2) + n_atoms)
                    if n_pairs < args.n_processors:
                        args.n_processors = n_pairs
                    print('in parallel over as many as {} cores...'.format(args.n_processors))
                    pool = mp.Pool(processes=args.n_processors, maxtasksperchild=10)
                    time.sleep(1)
                    try:
                        if args.coord_analysis_level == 'residue':
                            command_gen_a = ((coords, int(resi_i), int(resi_j),
args.coord_analysis_level, (None, None), resi_numbers, None, args.neighbors, args.mixnyn_path,
args.out_dir, False, True, False, args.usenpeet) for resi_i, resi_j in
it.combinations(unique_resi_numbers, 2))
                            command_gen_b = ((coords, int(resi), int(resi),
args.coord_analysis_level, (None, None), resi_numbers, None, args.neighbors, args.mixnyn_path,
args.out_dir, False, True, False, args.usenpeet) for resi in unique_resi_numbers)
                            command_gen = it.chain(command_gen_a, command_gen_b)

```

```

        elif args.coord_analysis_level == 'atom':
            command_gen = ((np.stack((coords[:,atom_i,:],
coords[:,atom_j,:]),axis=1), 0, 1, args.coord_analysis_level, (int(atom_i), int(atom_j)), None,
None, 6, args.mixnyn_path, args.out_dir, False, True, False) for atom_i, atom_j in
np.nditer(np.triu_indices(n_atoms)))
            results = []
            data_list = []
            start_i = 0
            for i,command in enumerate(command_gen):

results.append(pool.apply_async(calculate_pairwise_coord_mutual_information, args=command))
            del(command)
            if i > 0 and ((i + 1) % args.n_processors) == 0:
                stdout.write('\x1b[2K\rCalculating for {} pairs {}:{} of {} total
({:.0%})...'.format(args.coord_analysis_level, start_i+1, i+1, n_pairs, i/n_pairs))
                stdout.flush()
                data_list += [r.get() for r in results]
                results = []
                start_i = i+1

finally:
    stdout.write('\x1b[2K\rClosing pool... ')
    stdout.flush()
    time.sleep(1)
    pool.close()
    pool.join()
    del(pool)
    time.sleep(1)
print('Calculations completed successfully. Consolidating data...')
if args.coord_analysis_level == 'residue':
    mi = np.zeros([n_resis]*2)
elif args.coord_analysis_level == 'atom':
    mi = np.zeros([n_atoms]*2)
for i,j,m in data_list:
    if args.coord_analysis_level == 'residue':
        mi[unique_resi_numbers.index(i),unique_resi_numbers.index(j)] = m
    elif args.coord_analysis_level == 'atom':
        mi[i,j] = m
mi += np.triu(mi,1).T
else:
    print('serially.')
    raise ValueError('Serial calculation of MI matrices not yet implemented.')
print('Writing MI data to text and pickling...')
if args.coord_analysis_level == 'residue':
    multiconf_utilities.mat_to_txt(mi, unique_resi_numbers,
os.path.join(args.out_dir,'{}_sorted-resi-coord-mi.txt'.format(args.run_name)), header='{},
residue-level coordinate mutual information', start_diagonal=0)
    with open(os.path.join(args.out_dir,'{}_coord_{}_mi.pkl'.format(args.run_name,
args.coord_analysis_level)), 'wb') as f:
        pickle.dump({'m':mi, 'unique_resi_numbers':unique_resi_numbers},f)
    elif args.coord_analysis_level == 'atom':
        multiconf_utilities.mat_to_txt(mi, atom_names,
os.path.join(args.out_dir,'{}_sorted-atomic-coord-mi.txt'.format(args.run_name)), header='{},
atom-level coordinate mutual information', start_diagonal=0)
        with open(os.path.join(args.out_dir,'{}_coord_{}_mi.pkl'.format(args.run_name,
args.coord_analysis_level)), 'wb') as f:
            pickle.dump({'m':mi, 'atom_names':atom_names, 'resi_numbers':resi_numbers},f)
    print('Done!')
elif args.mi_observable == 'dihedrals':
    print('Calculating dihedrals and associated mutual information in \{}\{}
mode'.format(args.dihedral_analysis_level),end=' ')
    if args.multiprocess:
        print('in parallel over as many as {} cores...'.format(args.n_processors))
        n_resis = len(unique_resi_numbers)
        if n_resis < args.n_processors:
            n_procs_di = n_resis
        else:
            n_procs_di = args.n_processors
        pool = mp.Pool(processes=n_procs_di, maxtasksperchild=10)
        time.sleep(1)

```

```

        try:
            command_gen = ((super_ensemble, reference, [this_resi], None,
args.dihedral_analysis_level, True, False) for this_resi in unique_resi_numbers)
            results = []
            data_list = []
            start_i = 0
            for i, command in enumerate(command_gen):
                results.append(pool.apply_async(get_resi_super_ensemble_dihedrals,
args=command))

                del(command)
                if i > 0 and ((i + 1) % n_procs_di) == 0:
                    stdout.write('\x1b[2K\r\tCalculating dihedrals for residues {}:{}
({:.0%})...'.format(unique_resi_numbers[start_i], unique_resi_numbers[i], i/n_resis))
                    stdout.flush()
                    data_list += [r.get() for r in results]
                    results = []
                    start_i = i+1

        except:
            print('\t\tError calculating dihedrals!')
            raise
        finally:
            stdout.write('\x1b[2K\rClosing pool... ')
            stdout.flush()
            time.sleep(1)
            pool.close()
            pool.join()
            del(pool)
            time.sleep(1)
    del(super_ensemble, reference, lookup)
    dihedrals_dict = {}
    [dihedrals_dict.update(d) for d in data_list]
    del(data_list)
    print('Dihedral calculations complete.\nPerforming MI calculations...')
    n_pairs = int(((n_resis**2 - n_resis)/2) + n_resis)
    if n_pairs < args.n_processors:
        n_procs_mi = n_pairs
    else:
        n_procs_mi = args.n_processors
    pool = mp.Pool(processes=n_procs_mi, maxtasksperchild=10)
    time.sleep(1)
    try:
        command_gen_a = ((dihedrals_dict[int(resi_i)], dihedrals_dict[int(resi_j)],
6, args.mixnyn_path, (int(resi_i), int(resi_j)), args.out_dir, False, True, False) for resi_i,
resi_j in it.combinations(unique_resi_numbers,2))
        command_gen_b = ((dihedrals_dict[int(resi)], dihedrals_dict[int(resi)], 6,
args.mixnyn_path, (int(resi), int(resi)), args.out_dir, False, True, False) for resi in
unique_resi_numbers)
        command_gen = it.chain(command_gen_a, command_gen_b)
        results = []
        data_list = []
        start_i = 0
        for i,command in enumerate(command_gen):

results.append(pool.apply_async(calculate_pairwise_dihedral_mutual_information, args=command))
        del(command)
        if i > 0 and ((i + 1) % n_procs_mi) == 0:
            stdout.write('\x1b[2K\r\tCalculating MI for {} pairs {}:{} of {}
total {:.0%})...'.format(args.dihedral_analysis_level, start_i+1, i+1, n_pairs, i/n_pairs))
            stdout.flush()
            data_list += [r.get() for r in results]
            results = []
            start_i = i+1

    except:
        print('\t\tError calculating dihedral MI!')
        raise
    finally:
        stdout.write('\x1b[2K\rClosing pool... ')
        stdout.flush()
        time.sleep(1)

```

```

        pool.terminate()
        pool.join()
        del(pool)
        time.sleep(1)
    del(dihedrals_dict)
    print('Dihedral MI calculations completed successfully; consolidating data...')
    mi = np.zeros([n_resis]*2)
    for i,j,m in data_list:
        mi[unique_resi_numbers.index(i),unique_resi_numbers.index(j)] = m
    mi += np.triu(mi,1).T
else:
    print('serially.')
    raise ValueError('Serial calculation of dihedral MI matrices not yet
implemented.')
    print('Writing dihedral MI data to text and pickling...')
    multiconf_utilities.mat_to_txt(mi, unique_resi_numbers,
os.path.join(args.out_dir, '{}_sorted-{}-dihedral-mi.txt'.format(args.run_name,
args.dihedral_analysis_level)), header='{} {} dihedral mutual information', start_diagonal=0)
    with open(os.path.join(args.out_dir, '{}_{}_dihedral_mi.pkl'.format(args.run_name,
args.dihedral_analysis_level)), 'wb') as f:
        pickle.dump({'m':mi, 'unique_resi_numbers':unique_resi_numbers},f)
    print('Done!')

    if args.subparser_name == 'pairwisecc':
        if not preload:
            reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=None, chain='A',
selection_string=args.selection_string, run_name=args.run_name)
            with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
                pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup},f)
            resi_numbers = reference.getResnums()
            unique_resi_numbers = [resi_numbers[i] for i in sorted(np.unique(resi_numbers,
return_index=True)[1])]
            atom_names = reference.getNames()
            n_resis = len(unique_resi_numbers)
            print('Calculating pairwise atomic correlation coefficient matrix...')
            pca_dict = super_ensemble_pca(super_ensemble, reference, lookup, project=False,
plot=args.plot, run_name=args.run_name, out_dir=args.out_dir)
            print('Calculating pairwise residue correlation coefficient matrix...')
            _, _, _, resi_corr_mat_reduced, _, resi_lookup =
reduce_sort_correlations_by_residue(pca_dict['pca_corr'], resi_numbers, atom_names,
plot=args.plot, run_name=args.run_name, header=args.run_name, out_dir=args.out_dir)
            print('Pickling...')
            with open(os.path.join(args.out_dir, '{}_corcoef.pkl'.format(args.run_name)), 'wb') as f:
                pickle.dump({'resi_cc_mat':resi_corr_mat_reduced,
'resi_cc_mat_resnums':resi_lookup},f)
            print('Done!')

    if args.subparser_name == 'buildnm':
        sub_out_dir = os.path.join(args.out_dir, '{}_calc_{}'.format(args.run_name, args.nmmodel))
        if not os.path.isdir(sub_out_dir):
            os.mkdir(sub_out_dir)
        nm_dict = build_nm(args.nmpdb, selection_string=args.nmselection, model=args.nmmodel,
n_modes=args.nmmodes, run_name=args.run_name, out_dir=sub_out_dir)
        if args.nmmodel == 'anm':
            xyz = True
        elif args.nmmodel == 'gnm':
            xyz = False
        nm_corr_mat_atoms_reduced, _, _, nm_corr_mat_resis_reduced, _, nm_unique_resis =
reduce_sort_correlations_by_residue(nm_dict['{}_corr'.format(args.nmmodel)],
nm_dict['resi_nums'], nm_dict['atom_names'], xyz=xyz, start_diagonal=0, take_abs=False,
run_name='{}_{}'.format(args.run_name, args.nmmodel), header='{}
Correlations'.format(args.nmmodel.upper()), out_dir=sub_out_dir, plot=args.plot)
        out_dict =
{'nm_corr_mat_atoms':nm_dict['{}_corr'.format(args.nmmodel)], 'nm_resi_nums':nm_dict['resi_nums'],
'nm_atom_names':nm_dict['atom_names'], 'nm_corr_mat_atoms_reduced':nm_corr_mat_atoms_reduced,
'nm_corr_mat_resis_reduced':nm_corr_mat_resis_reduced, 'nm_unique_resis':nm_unique_resis}

```

```

if args.nmcompare is not None:
    if not preload and args.nmcompare is not None:
        reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=None, chain='A',
selection_string=args.nmselection, run_name=args.run_name)
        with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
            pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup},f)
            if reference.getSelstr() != nm_dict['prot'].getSelstr():
                print('PCA and NM selection strings are not identical ({} vs. {}), this may cause
problems!'.format(reference.getSelstr(), args.nmselection))
                pca_resi_nums = reference.getResnums()
                pca_atom_names = reference.getNames()

            print('\nCalculating pairwise atomic correlation coefficient matrix for super-
ensemble...')
            sub_out_dir = os.path.join(args.out_dir, '{}_pca'.format(args.run_name))
            if not os.path.isdir(sub_out_dir):
                os.mkdir(sub_out_dir)
            pca_dict = super_ensemble_pca(super_ensemble, reference, lookup, project=False,
plot=args.plot, run_name=args.run_name, out_dir=sub_out_dir)

            if xyz:
                if pca_dict['pca_corr'].shape != nm_dict['{}_corr'.format(args.nmmodel)].shape
and args.nmmodel == 'anm':
                    raise ValueError('Matrix dimensions must be equal!\npca_dict: {} \nnm_dict:
{}'.format(pca_dict['pca_corr'].shape, nm_dict['{}_corr'.format(args.nmmodel)].shape))
                    print('\nComparing correlation matrix modes from data and ANM in Cartesian
space.'.format(args.nmmodel))
                    sub_out_dir = os.path.join(args.out_dir, '{}_atom_xyz'.format(args.run_name))
                    if not os.path.isdir(sub_out_dir):
                        os.mkdir(sub_out_dir)
                    eig_dict_xyz = compare_modes({'pca_atom_corr':pca_dict['pca_corr'],
'nm_atom_corr':nm_dict['{}_corr'.format(args.nmmodel)], 'resi_nums':nm_dict['resi_nums'],
'atom_names':nm_dict['atom_names']}, n_modes=args.nmmodes, cc_threshold=0.4, start_diag=1,
run_name='{}_atom_xyz'.format(args.run_name), plot=args.plot, out_dir=sub_out_dir)
                    out_dict.update({'eig_dict_xyz':eig_dict_xyz})

                print('\nReducing dimensionality to a single value per atom and calculating pairwise
residue correlation coefficient matrix...')
                sub_out_dir = os.path.join(args.out_dir, '{}_pca'.format(args.run_name))
                if not os.path.isdir(sub_out_dir):
                    os.mkdir(sub_out_dir)
                pca_corr_mat_atoms_reduced, _, _, pca_corr_mat_resis_reduced, _, pca_unique_resis =
reduce_sort_correlations_by_residue(pca_dict['pca_corr'], pca_resi_nums, pca_atom_names,
take_abs=False, plot=args.plot, run_name='{}_pca'.format(args.run_name), header=args.run_name,
out_dir=sub_out_dir)
                if nm_corr_mat_atoms_reduced.shape != pca_corr_mat_atoms_reduced.shape:
                    raise ValueError('Reduced matrices from PCA and NM must have the same shape!')
                out_dict.update({'pca_corr_mat_atoms_reduced':pca_corr_mat_atoms_reduced,
'pca_corr_mat_resis_reduced':pca_corr_mat_resis_reduced, 'pca_unique_resis':pca_unique_resis})

                print('\nComparing modes between reduced atomic data from PCA and from
{}...'.format(args.nmmodel.upper()))
                sub_out_dir = os.path.join(args.out_dir, '{}_atom_mean'.format(args.run_name))
                if not os.path.isdir(sub_out_dir):
                    os.mkdir(sub_out_dir)
                eig_dict_reduced_atomic =
compare_modes({'pca_atom_corr_reduced':pca_corr_mat_atoms_reduced, 'nm_atom_corr_reduced':nm_corr_
mat_atoms_reduced, 'resi_nums':pca_resi_nums, 'atom_names':pca_atom_names},
n_modes=args.nmmodes, start_diag=1, run_name='{}_atom_mean'.format(args.run_name),
plot=args.plot, out_dir=sub_out_dir)
                out_dict.update({'eig_dict_reduced_atomic':eig_dict_reduced_atomic})

                print('\nComparing modes between reduced resi data from PCA and from
{}...'.format(args.nmmodel.upper()))
                sub_out_dir = os.path.join(args.out_dir, '{}_resi_mean'.format(args.run_name))

```

```

        if not os.path.isdir(sub_out_dir):
            os.mkdir(sub_out_dir)
        eig_dict_reduced_resis =
compare_modes({'pca_resi_corr_reduced':pca_corr_mat_resis_reduced,'nm_resi_corr_reduced':nm_corr_
mat_resis_reduced, 'resi_nums':pca_unique_resis,
'atom_names':np.array(['R']*len(pca_unique_resis))}, n_modes=args.nmmodes, start_diag=1,
run_name='{}_resi_mean'.format(args.run_name), plot=args.plot, out_dir=sub_out_dir)
        out_dict.update({'eig_dict_reduced_resis':eig_dict_reduced_resis})

    with open(os.path.join(args.out_dir, '{}_{}.pkl'.format(args.run_name,args.nmmodel)), 'wb')
as f:
        pickle.dump(out_dict,f)

    if args.subparser_name == 'covconverge':
        if not preload:
            reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=args.subset, chain=args.chain,
selection_string=args.selection_string, run_name=args.run_name)
            with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
                pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup},f)
            print('\nPerforming covariance and correlation coefficient matrix convergence
calculations',end=' ')
            if args.multiprocess and args.ccrepeats > 1:
                if args.ccrepeats < args.n_processors:
                    args.n_processors = args.ccrepeats
                print('over as many as {} cores.'.format(args.n_processors))
                pool = mp.Pool(processes=args.n_processors, maxtasksperchild=10)
                #pool = mp.Pool(processes=args.n_processors)
                time.sleep(1)
                try:
                    command_gen = ((super_ensemble, None, args.ccsample, (True,1), False,
args.run_name, None, False) for i in range(args.ccrepeats))
                    results = []
                    data_list = []
                    start_i = 0
                    for i,command in enumerate(command_gen):
                        print('Calculating convergence for repeat {} of {} ( {:.0%})...'.format(i,
args.ccrepeats-1, i/(args.ccrepeats-1)))
                        results.append(pool.apply_async(super_ensemble_covariance_convergence,
args=command))
                        del(command)
                        if i > 0 and ((i + 1) % args.n_processors) == 0:
                            data_list += [r.get() for r in results]
                            results = []
                            start_i = i+1
                    # results =
[pool.apply_async(super_ensemble_covariance_convergence,args=(super_ensemble, None, True,
(True,1), False, args.run_name, None)) for i in range(args.ccrepeats)]
                    # data_list = [p.get() for p in results]
                    cov_fnorm_diff = np.hstack([data_list[i][0] for i in range(len(data_list))])
                    corr_fnorm_diff = np.hstack([data_list[i][1] for i in range(len(data_list))])
                    tau = np.hstack([data_list[i][2] for i in range(len(data_list))])
                finally:
                    print('Closing pool... ')
                    time.sleep(1)
                    pool.terminate()
                    pool.join()
                    del(pool)
                    time.sleep(1)
                if os.path.isdir(args.out_dir):
                    pickle.dump({'cov_fnorm_diff':cov_fnorm_diff, 'corr_fnorm_diff':corr_fnorm_diff,
'tau':tau},open(os.path.join(args.out_dir, '{}_cov-corr-
convergence.pkl'.format(args.run_name)), 'wb'))
                else:
                    print('\nInvalid output directory specified, will not pickle data.')
            else:
                print('serially.')

```

```

        cov_fnorm_diff, corr_fnorm_diff, tau =
super_ensemble_covariance_convergence(Super_ensemble, subsample=args.ccsample, split=(True,
args.ccrepeats), plot=False, run_name=args.run_name, out_dir = args.out_dir, verbose=True)
    print('Analyzing convergence and plotting...')
    print('<tau> = {:.2} +/- {:.2}'.format(tau.mean(),tau.std()))
    plot_super_ensemble_covariance_convergence(corr_fnorm_diff,tau_tuple=(tau.mean(),
super_ensemble.numAtoms()*3, super_ensemble.numCoordsets()//2), out_dir=args.out_dir,
data_label='corr_mat')
    print('Done!')

    if args.subparser_name == 'projection':
        print('\nAnalyzing conformational space of PCA modes...')
        if not preload:
            reference, super_ensemble, lookup =
build_prody_super_ensemble(ensemble_list_distilled, subset=args.subset, chain=args.chain,
selection_string=args.selection_string, run_name=args.run_name)
            with open(os.path.join(args.out_dir, '{}_prody-
objects.pkl'.format(args.run_name)), 'wb') as f:
                pickle.dump({'reference':reference, 'super_ensemble':super_ensemble,
'lookup':lookup},f)

pca,_,_,_,proj_cov,proj_corr,proj_corr_supp=analyze_superensemble.super_ensemble_pca(super_ense
mble, reference, lookup, project=True, plot=True, run_name=args.run_name, out_dir=args.out_dir)
    for key, val in {'proj_cov':proj_cov, 'proj_corr':proj_corr,
'proj_corr_supp':proj_corr_supp}:
        visualize_projection(val, lookup, args.out_dir, '{}_{}'.format(run_name,key),
percentiles=args.percentiles, n_models=args.n_models, n_modes=args.n_modes)

    print('Elapsed time was {} seconds.'.format(time.time() - start_time))

```

1.2.7 create_mi_dict.py

This (ugly) set of classes creates a unified dictionary with the PDZ MSA and pairwise sequence identities for all proteins, as well as all coordinate and dihedral mutual information matrices, distance matrices, and contact graphs.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, pickle, json
from Bio import AlignIO

...

Create dictionary for a given .

Dict structure template:

...

class create_pdz_mi_dict_round5():
def __init__(self):
    self.name = 'pdz'
    self.miDict = dict(

```

```

consensus = dict(
  alignments = dict(
    pdz=AlignIO.read('/Volumes/Alpha./pdz_alignment_full_ligfix_v2.fasta','fasta')#
    Path to alignment file or Bio.Align MSA
  ),
  seq_id = dict(
    psd95pdz3_cript_rt = dict(
      sj2bppdz_pseudo = 0.44,
      mpdzpdz3_pseudo = 0.29,
      lnx2pdz2_pseudo = 0.28,
      pdlim7pdz_pseudo = 0.21,
      pick1pdz1_pseudo = 0.22,
      syntenin1pdz2_pseudo = 0.22,
      syntenin1pdz2_syndecan = 0.22,
      tiam1pdz_apo = 0.16,
      erbinshortpdz_apo = 0.31
    ),
    sj2bppdz_pseudo = dict(
      mpdzpdz3_pseudo = 0.36,
      lnx2pdz2_pseudo = 0.28,
      pdlim7pdz_pseudo = 0.28,
      pick1pdz1_pseudo = 0.22,
      syntenin1pdz2_pseudo = 0.17,
      syntenin1pdz2_syndecan = 0.17,
      tiam1pdz_apo = 0.21,
      erbinshortpdz_apo = 0.32
    ),
    mpdzpdz3_pseudo = dict(
      lnx2pdz2_pseudo = 0.28,
      pdlim7pdz_pseudo = 0.20,
      pick1pdz1_pseudo = 0.19,
      syntenin1pdz2_pseudo = 0.18,
      syntenin1pdz2_syndecan = 0.18,
      tiam1pdz_apo = 0.10,
      erbinshortpdz_apo = 0.28
    ),
    lnx2pdz2_pseudo = dict(
      pdlim7pdz_pseudo = 0.30,
      pick1pdz1_pseudo = 0.24,
      syntenin1pdz2_pseudo = 0.21,
      syntenin1pdz2_syndecan = 0.21,
      tiam1pdz_apo = 0.13,
      erbinshortpdz_apo = 0.19
    ),
    pdlim7pdz_pseudo = dict(
      pick1pdz1_pseudo = 0.22,
      syntenin1pdz2_pseudo = 0.16,
      syntenin1pdz2_syndecan = 0.16,
      tiam1pdz_apo = 0.17,
      erbinshortpdz_apo = 0.19
    ),
    pick1pdz1_pseudo = dict(
      syntenin1pdz2_pseudo = 0.18,
      syntenin1pdz2_syndecan = 0.18,
      tiam1pdz_apo = 0.14,
      erbinshortpdz_apo = 0.18
    ),
    syntenin1pdz2_pseudo = dict(
      syntenin1pdz2_syndecan = 1.0,
      tiam1pdz_apo = 0.17,
      erbinshortpdz_apo = 0.17
    ),
    syntenin1pdz2_syndecan = dict(
      tiam1pdz_apo = 0.17,
      erbinshortpdz_apo = 0.17
    ),
    tiam1pdz_apo = dict(
      erbinshortpdz_apo = 0.19
    )
  )
)

```



```

ds4_1 = dict(
    data = dict(
        dihedrals = dict(
            matrices = dict(
                m =
pickle.load(open('/Volumes/Alpha./pdz3_cript/ERMI41_both_MI.pickle', 'rb'), encoding='latin1')['MI_
a'][-127:,-127:]
            )
        ),
        coordinates = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/pdz3-
cript_ds41_prodifined_mi_filtered_fixed.pkl', 'rb'), encoding='latin1')['m'][-127:,-127:],
                r = pickle.load(open('./pdz3_cript/corrcoef/pdz3-
cript_ds41_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc_mat'][-127:,-127:]
            )
        ),
        contacts = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/cmap_ens_full/pdz3-
cript_ds41_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'][-127:,-127:],
            )
        ),
        distance = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/cmap_full/ds4-
1/pdz3_cript_ds4-1_refine_167_chain-merge_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
            )
        )
    )
),
ds4_2 = dict(
    data = dict(
        dihedrals = dict(
            matrices = dict(
                m =
pickle.load(open('/Volumes/Alpha./pdz3_cript/ERMI42_both_MI.pickle', 'rb'), encoding='latin1')['MI_
a'][-127:,-127:]
            )
        ),
        coordinates = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/pdz3-
cript_ds42_prodifined_mi_filtered_fixed.pkl', 'rb'), encoding='latin1')['m'][-127:,-127:],
                r = pickle.load(open('./pdz3_cript/corrcoef/pdz3-
cript_ds42_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc_mat'][-127:,-127:]
            )
        ),
        contacts = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/cmap_ens_full/pdz3-
cript_ds42_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'][-127:,-127:],
            )
        ),
        distance = dict(
            matrices = dict(
                m = pickle.load(open('./pdz3_cript/cmap_full/ds4-
2/pdz3_cript_ds4-2_refine_85_chain-merge_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
            )
        )
    )
),
ds4_3 = dict(
    data = dict(
        dihedrals = dict(
            matrices = dict(
                m =
pickle.load(open('/Volumes/Alpha./pdz3_cript/ERMI43_both_MI.pickle', 'rb'), encoding='latin1')['MI_
a'][-127:,-127:]
            )
        )
    )
)

```

```

    )
    ),
    coordinates = dict(
        matrices = dict(
            m = pickle.load(open('./pdz3_cript/pdz3-
cript_ds43_prodifined_mi_filtered_fixed.pkl', 'rb'), encoding='latin1')['m'][-127:,-127:],
            r = pickle.load(open('./pdz3_cript/corrcoef/pdz3-
cript_ds43_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc_mat'][-127:,-127:]
        )
    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./pdz3_cript/cmap_ens_full/pdz3-
cript_ds43_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'][-127:,-127:],
        )
    ),
    distance = dict(
        matrices = dict(
            m = pickle.load(open('./pdz3_cript/cmap_full/ds4-
3/pdz3_cript_ds4-3_refine_32_chain-merge_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
        )
    )
    )
    ),
    ds5_1 = dict(
        data = dict(
            dihedrals = dict(
                matrices = dict(
                    m =
pickle.load(open('/Volumes/Alpha./pdz3_cript/ERMI51_both_MI.pickle', 'rb'), encoding='latin1')['MI_
a'][-127:,-127:]
                )
            ),
            coordinates = dict(
                matrices = dict(
                    m = pickle.load(open('./pdz3_cript/pdz3-
cript_ds51_prodifined_mi_filtered_fixed.pkl', 'rb'), encoding='latin1')['m'][-127:,-127:],
                    r = pickle.load(open('./pdz3_cript/corrcoef/pdz3-
cript_ds51_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc_mat'][-127:,-127:]
                )
            ),
            contacts = dict(
                matrices = dict(
                    m = pickle.load(open('./pdz3_cript/cmap_ens_full/pdz3-
cript_ds11_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'][-127:,-127:],
                )
            ),
            distance = dict(
                matrices = dict(
                    m = pickle.load(open('./pdz3_cript/cmap_full/ds5-
1/pdz3_cript_ds5-1_refine_28_chain-merge_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
                )
            )
        )
    )
    )
    ),
    syntenin1pdz2_pseudo = dict(
        resiIndices = [273]+list(range(192,273)),
        sets = dict(
            ds1_1 = dict(
                data = dict(
                    dihedrals = dict(
                        matrices = dict(
                            m =
pickle.load(open('/Volumes/Alpha./synt_apo/ERMI11_both_MI_fix_roll.pickle', 'rb'), encoding='latin1
')['MI_a']
                        )
                    )
                )
            )
        )
    ),

```

```

        coordinates = dict(
            matrices = dict(
                m = pickle.load(open('./synt_apo/synt-
apo_ds11_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                r = pickle.load(open('./synt_apo/corrcoef/synt-
apo_ds11_corcoef.pkl', 'rb'), encoding='latin1')['resi_cc_mat']
            )
        ),
        contacts = dict(
            matrices = dict(
                m = pickle.load(open('./synt_apo/cmap_ensemble/synt-apo_ds1-
1_secmbin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
            )
        ),
        distance = dict(
            matrices = dict(
                m = pickle.load(open('./synt_apo/cmap/ds1-1/syntenin_apo_ds1-
1_refine_95_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
            )
        )
    ),
    ds2_1 = dict(
        data = dict(
            dihedrals = dict(
                matrices = dict(
                    m =
pickle.load(open('/Volumes/Alpha./synt_apo/ERMI21_both_MI_fix_roll.pickle', 'rb'), encoding='latin1
')['MI_a']
                )
            ),
            coordinates = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_apo/synt-
apo_ds21_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                    r = pickle.load(open('./synt_apo/corrcoef/synt-
apo_ds21_corcoef.pkl', 'rb'), encoding='latin1')['resi_cc_mat']
                )
            ),
            contacts = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_apo/cmap_ensemble/synt-apo_ds2-
1_secmbin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
                )
            ),
            distance = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_apo/cmap/ds2-1/syntenin_apo_ds2-
1_refine_20_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
                )
            )
        )
    ),
    ds3_1 = dict(
        data = dict(
            dihedrals = dict(
                matrices = dict(
                    m =
pickle.load(open('/Volumes/Alpha./synt_apo/ERMI31_both_MI_fix_roll.pickle', 'rb'), encoding='latin1
')['MI_a']
                )
            ),
            coordinates = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_apo/synt-
apo_ds31_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                    r = pickle.load(open('./synt_apo/corrcoef/synt-
apo_ds31_corcoef.pkl', 'rb'), encoding='latin1')['resi_cc_mat']
                )
            )
        )
    )

```

```

    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./synt_apo/cmap_ensemble/synt-apo_ds3-
1_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
        )
    ),
    distance = dict(
        matrices = dict(
            m = pickle.load(open('./synt_apo/cmap/ds3-1/syntenin_apo_ds3-
1_refine_21_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
        )
    )
)
),
syntenin1pdz2_syndecan = dict(
    resiIndices = list(range(3,9))+list(range(192,271)),
    sets = dict(
        ds3_1 = dict(
            data = dict(
                dihedrals = dict(
                    matrices = dict(
                        m =
pickle.load(open('/Volumes/Alpha./synt_synd/ERMI31_both_MI_fix.pickle', 'rb'), encoding='latin1')['
MI_a']
                    )
                ),
            coordinates = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_synd/synt-
synd_ds31_prodfied_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                    r = pickle.load(open('./synt_synd/corrcoef/synt-
synd_ds31_corcoef.pkl', 'rb'), encoding='latin1')['resi_cc_mat']
                )
            ),
            contacts = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_synd/cmap_ensemble/synt-
synd_ds3-1_secm_bin-true.pkl', 'rb'), encoding='latin1')['secm_mean'],
                )
            ),
            distance = dict(
                matrices = dict(
                    m = pickle.load(open('./synt_synd/cmap/syntenin_syndecan_ds3-
1_refine_43_fixed_ligmerge_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
                )
            )
        )
    )
),
pdlim7pdz_pseudo = dict(
    resiIndices = list(range(85,89))+list(range(0,85)),
    sets = dict(
        ds2_1 = dict(
            data = dict(
                dihedrals = dict(
                    matrices = dict(
                        m =
pickle.load(open('/Volumes/Alpha./sgc1/ERMI21_both_MI_roll.pickle', 'rb'), encoding='latin1')['MI_a
']
                    )
                ),
            coordinates = dict(
                matrices = dict(
                    m =
pickle.load(open('./sgc1/sgc1_ds21_prodfied_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],

```

```

        r =
pickle.load(open('./sgc1/corrcoef/sgc1_ds21_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc
_mat']
    )
    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./sgc1/cmap_ensemble/sgc1_ds2-
1_secm_bin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
        )
    ),
    distance = dict(
        matrices = dict(
            m = pickle.load(open('./sgc1/cmap/sgc1_ds2-
1_refine_20_fixed_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
        )
    )
    )
    )
    ),
    tiam1pdz_apo = dict( # Note, this is an apo structure despite being listed otherwise
        resiIndices = list(range(839,931)),
        sets = dict(
            ds1_1 = dict(
                data = dict(
                    dihedrals = dict(
                        matrices = dict(
                            m =
pickle.load(open('/Volumes/Alpha./tiam1_synd/ERMI11_both_MI.pickle', 'rb'), encoding='latin1')['MI_
a']
                        )
                    ),
                    coordinates = dict(
                        matrices = dict(
                            m = pickle.load(open('./tiam1_synd/tiam1-
apo_ds21_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                            r = pickle.load(open('./tiam1_synd/corrcoef/tiam-
apo_ds1_corcoef.pkl', 'rb'), encoding='latin1')['resi_cc_mat']
                        )
                    ),
                    contacts = dict(
                        matrices = dict(
                            m = pickle.load(open('./tiam1_synd/cmap_ensemble/tiam1-
apo_ds1_secm_bin-true.pkl', 'rb'), encoding='latin1')['secm_mean'],
                        )
                    ),
                    distance = dict(
                        matrices = dict(
                            m =
pickle.load(open('./tiam1_synd/cmap/Tiam1_syndecan1_ds1_refine_63_contacts.pkl', 'rb'), encoding='l
atin1')['D_raw_sub']
                        )
                    )
                )
            )
        )
    ),
    pick1pdz1_pseudo = dict( # Note, this is an apo structure despite being listed
otherwise
        resiIndices = list(range(106,110))+list(range(17,106)),
        sets = dict(
            ds1_1 = dict(
                data = dict(
                    dihedrals = dict(
                        matrices = dict(
                            m =
pickle.load(open('/Volumes/Alpha./sgc3/ERMI11_both_MI_roll.pickle', 'rb'), encoding='latin1')['MI_a
']
                        )
                    )
                )
            )
        )
    )

```

```

    )
    ),
    coordinates = dict(
        matrices = dict(
            m =
pickle.load(open('./sgc3/sgc3_277k_ds11_prodifined_rfilter_mi_fixed.pkl','rb'),encoding='latin1')['m'],
            r =
pickle.load(open('./sgc3/corrcoef/sgc3_277K_ds11_corcoef_rolled.pkl','rb'),encoding='latin1')['resi_cc_mat']
        )
    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./sgc3/cmap_ensemble/sgc3_ds1-
1_secm_bin-true_rolled.pkl','rb'),encoding='latin1')['secm_mean'],
        )
    ),
    distance = dict(
        matrices = dict(
            m = pickle.load(open('./sgc3/cmap/sgc3_277k_ds1-
1_refine_42_contacts.pkl','rb'),encoding='latin1')['D_raw_sub']
        )
    )
    )
    )
    ),
    lnx2pdz2_pseudo = dict(
        resiIndices = list(range(425,429))+list(range(-1,1))+list(range(336,425)),
        sets = dict(
            ds1_1 = dict(
                data = dict(
                    dihedrals = dict(
                        matrices = dict(
                            m =
pickle.load(open('/Volumes/Alpha./sgc5/ERMI11_both_MI_fix_roll.pickle','rb'),encoding='latin1')['MI_a']
                        )
                    ),
                    coordinates = dict(
                        matrices = dict(
                            m =
pickle.load(open('./sgc5/sgc5_ds11_prodifined_rfilter_mi_fixed.pkl','rb'),encoding='latin1')['m'],
                            r =
pickle.load(open('./sgc5/corrcoef/sgc5_ds1_corcoef_rolled.pkl','rb'),encoding='latin1')['resi_cc_mat']
                        )
                    ),
                    contacts = dict(
                        matrices = dict(
                            m = pickle.load(open('./sgc5/cmap_ensemble/sgc5_ds1-
1_secm_bin-true_rolled.pkl','rb'),encoding='latin1')['secm_mean'],
                        )
                    ),
                    distance = dict(
                        matrices = dict(
                            m =
pickle.load(open('./sgc5/cmap/ds1/SGC5_1_merged_refine_85_contacts.pkl','rb'),encoding='latin1')['D_raw_sub']
                        )
                    )
                )
            )
        ),
        ds3_1 = dict(
            data = dict(
                dihedrals = dict(
                    matrices = dict(
                        m =

```

```

pickle.load(open('/Volumes/Alpha./sgc5/ERMI31_both_MI_fix_roll.pickle','rb'),encoding='latin1')['
MI_a']
    )
    ),
    coordinates = dict(
        matrices = dict(
            m =
pickle.load(open('./sgc5/sgc5_ds31_prodified_rfilter_mi_fixed.pkl','rb'),encoding='latin1')['m'],
            r =
pickle.load(open('./sgc5/corrcoef/sgc5_ds3_corcoef_rolled.pkl','rb'),encoding='latin1')['resi_cc_
mat']
        )
    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./sgc5/cmap_ensemble/sgc5_ds3-
1_secm_bin-true_rolled.pkl','rb'),encoding='latin1')['secm_mean'],
        )
    ),
    distance = dict(
        matrices = dict(
            m =
pickle.load(open('./sgc5/cmap/ds3/SGC5_3_set2_refine_115_contacts.pkl','rb'),encoding='latin1')['
D_raw_sub']
        )
    )
    ),
    ds4_1 = dict(
        data = dict(
            dihedrals = dict(
                matrices = dict(
                    m =
pickle.load(open('/Volumes/Alpha./sgc5/ERMI41_both_MI_fix_roll.pickle','rb'),encoding='latin1')['
MI_a']
                )
            ),
            coordinates = dict(
                matrices = dict(
                    m =
pickle.load(open('./sgc5/sgc5_ds41_prodified_rfilter_mi_fixed.pkl','rb'),encoding='latin1')['m'],
                    r =
pickle.load(open('./sgc5/corrcoef/sgc5_ds4_corcoef_rolled.pkl','rb'),encoding='latin1')['resi_cc_
mat']
                )
            ),
            contacts = dict(
                matrices = dict(
                    m = pickle.load(open('./sgc5/cmap_ensemble/sgc5_ds4-
1_secm_bin-true_rolled.pkl','rb'),encoding='latin1')['secm_mean'],
                )
            ),
            distance = dict(
                matrices = dict(
                    m =
pickle.load(open('./sgc5/cmap/ds4/SGC5_4_merged_refine_25_contacts.pkl','rb'),encoding='latin1')['
D_raw_sub']
                )
            )
        )
    ),
    mpdzpdz3_pseudo = dict(
        resiIndices = list(range(464,468)) + list(range(371,464)),
        sets = dict(
            ds1_1 = dict(
                data = dict(
                    dihedrals = dict(

```

```

        matrices = dict(
            m =
pickle.load(open('/Volumes/Alpha./sgc7/ERMI11_both_MI_roll.pickle', 'rb'), encoding='latin1')['MI_a
']
        )
    ),
    coordinates = dict(
        matrices = dict(
            m =
pickle.load(open('./sgc7/sgc7_ds11_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
            r =
pickle.load(open('./sgc7/corrcoef/sgc7_ds11_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc
_mat']
        )
    ),
    contacts = dict(
        matrices = dict(
            m = pickle.load(open('./sgc7/cmap_ensemble/sgc7_ds1-
1_secmbin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
        )
    ),
    distance = dict(
        matrices = dict(
            m = pickle.load(open('./sgc7/cmap/ds1-1/sgc7_277k_ds1-
1_refine_27_fixed_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
        )
    )
)
),
),
sj2bppdz_pseudo = dict(
    resiIndices = list(range(101,105))+list(range(-1,1)) + list(range(6,101)),
    sets = dict(
        ds1_1 = dict(
            data = dict(
                dihedrals = dict(
                    matrices = dict(
                        m =
pickle.load(open('/Volumes/Alpha./sgc8/ERMI11_both_MI_fix_roll.pickle', 'rb'), encoding='latin1')['
MI_a']
                    )
                ),
                coordinates = dict(
                    matrices = dict(
                        m =
pickle.load(open('./sgc8/sgc8_ds11_prodifined_rfilter_mi_fixed.pkl', 'rb'), encoding='latin1')['m'],
                        r =
pickle.load(open('./sgc8/corrcoef/sgc8_ds11_corcoef_rolled.pkl', 'rb'), encoding='latin1')['resi_cc
_mat']
                    )
                ),
                contacts = dict(
                    matrices = dict(
                        m = pickle.load(open('./sgc8/cmap_ensemble/sgc8_ds1-
1_secmbin-true_rolled.pkl', 'rb'), encoding='latin1')['secm_mean'],
                    )
                ),
                distance = dict(
                    matrices = dict(
                        m = pickle.load(open('./sgc8/cmap/sgc8_2jin_ds1-
1_refine_8_fixed_contacts.pkl', 'rb'), encoding='latin1')['D_raw_sub']
                    )
                )
            )
        )
    )
),
erbinshortpdz_apo = dict(

```

```

resiIndices = list(range(1318,1413)),
sets = dict(
    ds1_1 = dict(
        data = dict(
            dihedrals = dict(
                matrices = dict(
                    m =
pickle.load(open('/Volumes/Alpha./erbin_apo/ERMI11_both_MI.pickle','rb'),encoding='latin1')['MI_a
']
                )
            ),
            coordinates = dict(
                matrices = dict(
                    m = pickle.load(open('./erbin_apo/erbin-
apo_ds21_prodifined_rfilter_mi_fixed.pkl','rb'),encoding='latin1')['m'],
                    r = pickle.load(open('./erbin_apo/corcoef/erbin-
apo_ds21_corcoef.pkl','rb'),encoding='latin1')['resi_cc_mat']
                )
            ),
            contacts = dict(
                matrices = dict(
                    m = pickle.load(open('./erbin_apo/cmap_ensemble/erbin-
apo_ds2-1_secm_bin-true.pkl','rb'),encoding='latin1')['secm_mean'],
                )
            ),
            distance = dict(
                matrices = dict(
                    m = pickle.load(open('./erbin_apo/cmap/ds2-1/erbin_apo_ds2-
1_refine_28_fixed_contacts.pkl','rb'),encoding='latin1')['D_raw_sub']
                )
            )
        )
    )
)
),
),
),
)
def save(self,outDir):
    if os.path.isdir(outDir):
pickle.dump(self.miDict,open(os.path.join(outDir,'miDict_{}.pkl'.format(self.name)),'wb'))
    else:
        print('Invalid directory!')

```

1.2.8 SymNMF.py

This file contains functions needed to perform symmetric non-negative matrix factorization. The core algorithm was inspired by the MATLAB code published by Da Kuang, Chris Ding, Haesun Park.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import numpy as np

```

```

from sklearn.decomposition.nmf import _initialize_nmf as initialize_nmf

def symnmf_newton(A, k, Hinit=None, max_iter=10000, tol=0.0001, sigma=0.1, beta=0.1, debug_lvl=0,
                 rand_mat=None):
    """
    The function symnmf_newton uses a Netwon-like algorithm to perform symmetric non-negative
    matrix factorization (SymNMF) on an input matrix.
    To do so, it optimizes the functional:
    min_H f(H) = ||A - HH'||_F^2 subject to H >= 0

    This approach was reported in the following paper and its associated code:
    Da Kuang, Chris Ding, Haesun Park. Symmetric Nonnegative Matrix Factorization for Graph
    Clustering. The 12th SIAM International Conference on Data Mining (SDM '12), pp. 106--117.

    Input:
    A: An n x n symmetric, non-negative matrix. Must be a numpy ndarray.
    k: Rank of factorization, typically k much less than n.
    Hinit: Initialization of H with dimensions n x k; by default, a random matrix is used to
    create this. If equal to a string in (nndsvd, nndslda, nndsldar), will use NNDSVD to generate
    Hinit instead.
    max_iter: Maximum number of iterations for the algorithm; default is 10000.
    tol: The tolerance parameter mu from the cited paper; used to determine convergence and
    terminate the algorithm. Defaults to 1e-4.
    sigma: The acceptance parameter sigma from the cited paper, to be used in the Armijo
    rule; default is 0.1.
    beta: The reduction factor beta from the cited paper; used to decrease the step size of
    the gradient search. Default is 0.1.
    debug_lvl: A value from (0, 1, 2); no output for 0 and increasing output for 1 and 2.
    rand_mat: An n x k random matrix to use in creating Hinit.

    Output:
    H: The low-rank n x k nonnegative matrix which approximates A.
    i: The number of iterations which were performed before termination.
    obj: Objective value f(H) at the final solution H.
    """
    def hessian_blkdiag(temp, H, idx, projnorm_idx):
        n, k=np.shape(H)
        subset = projnorm_idx[:, idx]
        hidx = H[subset, idx].T
        eye = np.dot(np.dot(H[:, idx].conj().T, H[:, idx]), np.eye(n))
        He = 4. * (temp[subset, :][:, subset] + np.outer(hidx, hidx) + eye[subset, :][:, subset])
        return(He)

    if debug_lvl > 0:
        print('\nInitializing SymNMF protocol...')

    # Check input data
    if not isinstance(A, np.ndarray):
        raise ValueError('Matrix A must be a numpy ndarray!')
    if A.ndim != 2:
        raise ValueError('Input matrix must be two dimensional!')
    if A.shape[0] == A.shape[1]:
        n = A.shape[0]
    else:
        raise ValueError('Input matrix must be symmetric!')

    # Make sure k is sane
    if type(k) != int:
        raise ValueError('k must be an integer!')
    elif k < 1:
        raise ValueError('k must be at least 1!')
    elif k > n:
        raise ValueError('k must be less than or equal to n!')

    # Generate Hinit if not supplied
    if Hinit in (None, 'random'): # If none, create a new random matrix
        if debug_lvl > 0:
            print('\tInitializing NMF procedure with randomized matrix.')
        if rand_mat is None:
            H = 2. * (np.sqrt( np.mean(np.mean(A)) / k) * np.random.rand(n, k))

```

```

elif isinstance(rand_mat, np.ndarray):
    if rand_mat.shape == (n, k):
        H = 2. * (np.sqrt( np.mean(np.mean(A)) / k) * rand_mat)
    else:
        raise ValueError('rand_mat must have dimensions n x k!')
    else:
        raise ValueError('rand_mat must be a numpy ndarray!')
Hinit = H.copy()
elif type(Hinit) is str: # Maybe its a command mode
    if Hinit == 'nndsvd':
        H, _ = initialize_nmf(A, k)
    elif Hinit == 'nndsvda':
        H, _ = initialize_nmf(A, k, variant = 'a')
    elif Hinit == 'nndsvdar':
        H, _ = initialize_nmf(A, k, variant = 'ar')
    else:
        raise ValueError('Initialization string must be nndsvd, nndsvda, or nndsvdar.')
Hinit = H.copy()
else:
    if not isinstance(Hinit, np.ndarray):
        raise ValueError('Hinit must be None, a string specifying operation mode (nndsvd,
nndsvda, nndsvdar), or a numpy ndarray!')
    n, kH = np.shape(Hinit)
    if n != np.shape(A)[0]:
        raise ValueError('A and kwarg Hinit must have the same number of rows!')
    if kH != k:
        raise ValueError('The kwarg Hinit must have k columns!')
    H = Hinit

# Make sure specified params are okay
if type(max_iter) != int or max_iter < 0:
    raise ValueError('The total number of iterations specified by max_iter must be a positive
integer!')
if type(tol) != float or tol < 0:
    raise ValueError('The tolerance parameter tol must be a positive float!')
if type(sigma) != float or sigma < 0:
    raise ValueError('The acceptance parameter sigma must be a positive float!')
if type(beta) != float or beta < 0:
    raise ValueError('The reduction factor beta must be a positive float!')
if debug_lvl not in (0, 1, 2):
    raise ValueError('Invalid debug level specified; choose from 0, 1, or 2.')

# Initialize working objects
projnorm_idx = np.zeros((n, k), dtype=bool)
R = {}
p = np.zeros(k)
obj = np.square(np.linalg.norm(A - np.dot(H, H.conj().T), ord='fro'))
gradH = 4.*(np.dot(H, (np.dot(H.conj().T, H)) - np.dot(A, H)) - np.dot(H, H))
initgrad = np.linalg.norm(gradH, ord='fro')
epsilon = np.spacing(1)
if debug_lvl > 0:
    print('\nOptimizing...\n\tinit grad norm = {}'.format(initgrad))

# Optimize...
for i in range(max_iter):
    gradH = 4.*(np.dot(H, (np.dot(H.conj().T, H)) - np.dot(A, H)) - np.dot(H, H))
    projnorm_idx_prev = projnorm_idx
    projnorm_idx = np.logical_or(gradH <= epsilon, H > epsilon)
    projnorm = np.linalg.norm(gradH[projnorm_idx], ord=2)
    if projnorm < tol * initgrad:
        if debug_lvl > 0:
            print('\tfinal grad norm = {}'.format(projnorm))
        break
    else:
        if debug_lvl > 1:
            print('\titer {}: grad norm = {}'.format(i, projnorm))
if i % 100 == 0:
    p = np.ones(k)
this_step = np.zeros((n, k))

```

```

hessian = {}
temp = np.dot(H, H.conj().T) - A
for j in range(k):
    if any(projnorm_idx_prev[:, j] != projnorm_idx[:, j]): # if any positions between the
two matrices are different...
        hessian[j] = hessian_blkdiag(temp, H, j, projnorm_idx)
        try:
            R[j]=np.linalg.cholesky(hessian[j]).T
            p[j] = 0
        except np.linalg.linalg.LinAlgError as err:
            if 'positive definite' in err.args[0]:
                p[j] = 1
            else:
                raise
    if p[j] > 0:
        this_step[:, j] = gradH[:, j]
    else:
        this_step_temp = np.linalg.solve(R[j].conj().T, gradH[projnorm_idx[:, j], j])
        this_step_temp = np.linalg.solve(R[j], this_step_temp)
        this_step_part = np.zeros(n)
        this_step_part[projnorm_idx[:, j]] = this_step_temp
        this_step_part[np.logical_and(this_step_part < -epsilon, H[:, j] <= epsilon)] = 0
        if np.sum(gradH[:, j] * this_step_part) / np.linalg.norm(gradH[:, j], ord=2) /
np.linalg.norm(this_step_part, ord=2) <= epsilon:
            p[j] = 1
            this_step[:, j] = gradH[:, j]
        else:
            this_step[:, j] = this_step_part[:]
    alpha_newton = 1;
    Hn = (H - (alpha_newton * this_step)).clip(0)
    newobj = np.square(np.linalg.norm(A-np.dot(Hn, Hn.conj().T), ord='fro'))
    if newobj - obj > sigma * np.sum(np.sum(gradH * (Hn - H))):
        while True:
            alpha_newton = alpha_newton * beta
            Hn = (H - (alpha_newton*this_step)).clip(0)
            newobj = np.square(np.linalg.norm(A-np.dot(Hn, Hn.conj().T), ord='fro'))
            if newobj - obj <= sigma * np.sum(np.sum(gradH * (Hn - H))):
                H = Hn
                obj = newobj
                break
    else:
        H = Hn
        obj = newobj
if debug_lvl > 0:
    print('Finished!\n')
return(Hinit, H, i, obj)

```

1.2.9 analyze_ermi.py

Compares all of the things in the data object created by create_mi_dict.py in many ways.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, argparse, pickle, glob, copy
import itertools as it
import numpy as np
import networkx as nx

```

```

#import multiprocessing as mp
from SymNMF import symnmf_newton
from multiconf_utilities import kth_diag_indices, triu_corr_coef, triu_cosine, exp_func,
cmap_bounds, simple_rescale
from Bio import Align, AlignIO, PDB
from uuid import uuid4
from sklearn.cluster import AffinityPropagation, AgglomerativeClustering, SpectralClustering
from sklearn.decomposition import PCA, FastICA, ProjectedGradientNMF
from sklearn.preprocessing import StandardScaler, normalize
from sklearn import manifold
from scipy.cluster import hierarchy
from scipy.optimize import curve_fit
from scipy.stats import linregress, norm, lognorm, halfnorm, gamma, kde, ttest_ind
from scipy.stats.mstats import gmean
from matplotlib import pyplot as plt
from matplotlib import colors
from mpl_toolkits.axes_grid1 import make_axes_locatable
try:
    import infomap
except:
    import sys
    sys.path.append('/usr/local/infomap_0.17.1/examples/python/infomap/')
    try:
        import infomap
    except:
        print('Failed to import infomap module, check path and ensure compatibility!')

'''
Assorted functions related to analysis of mutual information calculated for and/or dihedral
angles and coordinates using CALCULATE_ERMI.PY.
'''

def matrix_preprocess(mat_list, roll=0, idx=(0,0), run_name='run', out_dir=None):
    '''
    Crop, roll, and possibly stack and save some matrices.
    Input:
        mat_list: A matrix or list of square np.ndarrays with the same dimensions.
        roll: Number of rows/columns to roll; if positive, roll forward, if negative, roll
        backwards.
        idx: A tuple of integers. First and last indices for cropping matrices in mat_list. If
        crop is (0, 0), matrices will not be cropped.
        run_name: A string specifying a descriptive title/
        out_dir: If not None and a string specifying a valid directory, the new matrix will be
        pickled there.
    Output:
        R: A three dimensional np.ndarray with each processed matrix from mat_list indexed along
        the third dimension.
    '''
    if type(mat_list) is not list:
        mat_list = [mat_list]
    for i,m in enumerate(mat_list):
        print('Loading matrices from {}'.format(m))
        if type(m) is tuple and os.path.isfile(m[0]): # If loading the data from a pickled
            dictionary, provide a tuple for each matrix with structure ('path/to/file','dictionary_key')
            with open(m,'rb') as f:
                thisDict = pickle.load(f)
                this_mat_raw = thisDict[m[1]]
            elif isinstance(m,np.ndarray):
                print('Proceeding with ndarray...')
            else:
                raise ValueError('Supplied data must be path(s) to pickled ndarray(s) or
ndarray(s).')
        if (type(idx[0]) is int and type(idx[1]) is int) and sum(idx) != 0:
            this_mat = this_mat_raw[idx[0]:,idx[1]:]
        else:
            this_mat = this_mat_raw.copy()
        if type(roll) is int and roll != 0:
            this_mat = np.roll(np.roll(this_mat,roll,axis=0),roll,axis=1)
        if i == 0 and len(mat_list) > 1:

```

```

        R = np.zeros((this_mat.shape[0], this_mat.shape[1], len(mat_list)))
    elif i == 0 and len(mat_list) == 1:
        R = np.zeros((this_mat.shape[0], this_mat.shape[1]))
    if len(mat_list) > 1:
        R[:, :, i] = this_mat
    else:
        R[:, :] = this_mat
    if out_dir is not None:
        pickle.dump({'R':R}, open(os.path.join(out_dir, '{}_processed.pkl'.format(prefix)), 'wb'))
    return(R)

def rescale_matrix_diag(M, mode='multiplicative', geo_mean=False, runName='run', plot=False,
outDir=None, verbose=False):
    """
    This is the core rescaling protocol from MI_to_D, with multiplicative scaling incorporated as
    well. In the future, scale matrices with this, then calculate distance matrices if needed with
    MI_to_D.
    Input:
        M: A square np.ndarray with magnitudes decreasing exponentially from the diagonal.
        geo_mean: If True, use geometric mean to calculate diagonal means.
        runName: A string specifying a descriptive title.
        plot: If True, make plots of M before rescaling, of the diagonal means with fit function,
        and M after rescaling.
        outDir: If not None and a string to a valid directory, plots will be saved here.
        verbose: If True, say more.
    Output:
        M_scale: A square np.ndarray like M, but with diagonal elements rescaled.
    """
    print('\nPerforming diagonal scaling for {}'.format(runName))
    # If there are negative values and geometric mean is requested, set them to zero.
    if geo_mean and len(M[M<0]) > 0:
        print('\tInput matrix has negative values:\n{}\nThis is incompatible with calculation of
geometric mean, so will interpolate for diagonals with negative values.'.format(M[M<0]))
        # Calculate diagonal means/stds
        diag_mean_list = np.zeros(len(M))
        diag_perc_list = np.zeros((len(M),2))
        diag_std_list = np.zeros(len(M))
        for i in range(len(M)):
            if geo_mean:
                this_gmean = gmean(np.diag(M,i))
                if np.isnan(this_gmean):
                    if i == 0:
                        diag_mean_list[i] = gmean(np.diag(M,i+1))
                    elif i == len(M):
                        diag_mean_list[i] = gmean(np.diag(M,i-1))
                    else:
                        diag_mean_list[i] = np.mean([diag_mean_list[i-1], gmean(np.diag(M,i+1))])
                if diag_mean_list[i] == np.nan:
                    raise ValueError('\tConsecutive diagonals with negative elements; cannot
interpolate!')
            else:
                diag_mean_list[i] = this_gmean
        else:
            diag_mean_list[i]=np.mean(np.diag(M,i))
            diag_perc_list[i,:] = (np.percentile(np.diag(M,i),10), np.percentile(np.diag(M,i),90))
            diag_std_list[i]=np.std(np.diag(M,i))

        # Fit to get matrix properties
        tailCrop = int(round(0.1*len(M))) # At the end of the matrix, there will be very few elements
in diagonal so mean will be skewed; omit last 10%
        tailCropIdx = int(len(M)-tailCrop)
        xcrop = np.arange(2,tailCropIdx)
        xfull = np.arange(len(M))
        try:
            popt_M,pcov_M = curve_fit(exp_func,xcrop,diag_mean_list[xcrop],p0=[1,-1,0.5])
            #popt_M,pcov_M = curve_fit(exp_func,xfull,diag_mean_list)
            perr_M = np.sqrt(np.diag(pcov_M))
            fity_M = [exp_func(xi,*popt_M) for xi in xfull]
            print('\tUsed scipy.optimize.curve_fit to fit function\n          y=ae^(bx)+c \n          to MI data

```

```

with (value, sigma):\n      a = {}\n      b = {}\n      c =
{}'.format(*list(zip(popt_M.astype(float),perr_M.astype(float))))
    fit_success = True
except RuntimeError:
    print('\tWarning: failed to fit exponential decay function for {}, using simple mean
instead.'.format(runName))
    popt_M = [np.mean(diag_mean_list[int(0.2*len(M)):int(0.8*len(M))])]
    fit_success = False

# Scale the matrix
M_scale = np.zeros_like(M)
done=False
if mode == 'additive':
    neg_min_diag = []
for xi in xfull:
    theseIndices = kth_diag_indices(M_scale,xi)
    if verbose:
        print(diag_mean_list[xi], popt_M[-1])
    if mode == 'additive':
        if not done and diag_mean_list[xi] > popt_M[-1]:
            M_scale[theseIndices] = M[theseIndices]-np.mean(M[theseIndices])+popt_M[-1]
            if any(M_scale[theseIndices]<0):
                neg_min_diag.append(xi)
                M_scale[theseIndices] += np.abs(np.min(M_scale[theseIndices]))
        elif xi != 0: # Sometimes diagonal has been set to zero before-hand, this is not a
sign of leaving diagonal region
            done = True
            if done:
                M_scale[theseIndices]=M[theseIndices]
                #M_scale[theseIndices]=popt_M[-1]
    elif mode == 'multiplicative':
        if not done and diag_mean_list[xi] > popt_M[-1] + 0.05 * popt_M[-1]:
            M_scale[theseIndices] = M[theseIndices] * popt_M[-1] / diag_mean_list[xi]
        elif xi != 0:
            done = True
            if done:
                M_scale[theseIndices] = M[theseIndices]
M_scale+=np.triu(M_scale.T,1)

if mode == 'additive' and len(neg_min_diag) > 0:
    print('\tWarning: after rescaling, corrected negative values in the following diagonals:
{}'.format(neg_min_diag))

if plot:
    # Log scale
    # diag_mean_list_log = np.log(diag_mean_list)
    # diag_perc_list_log = np.log(diag_perc_list)
    # diag_std_list_log = np.log(diag_std_list)
    # fit_param_c_log = np.log(popt_M[2])
    n_points = len(diag_mean_list)

    # Create figure with three panels
    fig,ax=plt.subplots(1,3,figsize=(35,10))
    fig.suptitle('{}: {}, rescaled'.format(runName, mode))

    #
ax[0].errorbar(range(n_points),diag_mean_list_log,yerr=diag_std_list_log,fmt='o',color='k',ecolor
=None)
    # #ax[0].errorbar(range(n_points),diag_mean_list_log,yerr=[diag_perc_list_log[:,0],
diag_perc_list_log[:,1]])
    # ax[0].plot([0, n_points+1], [fit_param_c_log, fit_param_c_log], "r--")
ax[0].plot(xfull,diag_mean_list,'ko-', linewidth=2.0)
    if fit_success:
        ax[0].plot(xfull,fity_M, 'r-', linewidth=2.0,alpha=0.5)
    else:
        ax[0].plot([0,n_points],[popt_M[-1], popt_M[-1]])
ax[0].set_yscale('log')
ax[0].set_ylim([diag_mean_list.min()-
diag_mean_list.min()*0.1,diag_mean_list.max()+diag_mean_list.max()*0.1])

```

```

ax[0].set_xlim([-1,n_points+1])
ax[0].set_xlabel('k')
ax[0].set_ylabel('mean(M[diag(M,k)])')
ax[0].set_title('Input matrix, diagonal profile')
ax[0].grid(True)
vmin = np.percentile(M_scale,1)
vmax = np.percentile(M_scale,99)
a=ax[1].matshow(M,cmap=plt.cm.afmhot,vmin=vmin,vmax=vmax)
ax[1].set_xlabel('residue i')
ax[1].set_ylabel('residue j')
ax[1].set_title('Input matrix')
divA = make_axes_locatable(ax[1])
caxA = divA.append_axes('right',size='5%',pad=0.05)
cbarA = plt.colorbar(a,cax=caxA)
b=ax[2].matshow(M_scale,cmap=plt.cm.afmhot,vmin=vmin,vmax=vmax)
ax[2].set_xlabel('residue i')
ax[2].set_ylabel('residue j')
ax[2].set_title('Scaled matrix ({}).format(mode))
divB = make_axes_locatable(ax[2])
caxB = divB.append_axes('right',size='5%',pad=0.05)
cbarB = plt.colorbar(b,cax=caxB)
# Save the figure
if outDir is not None:
    plt.savefig(os.path.join(outDir,'{}_matrix-
rescale_{}.eps'.format(runName,mode)),transparent=True,format='eps')
else:
    plt.show()
return(M_scale)

def matrix_correlation_coef(miDict, compare_dict = None, n_permutations=100, msa_id=None,
seq_id_dict = {}, runName='run', plot=False, outDir=None):
    """
    Compares matrices in miDict (not limited to MI matrices) in many ways. Pretty gross code.
    mat_ids = (key for individual matrix, key for 3D consensus matrix with gaps removed, key for
    2D consensus matrix with gaps removed)
    """
    if compare_dict is None or compare_dict == {}:
        print('\nNo compare_dict specified, using defaults.')
        compare_dict =
{'dihedrals':('m','consensus_nogap_rescale','consensus_compressed_rescale','mi_mean_rescale'),
'coordinates':('m','consensus_nogap_rescale','consensus_compressed_rescale','mi_mean_rescale'),
'contacts':('m','consensus_nogap','consensus_compressed','m_mean'),
'distance':('m','consensus_nogap','consensus_compressed','m_mean')}
    corr_coef_dict = {}
    corr_mat = {}
    for dType, mat_ids in sorted(compare_dict.items()):
        if mat_ids[0] is not None:
            print('\nCalculating {} matrix correlation coefficients and cosines across replicates
for the same protein...'.format(dType))
            try:
                assert(corr_coef_dict['proteins'])
            except KeyError:
                corr_coef_dict.update({'proteins':{}})
            for prot_id,prot_dict in miDict['proteins'].items():
                set_ids = [x for x in list(prot_dict['sets'].keys()) if x != 'merge']
                if len(set_ids) == 0: # If at least one matrix look at shuffling
                    continue
                print('\tCalculating correlation coefficients and cosines between input and
shuffled {} matrices for each data set of protein {}...'.format(dType, prot_id))
                try:
                    assert(corr_coef_dict['proteins'][prot_id])
                except KeyError:
                    corr_coef_dict['proteins'].update({prot_id:{}})
                for this_set in set_ids:
                    try:
                        input_mat =
prot_dict['sets'][this_set]['data'][dType]['matrices'][mat_ids[0]]
                    except KeyError:
                        print('\t\t{} not found for {}, {}, {}... skipping.'.format(mat_ids[0],

```

```

prot_id, this_set, dType))
        continue
        rand_mat_ccs = np.zeros(n_permutations)*np.nan
        rand_mat_cosines = np.zeros(n_permutations)*np.nan
        for i, rand_triu in
((j, np.random.permutation(input_mat[np.triu_indices_from(input_mat)])) for j in
range(n_permutations)):
            rand_mat_ccs[i] =
triu_corr_coef(input_mat[np.triu_indices_from(input_mat)], rand_triu, diag=0)
            rand_mat_cosines[i] =
triu_cosine(input_mat[np.triu_indices_from(input_mat)], rand_triu, diag=0)
            cc = (np.mean(rand_mat_ccs), np.std(rand_mat_ccs))
            cosine = (np.mean(rand_mat_cosines), np.std(rand_mat_cosines))
            print('\t\tMatrix {}: {} vs {} random permutations: {}: CC = {:.4f} +/-
{:.4f}, cosine = {:.4f} +/- {:.4f}'.format(mat_ids[0], this_set, n_permutations, dType, cc[0],
cc[1], cosine[0], cosine[1]))
            try:
                assert(corr_coef_dict['proteins'][prot_id][dType])
            except:
                corr_coef_dict['proteins'][prot_id][dType] = {}

corr_coef_dict['proteins'][prot_id][dType].update({'{}_random'.format(this_set): {'cc_mean': cc[0],
'cc_std': cc[1], 'cosine_mean': cosine[0], 'cosine_std': cosine[1]}})
        if len(set_ids) < 2: # If at least two data sets, calculate correlation
coefficients between them
            continue
            print('\tCalculating correlation coefficients and cosines between {} data sets
for protein {}...'.format(dType, prot_id))
            for set_pair in sorted(it.combinations(set_ids, 2)):
                if dType in prot_dict['sets'][set_pair[0]]['data'].keys() and dType in
prot_dict['sets'][set_pair[1]]['data'].keys(): # For shared data types...
                    try:
                        mats =
[prot_dict['sets'][set_pair[i]]['data'][dType]['matrices'][mat_ids[0]] for i in (0,1)]
                    except KeyError:
                        print('\t\t{} vs. {} for matrix {}, data type {} failed,
skipping...'.format(set_pair[0], set_pair[1], mat_ids[0], dType))
                        continue
                    cosine = triu_cosine(*mats)
                    corr_coef = triu_corr_coef(*mats)
                    print('\t\t{} vs. {}: {}: CC = {:.4f}, cosine =
{:.4f}'.format(set_pair[0], set_pair[1], dType, corr_coef, cosine))
                    # try:
                    #     assert(corr_coef_dict['proteins'][prot_id][dType])
                    # except KeyError:
                    #     corr_coef_dict['proteins'][prot_id][dType] = {}

corr_coef_dict['proteins'][prot_id][dType].update({'{}_vs_{}'.format(*set_pair): {'cc': corr_coef, '
cosine': cosine}})

if mat_ids[1] is not None or mat_ids[2] is not None:
    if msa_id == None:
        msa_id = sorted(miDict['consensus']['data'].keys())[0]
    else:
        try:
            assert(miDict['consensus']['data'][msa_id])
        except KeyError:
            print('Supplied msa_id {} not found in input dictionary; this is required for
comparison of different proteins!'.format(msa_id))
            raise

if mat_ids[2] is not None:
    print('\nCalculating correlation coefficients and cosines between input consensus
matrix and shuffled consensus matrices...'.format(prot_id))
    try:
        assert(corr_coef_dict['consensus'][msa_id])
    except KeyError:
        corr_coef_dict['consensus'] = {msa_id: {}}
        corr_coef_dict['consensus'][msa_id].update({dType: {}})

```

```

    try:
        input_mat =
miDict['consensus']['data'][msa_id][dtype]['{}_{}_full'.format(mat_ids[0],msa_id)][mat_ids[2]]
    except KeyError:
        print('\tFailed to find {} matrix {} for MSA {}; skipping...'.format(dtype,
mat_ids[2],msa_id, dtype))
    else:
        rand_mat_ccs = np.zeros(n_permutations)*np.nan
        rand_mat_cosines = np.zeros(n_permutations)*np.nan
        for i,rand_triu in
((j,np.random.permutation(input_mat[np.triu_indices_from(input_mat)])) for j in
range(n_permutations)):
            rand_mat_ccs[i] =
triu_corr_coef(input_mat[np.triu_indices_from(input_mat)],rand_triu,diag=0)
            rand_mat_cosines[i] =
triu_cosine(input_mat[np.triu_indices_from(input_mat)],rand_triu,diag=0)
            cc = (np.mean(rand_mat_ccs), np.std(rand_mat_ccs))
            cosine = (np.mean(rand_mat_cosines), np.std(rand_mat_cosines))
            print('\tConsensus {} matrix from MSA {} vs {} random permutations: CC = {:.4f}
+/- {:.4f}, cosine = {:.4f} +/- {:.4f}'.format(dtype, msa_id, n_permutations, cc[0], cc[1],
cosine[0], cosine[1]))

corr_coef_dict['consensus'][msa_id][dtype].update({'consensus_random':{'cc_mean':cc[0],'cc_std':c
c[1],'cosine_mean':cosine[0],'cosine_std':cosine[1]})
# Compare homologs and look at relationship between seq. id. and MI
if mat_ids[1] is not None:
    print('\nCalculating correlation coefficients between average matrices of
homologues...')
    try:
        assert(corr_coef_dict['protein_pairs'])
    except KeyError:
        corr_coef_dict.update({'protein_pairs':{}})
        seq_id_compare = []
        if dtype in miDict['consensus']['data'][msa_id].keys():
            try:
                dat_nogap =
miDict['consensus']['data'][msa_id][dtype]['{}_{}_full'.format(mat_ids[0],msa_id)][mat_ids[1]]
            except KeyError:
                print('\tFailed to find consensus {} matrix {} for MSA {};
skipping...'.format(dtype,mat_ids[2],msa_id))
                continue
            order =
miDict['consensus']['data'][msa_id][dtype]['lookup_{}_consensus'.format(msa_id)]['full_order']
            corr_coef_dict['protein_pairs'][dtype] = {'corr_coef':{},'cosine':{}}
            dims = len(order)
            corr_mat[dtype] =
{'C':np.zeros((dims,dims)), 'O':np.zeros((dims,dims)), 'order':[]}
            if seq_id_dict == {}:
                try:
                    seq_id_dict = miDict['consensus']['seq_id']
                except:
                    print('\tFailed to load seq_id_dict from miDict, moving on...')
            if type(seq_id_dict) is dict and len(seq_id_dict.keys()) > 0:
                seq_id_compare.append((dtype,True))
                corr_coef_dict['protein_pairs'][dtype].update({'seq_id':{}})
                corr_mat[dtype].update({'S':np.zeros((dims,dims))})
                mess = ' and creating %ID matrix'
            else:
                seq_id_compare.append((dtype,False))
                #corr_mat[dtype]['C'] = np.zeros((dims,dims))
            print('\tCalculating correlation coefficients and cosines for {} {}
matrix{}...'.format(mat_ids[1], dtype, mess))
            for prot_pair in it.combinations(order,2):
                prot_a_idx = prot_pair[0][0]
                prot_b_idx = prot_pair[1][0]
                if dtype in ('coordinates', 'dihedrals'):
                    prot_a = '_'.join(prot_pair[0][1].split('_')[:-2])
                    prot_b = '_'.join(prot_pair[1][1].split('_')[:-2])
                else:

```

```

        prot_a = '_' .join(prot_pair[0][1].split('_')[:-1])
        prot_b = '_' .join(prot_pair[1][1].split('_')[:-1])
        if prot_a not in corr_mat[dType]['order']:
            corr_mat[dType]['order'].append(prot_a)
        mat_a = dat_nogap[:, :, prot_a_idx].copy()
        mat_b = dat_nogap[:, :, prot_b_idx].copy()
        corr_coef = triu_corr_coef(mat_a, mat_b)
        cosine = triu_cosine(mat_a, mat_b)
        print('\t\t{} vs. {}: {}: CC = {:.4f}, cosine = {:.4f}'.format(prot_a,
prot_b, dType, corr_coef, cosine), end='')
        corr_mat[dType]['C'][prot_a_idx, prot_b_idx] = corr_coef
        corr_mat[dType]['O'][prot_a_idx, prot_b_idx] = cosine

corr_coef_dict['protein_pairs'][dType]['corr_coef']['{}_vs_{}'.format(prot_a, prot_b)] =
{'cc': corr_coef, 'cosine': cosine}
        if seq_id_compare[-1][1]:
            try:

corr_coef_dict['protein_pairs'][dType]['seq_id']['{}_vs_{}'.format(prot_a, prot_b)] =
seq_id_dict[prot_a][prot_b]
                corr_mat[dType]['S'][prot_a_idx, prot_b_idx] =
seq_id_dict[prot_a][prot_b]
            except KeyError:
                try:

corr_coef_dict['protein_pairs'][dType]['seq_id']['{}_vs_{}'.format(prot_a, prot_b)] =
seq_id_dict[prot_b][prot_a]
                corr_mat[dType]['S'][prot_a_idx, prot_b_idx] =
seq_id_dict[prot_b][prot_a]
            except KeyError:
                print('\n\t\tCouldn\t find % identity for {} and {},
skipping!'.format(prot_a, prot_b))
                raise
                continue
                print(', seq. ID:
{:.4f}'.format(corr_mat[dType]['S'][prot_a_idx, prot_b_idx]))
            else:
                print('')
                for k in ('C', 'O'): # C is corr. coeff., O is cosine, S is seq. ID.
                    corr_mat[dType][k] += corr_mat[dType][k].T
                    corr_mat[dType][k][np.diag_indices(dims)] = 1.0
                if seq_id_compare[-1][1]:
                    corr_mat[dType]['S'] += corr_mat[dType]['S'].T
                    corr_mat[dType]['S'][np.diag_indices(dims)] = 1.0
                    m_cc, x_cc, r_cc, p_cc, se_cc =
linregress(corr_mat[dType]['S'][np.triu_indices_from(corr_mat[dType]['S'], 1)], corr_mat[dType]['C']
)[np.triu_indices_from(corr_mat[dType]['C'], 1)])
                    corr_mat[dType]['cc-vs-seq_linregress'] =
{'m': m_cc, 'x': x_cc, 'r': r_cc, 'p': p_cc, 'se': se_cc}
                    m_cos, x_cos, r_cos, p_cos, se_cos =
linregress(corr_mat[dType]['S'][np.triu_indices_from(corr_mat[dType]['S'], 1)], corr_mat[dType]['O']
)[np.triu_indices_from(corr_mat[dType]['O'], 1)])
                    corr_mat[dType]['cos-vs-seq_linregress'] =
{'m': m_cos, 'x': x_cos, 'r': r_cos, 'p': p_cos, 'se': se_cos}

        # Compare MI and contacts
        mi_types = [t for t in compare_dict.keys() if t in ('coordinates', 'dihedrals')]
        if 'contacts' in compare_dict.keys() and len(mi_types) > 0 and compare_dict['contacts'][3] is
not None:
            for dType in mi_types:
                if compare_dict[dType][3] is None:
                    continue
                print('\nDetermining relationship between {} MI and contact graph by shuffling ({}
permutations)...'.format(dType[:-1], n_permutations))
                try:
                    assert(corr_coef_dict['proteins'])
                except KeyError:
                    corr_coef_dict.update({'proteins': {}})
                    keys = ('mi_full_all_rand', 'mi_full_c_rand',

```

```

'mi_full_nonc_rand', 'mi_full_numc_rand', 'mi_full_numnonc_rand')
dtype_cos_dat = np.zeros((len(miDict['proteins'].keys()), len(keys)*2))*np.nan
dtype_ccs_dat = np.zeros((len(miDict['proteins'].keys()), len(keys)*2))*np.nan
for prot_idx, prot_id in enumerate(sorted(miDict['proteins'].keys())):
    try:
        assert(corr_coef_dict['proteins'][prot_id])
    except:
        corr_coef_dict['proteins'].update({prot_id: {}})
    try:
        assert(corr_coef_dict['proteins'][prot_id][dtype])
    except:
        corr_coef_dict['proteins'][prot_id].update({dtype: {}})
    mi_full =
miDict['proteins'][prot_id]['sets']['merge']['data'][dtype]['matrices'][compare_dict[dtype][3]][n
p.triu_indices_from(miDict['proteins'][prot_id]['sets']['merge']['data'][dtype]['matrices'][compa
re_dict[dtype][3]))]
    contacts =
miDict['proteins'][prot_id]['sets']['merge']['data']['contacts']['matrices'][compare_dict['contac
ts'][3]][np.triu_indices_from(miDict['proteins'][prot_id]['sets']['merge']['data']['contacts']['m
atrices'][compare_dict['contacts'][3]))]
    if len(mi_full) != len(contacts):
        print('Warning: lengths of upper triangles from MI matrix and contacts matrix
are not equal for {}, skipping!'.format(prot_id))
        continue
    c_mask = contacts > 0
    nonc_mask = contacts == 0
    mi_sub_c = mi_full[c_mask]
    mi_sub_nonc = mi_full[nonc_mask]
    ccs = np.zeros((n_permutations, len(keys)))*np.nan # correlation coefficients over
n_permutations: c_rand, nonc_rand, numc_rand, numnonc_rand, full_rand
    cosines = np.zeros((n_permutations, len(keys)))*np.nan # cosines over
n_permutations: c_rand, nonc_rand, numc_rand, numnonc_rand, full_rand
    for this_perm in range(n_permutations): # Repeat for many permutations
        mi_full_all_rand = np.random.permutation(mi_full)
        mi_sub_c_rand = np.random.permutation(mi_sub_c)
        mi_sub_nonc_rand = np.random.permutation(mi_sub_nonc)
        c_mask_rand = np.random.permutation(c_mask)
        nonc_mask_rand = np.random.permutation(nonc_mask)
        mi_full_c_rand = np.zeros_like(mi_full)*np.nan
        mi_full_nonc_rand = np.zeros_like(mi_full)*np.nan
        mi_full_numc_rand = np.zeros_like(mi_full)*np.nan
        mi_full_numnonc_rand = np.zeros_like(mi_full)*np.nan
        mi_sub_numc_rand = mi_full[c_mask_rand]
        mi_sub_numnonc_rand = mi_full[nonc_mask_rand]
        n_c = len(mi_sub_numc_rand)
        n_nonc = len(mi_sub_numnonc_rand)
        j = k = 0
        for i in range(len(mi_full)):
            if c_mask[i] == True: # If it's a contact position, take a shuffled value
in *_c_* and a non-shuffled value in *_nonc_*
                mi_full_nonc_rand[i] = mi_full[i]
                mi_full_numnonc_rand[i] = mi_full[i]
                mi_full_c_rand[i] = mi_sub_c_rand[j]
                mi_full_numc_rand[i] = mi_sub_numc_rand[j]
                j += 1
            elif c_mask[i] == False: # If it's not a contact position, take a non-
shuffled value in *_c_* and a shuffled value in *_nonc_*
                mi_full_c_rand[i] = mi_full[i]
                mi_full_numc_rand[i] = mi_full[i]
                mi_full_nonc_rand[i] = mi_sub_nonc_rand[k]
                mi_full_numnonc_rand[i] = mi_sub_numnonc_rand[k]
                k += 1
            ccs[this_perm, :] = [np.corrcoef(mi_full_all_rand, mi_full)[0,1],
np.corrcoef(mi_full_c_rand, mi_full)[0,1], np.corrcoef(mi_full_nonc_rand, mi_full)[0,1],
np.corrcoef(mi_full_numc_rand, mi_full)[0,1], np.corrcoef(mi_full_numnonc_rand, mi_full)[0,1]]
            cosines[this_perm, :] = [triu_cosine(mi_full_all_rand, mi_full),
triu_cosine(mi_full_c_rand, mi_full), triu_cosine(mi_full_nonc_rand, mi_full),
triu_cosine(mi_full_numc_rand, mi_full), triu_cosine(mi_full_numnonc_rand, mi_full)]
            ccs_dat = [item for sublist in list(zip(np.mean(ccs, 0), np.std(ccs, 0)))] for item

```

```

in sublist]
    cos_dat = [item for sublist in list(zip(np.mean(cosines,0),np.std(cosines,0)))
for item in sublist]
    print('\t\t{} {} ({} contacts, {} non-contacts):'.format(prot_id, dType, n_c,
n_nonc))
    print('\t\t\tCorrelation coefficient:\n\t\t\t\tFully shuffled: {:.4f} +/-
{:.4f}\n\t\t\t\tContacts shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tNon-contacts shuffled: {:.4f}
+/- {:.4f}\n\t\t\t\tRandom positions shuffled (# contacts): {:.4f} +/- {:.4f}\n\t\t\t\tRandom
positions shuffled (# non-contacts): {:.4f} +/- {:.4f}'.format(*ccs_dat))
    print('\t\t\tCosines:\n\t\t\t\tFully shuffled: {:.4f} +/-
{:.4f}\n\t\t\t\tContacts shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tNon-contacts shuffled: {:.4f}
+/- {:.4f}\n\t\t\t\tRandom positions shuffled (# contacts): {:.4f} +/- {:.4f}\n\t\t\t\tRandom
positions shuffled (# non-contacts): {:.4f} +/- {:.4f}'.format(*cos_dat))
    corr_coef_dict['proteins'][prot_id][dType]['contacts_vs_mi'] =
{'cc':{},'cosines':{}}
    for i,k in enumerate(keys):
        if i%2 == 0:

corr_coef_dict['proteins'][prot_id][dType]['contacts_vs_mi']['cc'].update({k+'_mean':ccs_dat[i]})
corr_coef_dict['proteins'][prot_id][dType]['contacts_vs_mi']['cosines'].update({k+'_mean':cos_dat
[i]})
        else:

corr_coef_dict['proteins'][prot_id][dType]['contacts_vs_mi']['cc'].update({k+'_std':ccs_dat[i]})
corr_coef_dict['proteins'][prot_id][dType]['contacts_vs_mi']['cosines'].update({k+'_std':cos_dat[
i]})
        dtype_cos_dat[prot_idx,:] = cos_dat
        dtype_ccs_dat[prot_idx,:] = ccs_dat
        mean_dtype_cos_dat = np.mean(dtype_cos_dat,axis=0)
        mean_dtype_ccs_dat = np.mean(dtype_ccs_dat,axis=0)
        print('\t\tAverage over all proteins for {}'.format(dType))
        print('\t\t\tCorrelation coefficient:\n\t\t\t\tFully shuffled: {:.4f} +/-
{:.4f}\n\t\t\t\tContacts shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tNon-contacts shuffled: {:.4f}
+/- {:.4f}\n\t\t\t\tRandom positions shuffled (# contacts): {:.4f} +/- {:.4f}\n\t\t\t\tRandom
positions shuffled (# non-contacts): {:.4f} +/- {:.4f}'.format(*mean_dtype_ccs_dat))
        print('\t\t\tCosines:\n\t\t\t\tFully shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tContacts
shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tNon-contacts shuffled: {:.4f} +/- {:.4f}\n\t\t\t\tRandom
positions shuffled (# contacts): {:.4f} +/- {:.4f}\n\t\t\t\tRandom positions shuffled (# non-
contacts): {:.4f} +/- {:.4f}'.format(*mean_dtype_cos_dat))

        if 'distance' not in compare_dict.keys() or 'contacts' not in compare_dict.keys():
            continue
        print('\nComparing {} MI eigenreconstructions with contact graph...'.format(dType[:-
1], n_permutations))
        for prot_idx,prot_id in enumerate(sorted(miDict['proteins'].keys()+['consensus'])):
            if prot_id == 'consensus':
                try:
                    assert(corr_coef_dict['consensus'])
                except:
                    corr_coef_dict.update({'consensus':{}})
                try:
                    assert(corr_coef_dict['consensus'][dType])
                except:
                    corr_coef_dict['consensus'].update({dType:{}})
                mi_full =
miDict['consensus']['data'][msa_id][dType]['m_{}_full'.format(msa_id)]['consensus_compressed']
                contacts =
miDict['consensus']['data'][msa_id]['contacts']['m_pdz_full']['consensus_compressed']
                distance =
miDict['consensus']['data'][msa_id]['distance']['m_pdz_full']['consensus_compressed']
            else:
                try:
                    assert(corr_coef_dict['proteins'][prot_id])
                except:
                    corr_coef_dict['proteins'].update({prot_id:{}})
                try:
                    assert(corr_coef_dict['proteins'][prot_id][dType])

```

```

        except:
            corr_coef_dict['proteins'][prot_id].update({dtype: {}})
            mi_full =
miDict['proteins'][prot_id]['sets']['merge']['data'][dtype]['matrices'][compare_dict[dtype][3]]
            contacts =
miDict['proteins'][prot_id]['sets']['merge']['data']['contacts']['matrices'][compare_dict['contac
ts'][3]]
            distance =
miDict['proteins'][prot_id]['sets']['merge']['data']['contacts']['matrices'][compare_dict['distan
ce'][3]]
            distance_inverted = (distance.max()-distance)/(distance.max()-distance.min())

        if len(mi_full) != len(contacts):
            print('Warning: lengths of upper triangle from MI matrix and contacts matrix
are not equal for {}, skipping!'.format(prot_id))
            continue

        # First, decompose the MI
        mi_msub = mi_full - mi_full.mean()
        mi_evals, mi_evecs = np.linalg.eigh(mi_msub)
        mi_evals_abs = np.abs(mi_evals)
        mi_evals_idx = mi_evals_abs.argsort()[::-1] # Sort from greatest to least
        mi_evals_abs_sort = mi_evals_abs[mi_evals_idx]
        mi_evals_sort = mi_evals[mi_evals_idx]
        mi_evecs_sort = mi_evecs[:, mi_evals_idx]
        nEle = mi_msub.shape[0]
        mi_recon_top = np.zeros([nEle, nEle, nEle])
        mi_recon_bottom = np.zeros([nEle, nEle, nEle])
        for i in range(nEle):
            top_evecs_temp = np.zeros_like(mi_evecs_sort)
            top_evecs_temp[:, i+1:] = mi_evecs_sort[:, i+1:]
            top_evals_temp = np.zeros_like(mi_evals_sort)
            top_evals_temp[i+1:] = mi_evals_sort[i+1:]
            top_evals_temp = np.diag(top_evals_temp)
            mi_recon_top[:, :, i] =
np.dot(top_evecs_temp, np.dot(top_evals_temp, top_evecs_temp.T))
            bottom_evecs_temp = np.zeros_like(mi_evecs_sort)
            bottom_evecs_temp[:, i+1:] = mi_evecs_sort[:, i+1:]
            bottom_evals_temp = np.zeros_like(mi_evals_sort)
            bottom_evals_temp[i+1:] = mi_evals_sort[i+1:]
            bottom_evals_temp = np.diag(bottom_evals_temp)
            mi_recon_bottom[:, :, i] =
np.dot(bottom_evecs_temp, np.dot(bottom_evals_temp, bottom_evecs_temp.T))

        # Calculate difference of norms over reconstructions.
        delta_norm = np.zeros((nEle, 11)) * np.nan # ((top_i, bottom_i), (top_i+1,
bottom_i+1), ...)
        for i in range(nEle):
            delta_norm[i, 0] = np.linalg.norm(mi_recon_top[:, :, i], ord='fro') -
np.linalg.norm(distance_inverted, ord='fro')
            delta_norm[i, 2] = np.linalg.norm(mi_recon_bottom[:, :, i], ord='fro') -
np.linalg.norm(distance_inverted, ord='fro')
            delta_norm[i, 5] = np.linalg.norm(mi_recon_top[:, :, i], ord='fro') -
np.linalg.norm(contacts, ord='fro')
            delta_norm[i, 7] = np.linalg.norm(mi_recon_bottom[:, :, i], ord='fro') -
np.linalg.norm(contacts, ord='fro')

            delta_norm[:, 1] = simple_rescale(delta_norm[:, 0])
            delta_norm[:, 3] = simple_rescale(delta_norm[:, 2])
            delta_norm[:, 4] = delta_norm[:, 3] + delta_norm[:, 1]
            #delta_norm[:, 4] = delta_norm[:, 2] + delta_norm[:, 0]
            delta_norm[:, 6] = simple_rescale(delta_norm[:, 5])
            delta_norm[:, 8] = simple_rescale(delta_norm[:, 7])
            delta_norm[:, 9] = delta_norm[:, 8] + delta_norm[:, 6]
            #delta_norm[:, 9] = delta_norm[:, 7] + delta_norm[:, 5]
            delta_norm[:, 10] = simple_rescale(delta_norm[:, 9])
            optimal_cut_idx = delta_norm[:, 9].argmax()
            print('\tOptimal eigenmode for cut, {}, {}:'
{}').format(prot_id, dtype, optimal_cut_idx)

```

```

# dat = mi_recon_bottom[:, :, optimal_cut_idx] + mi_full.mean()
# dat_bin = dat > np.percentile(dat, 0.9)
# dist_bin = distance > np.percentile(distance, 0.9)
# print('\t\tCC = {}, cosine = {}'.format(triu_corr_coef(dat_bin, dist_bin),
triu_cosine(dat_bin, dist_bin))

if prot_id == 'consensus':
    corr_coef_dict['consensus'][msa_id][dtype]['eigenrecon'] = delta_norm
else:
    corr_coef_dict['proteins'][prot_id][dtype]['eigenrecon'] = delta_norm

if plot:
    figE, axE = plt.subplots(nrows=1, ncols=3, figsize=(20, 10), squeeze=False)
    bounds = (mi_full.min(), mi_full.max())
    axE[0][0].matshow(mi_full, cmap=plt.cm.afmhot, vmin = bounds[0], vmax =
bounds[1])

axE[0][1].matshow(mi_recon_top[:, :, optimal_cut_idx] + mi_full.mean(), cmap=plt.cm.afmhot, vmin =
bounds[0], vmax = bounds[1])

axE[0][2].matshow(mi_recon_bottom[:, :, optimal_cut_idx] + mi_full.mean(), cmap=plt.cm.afmhot, vmin =
bounds[0], vmax = bounds[1])
    [axE[0][idx].set_title(title) for idx, title in enumerate(['Input', 'Top {}
modes'.format(optimal_cut_idx), 'Bottom {} modes'.format(mi_full.shape[0] - optimal_cut_idx) ])]
    [axE[0][idx].set_xlabel('Residue i') for idx in range(3)]
    [axE[0][idx].set_ylabel('Residue j') for idx in range(3)]
    figE.suptitle('{} {} data reconstruction'.format(prot_id, dtype), fontsize=24)
    if outDir is not None:
        plt.savefig(os.path.join(outDir, '{}_{}_{}_opt-
recon.eps'.format(prot_id, dtype, runName)), transparent=True)
        plt.close()
    else:
        plt.show()

if plot:
    # Correlation coefficient matrix and clustering
    n_plots = len(corr_mat.keys())
    if any([s[1] for s in seq_id_compare]):
        ncols=3
    else:
        ncols=2
    figA, axA = plt.subplots(nrows=n_plots, ncols=ncols, figsize=(50, 10), squeeze=False)
    fDict = {}

fDict.update({'p': {}, 'pdiv': {}, 'pcax': {}, 'pbar': {}, 'q': {}, 'qdiv': {}, 'qcax': {}, 'qbar': {}})
for i, dtype in enumerate(corr_mat.keys()):
    try:
        order =
miDict['consensus']['data'][msa_id][dtype]['lookup_{}_consensus'.format(msa_id)]['full_order']
    except KeyError:
        print('Failed to plot for {}, skipping...'.format(dtype))
        continue
    labels = ['_'.join(o[1].split('_')[:-1]) for o in order]
    n_prot = len(labels)
    # Correlation matrix
    fDict['p'][i] =
axA[i][0].matshow(corr_mat[dtype]['C'], cmap=plt.cm.afmhot, vmin=0, vmax=1)
    axA[i][0].set_xticks([])
    axA[i][0].set_yticks(range(n_prot))
    axA[i][0].tick_params(axis='both', which='major', labelsize=8)
    j = [int(n) for n in axA[i][0].get_yticks()]
    y = [labels[n] for n in j]
    axA[i][0].set_yticklabels(y)
    axA[i][0].set_title('Correlation coefficients, {}'.format(dtype))
    axA[i][0].set_xlabel('Protein i')
    axA[i][0].set_ylabel('Protein j')
    fDict['pdiv'][i] = make_axes_locatable(axA[i][0])

```

```

fDict['pcax'][i] = fDict['pdiv'][i].append_axes('right',size='5%',pad=0.05)
fDict['pcbar'][i] = plt.colorbar(fDict['p'][i],cax=fDict['pcax'][i])
# Dendrogram
Y = hierarchy.linkage(1-corr_mat[dType]['C'],method='ward')
Z = hierarchy.dendrogram(Y,labels=labels,ax=axA[i][1],orientation='right')
axA[i][1].set_title('Dendrogram, {}'.format(dType))
axA[i][1].set_xlabel('Distance')
axA[i][1].set_ylabel('Protein')
# Dendrogram-sorted matrices
idx = Z['leaves'][:-1]
C_sort = corr_mat[dType]['C'].copy()
C_sort = C_sort[idx,:]
C_sort = C_sort[:,idx]
labels_sort = [labels[i] for i in idx]
fDict['q'][i] = axA[i][2].matshow(C_sort,cmap=plt.cm.afmhot,vmin=0,vmax=1)
axA[i][2].set_xticks([])
axA[i][2].set_yticks(range(n_prot))
axA[i][2].tick_params(axis='both', which='major', labels=8)
j = [int(n) for n in axA[i][2].get_yticks()]
y = [labels_sort[n] for n in j]
axA[i][2].set_yticklabels(y)
axA[i][2].set_title('Clustered correlation coefficients, {}'.format(dType))
axA[i][2].set_xlabel('Protein i')
axA[i][2].set_ylabel('Protein j')
fDict['qdiv'][i] = make_axes_locatable(axA[i][2])
fDict['qcax'][i] = fDict['qdiv'][i].append_axes('right',size='5%',pad=0.05)
fDict['qcbar'][i] = plt.colorbar(fDict['q'][i],cax=fDict['qcax'][i])
if outDir is not None:
    plt.savefig(os.path.join(outDir,'{}_corr-
coefs.eps'.format(runName)),transparent=True)
    plt.close()
else:
    plt.show()
# Scatter plots for seq ID vs correlation
if len(seq_id_compare) > 0:
    figB,axB=plt.subplots(nrows=1,ncols=n_plots,figsize=(50,10),squeeze=False)
    for i,dType in enumerate(corr_mat.keys()):
        try:
            s=corr_mat[dType]['S'][np.triu_indices_from(corr_mat[dType]['S'],1)]
            c=corr_mat[dType]['C'][np.triu_indices_from(corr_mat[dType]['C'],1)]
        except KeyError:
            print('Failed to find correlation matrix for data type {},
skipping...'.format(dType))
            continue
        dat = list(zip(s,c))
        # for j,p in enumerate(dat):
        #     if p[0] > 0.99:
        #         dat.pop(j)
        [dat.pop(j) for j,p in enumerate(dat) if p[0] > 0.99]
        axB[0][i].scatter([d[0] for d in dat],[d[1] for d in dat],c='k')
        axB[0][i].set_title('{} data vs. % Seq Identity\nr = {}'.format(dType,
np.corrcoef([d[0] for d in dat],[d[1] for d in dat])[0]))
        axB[0][i].set_ylabel('triu(MI,1) correlation coefficient')
        axB[0][i].set_xlabel('% Sequence Identity')
        axB[0][i].grid()
        if outDir is not None:
            plt.savefig(os.path.join(outDir,'{}_corr-coefs_vs_seq-
id.eps'.format(runName)),transparent=True)
            plt.close()
        else:
            plt.show()
# Recon. vs. contacts/distance
# if 'eigenrecon' in
corr_coef_dict['proteins'][corr_coef_dict['proteins'].keys()[0]][corr_coef_dict['proteins'][corr_
coef_dict['proteins'].keys()[0]].keys()[0]].keys():
    figC,axC=plt.subplots(nrows=1,ncols=2,figsize=(20,10),squeeze=False)
    max_xlim = 0
    for prot_id, prot_dict in corr_coef_dict['proteins'].items():
        for i,dType in enumerate(('coordinates','dihedrals')):

```

```

        axC[0][i].plot(range(prot_dict[dType]['eigenrecon'].shape[0]),
prot_dict[dType]['eigenrecon'][:,6], 'r', alpha=0.4)
        axC[0][i].plot(range(prot_dict[dType]['eigenrecon'].shape[0]),
prot_dict[dType]['eigenrecon'][:,8], 'b', alpha=0.4)
        axC[0][i].plot(range(prot_dict[dType]['eigenrecon'].shape[0]),
prot_dict[dType]['eigenrecon'][:,10], 'k', alpha=1)
        if prot_dict[dType]['eigenrecon'].shape[0] > max_xlim:
            max_xlim = prot_dict[dType]['eigenrecon'].shape[0]
        for i in range(2):
            axC[0][i].set_title(('Coordinates', 'Dihedrals')[i], fontsize=18)
            axC[0][i].set_xlabel('Number of top eigenmodes')
            axC[0][i].set_ylabel('\n+ rescale(fnorm(mi_recon_top)-fnorm(distance_inverted))
\n+ rescale(fnorm(mi_recon_bottom)-fnorm(distance_inverted))')
            axC[0][i].set_xlim((0, max_xlim))
        figC.suptitle('Eigenreconstruction similarity to distance matrices of all
proteins', fontsize=24)
        if outDir is not None:
            plt.savefig(os.path.join(outDir, '{}_eigenrecon-
all.eps'.format(runName)), transparent=True)
            plt.close()
        else:
            plt.show()
        figD, axD=plt.subplots(nrows=1, ncols=2, figsize=(20, 10), squeeze=False)
        for i, dType in enumerate(('coordinates', 'dihedrals')):
            axD[0][i].plot(range(corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'].shape[0]),
corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'][:,6], 'ro-', alpha=1)

            axD[0][i].plot(range(corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'].shape[0]),
corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'][:,8], 'bo-', alpha=1)

            axD[0][i].plot(range(corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'].shape[0]),
corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'][:,10], 'ko-', alpha=1)
            for i in range(2):
                axD[0][i].set_title(('Coordinates', 'Dihedrals')[i], fontsize=18)
                axD[0][i].set_xlabel('Number of top eigenmodes')
                axD[0][i].set_ylabel('\n+ rescale(fnorm(mi_recon_top)-fnorm(distance_inverted))
\n+ rescale(fnorm(mi_recon_bottom)-fnorm(distance_inverted))')
            axD[0][i].set_xlim((0, corr_coef_dict['consensus'][msa_id][dType]['eigenrecon'].shape[0]))
            figD.suptitle('Consensus eigenreconstruction similarity to consensus distance
matrices', fontsize=24)
            if outDir is not None:
                plt.savefig(os.path.join(outDir, '{}_eigenrecon-
consensus.eps'.format(runName)), transparent=True)
                plt.close()
            else:
                plt.show()
        return(corr_coef_dict, corr_mat)

def mat_vs_cm(M, C, suppress_diag=None, n_iter=10000, run_name='run', plot=(False, 'hist'),
out_dir=None):
    """
    """
    print('Binarizing contact matrix and counting contacts...')
    if isinstance(suppress_diag, int):
        print('Will suppress contacts for diagonals up to {}'.format(suppress_diag))
        suppress_diag = np.abs(suppress_diag)
        for i in range(suppress_diag):
            C[kth_diag_indices(C, i)] = 0
    C_bin = C.astype('bool')
    n_contacts = C_bin.sum()
    n_noncontacts = C.size - n_contacts
    print('Found {} contacts out of {} possible in input contact matrix.\nMasking input
matrix...'.format(n_contacts, C.size))
    mat_dict = {'M_mask':M[C_bin].copy(),
                'M_inv_mask':M[~C_bin].copy(),
                'M_rand_mask':np.ndarray((n_iter, n_contacts)),
                'M_rand_inv_mask':np.ndarray((n_iter, n_noncontacts))
    
```

```

    }
    print('Masking {} shuffled input matrices and performing T-tests...'.format(n_iter))
    pvals = np.zeros((n_iter,2))*np.nan
    for i in range(n_iter):
        M_rand = np.random.permutation(M.flat).reshape(M.shape)
        mat_dict['M_rand_mask'][i,:] = M_rand[C_bin].copy()
        mat_dict['M_rand_inv_mask'][i,:] = M_rand[~C_bin].copy()
        pvals[i,:] = ttest_ind(mat_dict['M_mask'],
mat_dict['M_rand_mask'][i,:],equal_var=False)[1], ttest_ind(mat_dict['M_inv_mask'],
mat_dict['M_rand_inv_mask'][i,:],equal_var=False)[1]
        pvals = pvals.mean(axis=0)
        stat_dict = {'mean_M':M.mean(), 'std_M':M.std(),
'contacts_pval':pvals[0], 'noncontacts_pval':pvals[1]}
        for name,matrix in mat_dict.items():
            if matrix.ndim == 1:
                stat_dict.update({'mean_'+name:matrix.mean(), 'std_'+name:matrix.std()})
            elif matrix.ndim == 2:
                stat_dict.update({'mean_'+name:matrix.mean(axis=1).mean(),
'std_'+name:matrix.std(axis=1).std()})
                stat_dict.update({'z_'+name:(stat_dict['mean_'+name] -
stat_dict['mean_M'])/stat_dict['std_M']})
        print('Statistics:')
        for name,stat in stat_dict.items():
            print('\t{:} {:.4f}'.format(name,stat))
        if isinstance(plot,bool):
            plot = (plot, 'hist')
        if plot[0]:
            print('Plotting...')
            figA,axA=plt.subplots(nrows=1,ncols=1,figsize=(15,10))
            x = np.arange(np.percentile(M,0.1), np.percentile(M,99.9), .01)
            if plot[1].lower() in ('kde','both'):
                kde_list = []
                colors = 'krgbcmy'
                for idx,item in enumerate(mat_dict.items()):
                    print('\t{}...'.format(item[0]))
                    kde_list.append(kde.gaussian_kde(item[1].flat,bw_method=0.1))
                    axA.plot(x,kde_list[-1](x), color=colors[idx], label=item[0])
                    # [axA.plot(x,dat[1](x),label=dat[0]) for dat in ((k,
kde.gaussian_kde(v.flat,bw_method=0.1)) for k,v in mat_dict.items())] # One-liner
                elif plot[1].lower() in ('hist','both'):
                    axA.hist(
                        [mat_dict['M_rand_mask'],mat_dict['M_rand_inv_mask'],
mat_dict['M_mask'],mat_dict['M_inv_mask']],
                        bins=20,
                        normed=True,
                        color=['k','k','r','b'],
                        edgecolor='w',
                        label=['Random contacts', 'Random non-contacts', 'Input contacts', 'Input
non-contacts'])
            )
            axA.set_xlim((np.percentile(M,0.1), np.percentile(M,99.9)))
            axA.set_xlabel('Matrix units')
            axA.set_ylabel('Probability density')
            axA.set_title(run_name)
            plt.legend()
            if isinstance(out_dir,str) and os.path.isdir(out_dir):
                plt.savefig(os.path.join(out_dir,'{}_mat-vs-
contacts.eps'.format(run_name)),transparent=True,format='eps')
                plt.close()
            else:
                plt.show()
        print('Done.')
        return(mat_dict, stat_dict)

def MI_to_D(MI,scale=True,normMode='half',runName='',plot=False,outDir=None):
    """
    * Description:

    Calculates a distance matrix D from a mutual information matrix M.

```

If scaling is requested, a distance matrix D is first calculated as $e^{(-M/\text{variance}(M))}$ where the variance in M is calculated relative to a half-normal distribution. The diagonal of D is set to 0.

Next, a scaled distance matrix D_s is calculated. First, the mean of each diagonal of D is calculated. An exponential function, $y = a * e^{(-b * x)} + c$, is fit to this series of means. Moving out from the 0-diagonal, the difference between that mean and the fit parameter c is added to all elements of the diagonal of D . Once the mean of a diagonal is greater than or equal to c , the procedure stops, and D_s is returned.

This corrects for overweighting of the distance matrix due to trivially high mutual information between consecutive residues.

* Input:

* `MI`

An $n_{\text{Resi}} \times n_{\text{Resi}}$ `numpy` array where each MI_{ij} is the mutual information calculated from dihedral angles at resi i and resi j .

* `scale`

A boolean; if `True`, the program will scale the distance matrix near the diagonal. The scaling procedure has the effect of down-weighting trivial MI which arises from residues in direct contact due to proximity in primary sequence.

* `normMode`

`{half, full}`

A string which specifies the probability distribution to be used for calculating the variance of the distribution of MI values. This defaults to "half", as `MIxyn` does not yield negative MI values and the distribution is one-tailed.

* `runName`

An optional name to prepend to output for the run.

* `plot`

If `True`, will use `matplotlib.pyplot` to visualize MI, D , and D_s .

* `outDir`

The path to an output directory. If plotting has been requested, figures will be saved there.

* Output:

* `D`

A raw distance matrix D .

* `D_scale`

A scaled distance matrix D_s ; output by default.

...

```
D = MI.copy()
np.fill_diagonal(D,0) # Set D equal to MI without diagonal
if normMode == 'half':
    mean,var=halfnorm.fit(D.flatten())
elif normMode == 'full':
    mean,var=norm.fit(D.flatten())
else:
    raise ValueError('Must use either halfnorm or norm; use \'half\' or \'full\')
```

```

respectively.')
print('\nFit {}-normal distribution to MI values with mean of {:.4f} bits and a variance of
{:.4f} bits.'.format(normMode,mean,var))
D = np.exp(-D/var)
m_MI = np.zeros(len(MI))
m_D = np.zeros(len(MI))
s_MI = np.zeros(len(MI))
s_D = np.zeros(len(MI))
for i in range(len(MI)):
    m_MI[i]=np.mean(np.diag(MI,i))
    s_MI[i]=np.std(np.diag(MI,i))
    m_D[i]=np.mean(np.diag(D,i))
    s_D[i]=np.std(np.diag(D,i))
if scale:
    tailCrop = int(round(0.2*len(MI)))
    tailCropIdx = int(len(MI)-tailCrop)
    xcrop = np.arange(1,tailCropIdx)
    xfull = np.arange(len(MI))

    # Scale MI
    popt_M,pcov_M = curve_fit(exp_func,xcrop,m_MI[1:tailCropIdx])
    perr_M = np.sqrt(np.diag(pcov_M))
    fity_M = [exp_func(xi,*popt_M) for xi in xfull]
    print('\nUsed scipy.optimize.curve_fit to fit function\n\ty=ae^(bx)+c \nto MI data with
(value, sigma):\n\ta = {:.4f}\n\tb = {:.4f}\n\tc =
{:.4f}'.format(*list(zip(popt_M.astype(float),perr_M.astype(float))))))
M_temp = np.zeros_like(MI)
done=False
intro_neg = []
for xi in xfull:
    theseIndices = kth_diag_indices(M_temp,xi)
    print(m_MI[xi], popt_M[2])
    if m_MI[xi] > popt_M[2] and done == False:
        M_temp[theseIndices]=MI[theseIndices]-np.mean(MI[theseIndices])+popt_M[2]
        if any(M_temp[theseIndices]<0):
            intro_neg.append(xi)
            M_temp[theseIndices] += np.abs(np.min(M_temp[theseIndices]))
    elif xi != 0: # Sometimes diagonal has been set to zero before-hand, this is not a
sign of leaving diagonal region
        done = True
    if done:
        M_temp[theseIndices]=MI[theseIndices]
        #M_temp[theseIndices]=popt_M[2]
if len(intro_neg) > 0:
    print('\nWarning: after rescaling, corrected negative values in the following
diagonals:\n{}`.format(intro_neg))
MI_scale=np.zeros_like(MI)
MI_scale=M_temp+np.triu(M_temp.T,1)
m_MIs = np.zeros_like(MI)
m_MIs[0]=popt_M[2]
for i in range(1,len(MI)):
    m_MIs[i]=np.mean(np.diag(MI_scale,i))

# Scale D (scaling should be done by function to avoid redundancy)
popt_D,pcov_D = curve_fit(exp_func,xcrop,m_D[1:tailCropIdx])
perr_D = np.sqrt(np.diag(pcov_D))
fity_D = [exp_func(xi,*popt_D) for xi in xfull]
print('\nUsed scipy.optimize.curve_fit to fit function\n\ty=ae^(-bx)+c \nto distance data
with (value, sigma):\n\ta = {:.4f}\n\tb = {:.4f}\n\tc =
{:.4f}'.format(*list(zip(popt_D.astype(float),perr_D.astype(float))))))
D_temp = np.zeros_like(D)
done=False
for xi in xfull:
    theseIndices = kth_diag_indices(D_temp,xi)
    if xi != 0:
        if m_D[xi] < popt_D[2] and done is False:
            D_temp[theseIndices]=D[theseIndices]-np.mean(D[theseIndices])+popt_D[2]
        else:
            done = True

```

```

        if done:
            D_temp[theseIndices]=D[theseIndices]
        else:
            D_temp[theseIndices]=0.
            #D_temp[theseIndices]=popt_D[2]
D_scale=np.zeros_like(D)
D_scale=D_temp+np.triu(D_temp.T,1)
m_Ds = np.zeros(len(D))
m_Ds[0]=popt_D[2]
for i in range(1,len(D)):
    m_Ds[i]=np.mean(np.diag(D_scale,i))

if plot:
    print('Plotting...')
    if runName == '':
        runName = 'run'
    figA,axA=plt.subplots(nrows=1,ncols=2)
    for i,t in enumerate((( 'MI',m_MI,s_MI,m_MIs,fity_M), ('Distance',m_D,s_D,m_Ds,fity_D))):
        #ax[0].plot(range(len(m_D)),m_D,'.-',c='r')
        axA[i].errorbar(range(len(t[1])),t[1],yerr=t[2],fmt='o',c='r')
        if scale:
            axA[i].plot(xfull,t[4],'-',c='k')
            axA[i].plot(range(len(t[3])),t[3],'.',c='b')
            axA[i].set_xlabel('Diagonal index (distance from primary diagonal at index 0)')
            axA[i].set_ylabel('<diag(i)>')
            axA[i].set_xlim((-5,len(MI)+5))
            axA[i].set_title('{}: {} matrix diagonal analysis'.format(runName,t[0]))
            axA[i].grid()
        if outDir is not None:
            if os.path.isdir(outDir):
                try:
                    plt.savefig(os.path.join(outDir,runName+'_diagonal-
analysis.eps'),transparent=True,close=True,verbose=True)
                except:
                    print('Failed to save diagonal-analysis figure for {}, displaying
instead.'.format(runName))
            else:
                plt.show()
                #plt.close()
        if not scale:
            figB,axB=plt.subplots(nrows=1,ncols=2)
        else:
            figB,axB=plt.subplots(nrows=1,ncols=4)
        axB[0].matshow(MI,cmap=plt.cm.afmhot,vmax=np.max(np.tril(MI,-5)))
        axB[0].set_title('Raw MI, M')
        axB[1].matshow(D,cmap=plt.cm.afmhot)
        axB[1].set_title('Raw distance, D_ij = e^(-Mij/var(M))')
        if scale:
            axB[2].matshow(MI_scale,cmap=plt.cm.afmhot)
            axB[2].set_title('Scaled MI, MIs_ij = MI_ij + MI_ij')
            axB[3].matshow(D_scale,cmap=plt.cm.afmhot)
            axB[3].set_title('Scaled distance, Ds_ij = D_ij + S_ij')

        if outDir is not None:
            if os.path.isdir(outDir):
                try:
                    plt.savefig(os.path.join(outDir,runName+'_mi-d-
ds.eps'),transparent=True,close=True,verbose=True)
                except:
                    print('Failed to save mi-d-ds figure for {}, displaying
instead.'.format(runName))
            else:
                plt.show()
                plt.close()
    if scale:
        return(MI_scale,D,D_scale)
    else:
        return(D)

```

```

def calculate_merged_mi_corr_matrices(miDict,norm=False,plot=False,outDir=None,figDict={}):
    """
    * Description

        Calculate average matrices for each protein in `miDict` across datasets, then update
        `miDict` with them.

    * Input:

        * `miDict`:

            A dictionary with MI matrices for more than one dataset for at least one protein,
            created using `create_mi_dict.py`.
            See documentation for `create_mi_dict.py` for more information about each item.
            Here, the basic hierarchical structure for a single dataset for your favorite protein
            A.

            * `consensus`: `dict`

                * `alignments`: `dict`

                    * `aln`: `Bio.Align.MultipleSeqAlignment`

            * `proteins`: `dict`

                * `your_favorite_protein_A`: `dict`

                    * `resiIndices`: `list` of `int`s

                    * `sets`: `dict`

                        * `ds1_1`: `dict`

                            * `data`: `dict`

                                * `dihedrals`: `dict`

                                    * `matrices`: `dict`

                                        * `m`: `numpy.ndarray`

                                * `coordinates`: `dict`

                                    * `matrices`: `dict`

                                        * `m`: `numpy.ndarray`

                            * `norm`

                                A boolean which specifies whether to take the L1 norm of matrices before averaging;
                                defaults to `False`.

                                * `plot`

                                    A boolean specifying whether or not to plot throughout execution.

                                * `outDir`

                                    The path to an output directory.
                                    If plotting has been requested, figures will be saved there.

                                * `figDict`

                                    A dictionary in which to store figures.

            * Output:

                * `miDict`
    """

```

```

    Like the input `miDict`, but updated with a new `merge` item in the `sets`
dictionary:
    * `proteins`: `dict`
        * `your_favorite_protein_A`: `dict`
            * `sets`: `dict`
                * `merge`: `dict`
                    * `data`: `dict`
                        * `dihedrals`: `dict`
                            * `matrices`: `dict`
                                * `mi_mean`: `numpy.ndarray`
                                    The mean calculated from dihedral MI matrices for all
datasets of your favorite protein A.
                                * `mi_std`: `numpy.ndarray`
                                    The standard deviation calculated from dihedral MI
matrices for all datasets of your favorite protein A.
                            * `coordinates`: `dict`
                                * `matrices`: `dict`
                                    * `mi_mean`: `numpy.ndarray`
                                        The mean calculated from coordinate MI matrices for
all datasets of your favorite protein A.
                                    * `mi_std`: `numpy.ndarray`
                                        The standard deviation calculated from coordinate MI
matrices for all datasets of your favorite protein A.
            * `figDict`
                ... If plotting was enabled, a dictionary with all plots will be returned.
    for prot in miDict['proteins'].keys():
        print('\nAggregating MI matrices for {}'.format(prot))
        mats =
{'dihedrals':{'m':[],'r':[],'m_harm':[],'ds':[]},'coordinates':{'m':[],'r':[],'m_harm':[],'ds':[]}}
        for dSet in miDict['proteins'][prot]['sets'].keys():
            if dSet == 'merge':
                continue
            for dType in miDict['proteins'][prot]['sets'][dSet]['data'].keys():
                if dType not in ('coordinates','dihedrals'):
                    continue
                for mType in mats[dType].keys():
                    try:
mats[dType][mType].append(miDict['proteins'][prot]['sets'][dSet]['data'][dType]['matrices'][mType
])
                        except KeyError:
                            pass
                    except:
                        raise
        print('\tCalculating mean/std across all datasets for {}'.format(prot))
        try:
            assert(miDict['proteins'][prot]['sets']['merge']['data'])
        except KeyError:

```

```

miDict['proteins'][prot]['sets']['merge'] = {'data':{}}
for dType in ('dihedrals', 'coordinates'):
miDict['proteins'][prot]['sets']['merge']['data'][dType] = {'matrices':{}}
for mType in mats[dType].keys():
    if norm:
        #[np.fill_diagonal(m,0) for m in mats[dType]]

#miDict['proteins'][prot]['sets']['merge']['data'][dType]={'matrices':{'mi_mean':normalize(np.mean(mats[dType],0), axis=1,
norm='l1'),'ds_mean':np.mean(dsMats[dType],0),'mi_std':normalize(np.std(mats[dType],0), axis=1,
norm='l1'),'ds_std':np.std(dsMats[dType],0)}}
        if len(mats[dType][mType]) > 0:
            for i in range(len(mats[dType])):
                mats[dType][mType][i] = mats[dType][mType][i] -
mats[dType][mType][i].min()
                mats[dType][mType][i] /= mats[dType][mType][i].max() -
mats[dType][mType][i].min()
            #else:

#miDict['proteins'][prot]['sets']['merge']['data'][dType]={'matrices':{'mi_mean':np.mean(mats[dType],0),'mi_mean_norm':normalize(np.mean(mats[dType],0), axis=1,
norm='l1'),'ds_mean':np.mean(dsMats[dType],0),'mi_std':np.std(mats[dType],0),'mi_std_norm':normalize(np.std(mats[dType],0), axis=1, norm='l1'),'ds_std':np.std(dsMats[dType],0)}}
        if len(mats[dType][mType]) > 0 and mType == 'm':

miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].update({'mi_mean':np.mean(mats[dType][mType],0),'mi_std':np.std(mats[dType][mType],0)})
        if len(mats[dType][mType]) > 0 and mType == 'm_harm':

miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].update({'mi_harm_mean':np.mean(mats[dType][mType],0),'mi_harm_std':np.std(mats[dType][mType],0)})
        if len(mats[dType][mType]) > 0 and mType == 'r':

miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].update({'corcoef_mean':np.mean(mats[dType][mType],0),'corcoef_std':np.std(mats[dType][mType],0)})
        if len(mats[dType][mType]) > 0 and mType == 'ds':

miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].update({'ds_mean':np.mean(mats[dType][mType],0),'ds_std':np.std(mats[dType][mType],0)})
        if plot:
            for dType in ('dihedrals', 'coordinates'):
                if len(miDict['proteins'][prot]['sets'].keys()) <= 2:
                    continue
                thisFig = 'mergeMats_{_}_{_}'.format(prot,dType)
                print('\tPlotting {_}...'.format(thisFig))
                mats =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].keys()
                n_mats = len(mats)
                figDict[thisFig] = {}
                figDict[thisFig]['fig'],figDict[thisFig]['ax'] =
plt.subplots(nrows=1,ncols=n_mats,squeeze=False,figsize=(10*n_mats,10))
                for i,mat in enumerate(mats):
                    if mat == 'm':
                        cmap=plt.cm.afmhot_r
                    else:
                        cmap=plt.cm.afmhot
                    if mat.split('_')[0] == 'corcoef':
                        vmin = 0
                        vmax = 1
                    else:
                        vmin =
np.percentile(miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'][mat][np.triu_indices_from(miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'][mat],5)],1)
                        vmax =
np.percentile(miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'][mat][np.triu_indices_from(miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'][mat],5)],99)

p=figDict[thisFig]['ax'][0][i].matshow(miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'][mat],cmap=cmap,vmin=vmin,vmax=vmax)

```

```

figDict[thisFig]['div'] = make_axes_locatable(figDict[thisFig]['ax'][0][i])
figDict[thisFig]['cax'] =
figDict[thisFig]['div'].append_axes('right', size='5%', pad=0.05)
figDict[thisFig]['cbar'] = plt.colorbar(p, cax=figDict[thisFig]['cax'])
try:
    j = [int(n) for n in figDict[thisFig]['ax'][0][i].get_xticks()[::-1]]
    k = [miDict['proteins'][prot]['resiIndices'][n] for n in j]
    figDict[thisFig]['ax'][0][i].set_xticklabels(k)
    figDict[thisFig]['ax'][0][i].set_yticklabels(k)
except:
    print('\tFailed to relabel axes ticks, skipping. Check
miDict[\`protein\`][prot][\`set\`][dataSet][\`resiIndices\`] and try again.')
figDict[thisFig]['ax'][0][i].set_title('{ }-{}-{}'.format(prot, dType ,mat))
figDict[thisFig]['ax'][0][i].set_xlabel('residue i')
figDict[thisFig]['ax'][0][i].set_ylabel('residue j')

#plt.colorbar(p, ax=figDict[thisFig]['ax'].ravel().tolist())
if outDir is None:
    plt.show()
elif outDir is not None:
    try:
plt.savefig(os.path.join(outDir, thisFig+'.eps'), transparent=True, format='eps')
    except:
        print('\tFailed to save figure {}; will carry on
anyways.'.format(thisFig))
        plt.close()
    if plot:
        return(miDict, figDict)
    else:
        return(miDict)

def calculate_merged_cm_matrices(miDict, plot=False, outDir=None, figDict={}):
    """
    * Description

    Calculate average contact graph for each protein in `miDict` across datasets, then update
    `miDict` with them.
    In the future, this and `calculate_merged_mi_matrices` could be merged into a single non-
    redundant function.

    * Input:

    * `miDict`:

    A dictionary with contact graphs for more than one dataset for at least one protein,
    created using `create_mi_dict.py`.
    See documentation for `create_mi_dict.py` for more information about each item.
    Here, the basic hierarchical structure for a single dataset for your favorite protein
    A.

    * `consensus`: `dict`

    * `alignments`: `dict`

    * `aln`: `Bio.Align.MultipleSeqAlignment`

    * `proteins`: `dict`

    * `your_favorite_protein_A`: `dict`

    * `resiIndices`: `list` of `int`s

    * `sets`: `dict`

    * `ds1_1`: `dict`

    * `data`: `dict`

```

```

* `dihedrals`: `dict`
  * `matrices`: `dict`
    * `m`: `numpy.ndarray`
* `coordinates`: `dict`
  * `matrices`: `dict`
    * `m`: `numpy.ndarray`
* `contacts`: `dict`
  * `matrices`: `dict`
    * `d_raw`: `numpy.ndarray`
    * `d_bin`: `numpy.ndarray`
* `plot`
  A boolean specifying whether or not to plot throughout execution.
* `outDir`
  The path to an output directory.
  If plotting has been requested, figures will be saved there.
* `figDict`
  A dictionary in which to store figures.
* Output:
  * `miDict`
    Like the input `miDict`, but updated with a new `merge` item called 'contacts' in the
    `sets` dictionary:
    * `proteins`: `dict`
      * `your_favorite_protein_A`: `dict`
        * `sets`: `dict`
          * `merge`: `dict`
            * `data`: `dict`
              * `contacts`: `dict`
                * `matrices`: `dict`
                  * `d_raw_mean`: `numpy.ndarray`
                    The mean calculated over raw contact graphs from all
                    datasets of your favorite protein A.
                  * `d_raw_std`: `numpy.ndarray`
                    The standard deviation calculated over raw contact
                    graphs from all datasets of your favorite protein A.
                  * `d_bin_mean`: `numpy.ndarray`
                    The mean calculated over binarized contact graphs
                    from all datasets of your favorite protein A.

```

```

* `d_bin_std`: `numpy.ndarray`

The standard deviation calculated over binarized
contact graphs from all datasets of your favorite protein A.
* `figDict`

...
If plotting was enabled, a dictionary with all plots will be returned.
...
for prot in miDict['proteins'].keys():
    print('\nAggregating distance and contact graph matrices for {}'.format(prot))
    d_mats = {'contacts': [], 'distance': []}
    for dSet, dType in
it.product(list(miDict['proteins'][prot]['sets'].keys()), ['distance', 'contacts']):
        if dSet == 'merge':
            continue
        try:
d_mats[dType].append(miDict['proteins'][prot]['sets'][dSet]['data'][dType.split('_')[0]]['matrices']
s')[dType])
            except KeyError:
                pass
            except:
                raise
print('\tCalculating contact graph mean/std across all datasets for {}'.format(prot))
try:
    assert(miDict['proteins'][prot]['sets']['merge']['data'])
except KeyError:
    miDict['proteins'][prot]['sets']['merge'] = {'data': {}}
for k in d_mats.keys():
    try:
        assert(miDict['proteins'][prot]['sets']['merge']['data'][k])
    except:
        miDict['proteins'][prot]['sets']['merge']['data'].update({k: {}})
[miDict['proteins'][prot]['sets']['merge']['data'][t].update({'matrices': {}}) for t in
d_mats.keys()]

[miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices'].update({'m_mean': np.mean(d_
mats[dType], 0), 'm_std': np.std(d_mats[dType], 0)}) for dType in d_mats.keys() if len(d_mats[dType])
> 0]

if plot:
    if len(miDict['proteins'][prot]['sets'].keys()) <= 2: # Don't try to plot anything
with only one set
        continue
        thisFig = 'meanCmap_{}'.format(prot)
        print('\tPlotting {}'.format(thisFig))
        mats = {}

[mats.update({'{}_{}'.format(d,t): miDict['proteins'][prot]['sets']['merge']['data'][d]['matrices'
][t]} for d,t in it.product(d_mats.keys(), ['m_mean', 'm_std'])]
    if len(mats.keys()) != 4:
        print('\t\tExpected four matrices, skipping...')
        continue
    figDict[thisFig] = {}
    figDict[thisFig]['fig'], figDict[thisFig]['ax'] =
plt.subplots(nrows=2, ncols=2, squeeze=False, figsize=(20, 20))
    for col, this_mat in enumerate(sorted(mats.keys())):
        if 'distance' in this_mat:
            cmap=plt.cm.afmhot_r
            row = 0
            cbar_title = 'Distance (Angstroms)'
        elif 'contacts' in this_mat:
            cmap=plt.cm.afmhot
            row = 1
            cbar_title = 'Contact (fractional over MSA)'
        else:
            print('\t\tUnexpected matrix format {}, skipping!'.format(mat))
            continue
        p=figDict[thisFig]['ax'][row][col%2].matshow(mats[this_mat], cmap=cmap)

```

```

        figDict[thisFig]['div'] = make_axes_locatable(figDict[thisFig]['ax'][row][col%2])
        figDict[thisFig]['cax'] =
figDict[thisFig]['div'].append_axes('right',size='5%',pad=0.05)
        figDict[thisFig]['cbar'] =
plt.colorbar(p,cax=figDict[thisFig]['cax'],label=cbar_title)
        try:
            j = [int(n) for n in figDict[thisFig]['ax'][row][col%2].get_xticks()[:-1]]
            k = [miDict['proteins'][prot]['resiIndices'][n] for n in j]
            figDict[thisFig]['ax'][row][col%2].set_xticklabels(k)
            figDict[thisFig]['ax'][row][col%2].set_yticklabels(k)
        except:
            print('\tFailed to relabel axes ticks, skipping. Check
miDict['protein\'][prot][\set\[dataSet][\resiIndices\] and try again.')
            figDict[thisFig]['ax'][row][col%2].set_title('{}-{}-{}'.format(prot, dType
, this_mat))
            figDict[thisFig]['ax'][row][col%2].set_xlabel('residue i')
            figDict[thisFig]['ax'][row][col%2].set_ylabel('residue j')
            figDict[thisFig]['fig'].suptitle('{}: Contact graph analysis over
datasets'.format(prot))
            if outDir is None:
                plt.show()
            elif outDir is not None:
                try:
plt.savefig(os.path.join(outDir,thisFig+'.eps'),transparent=True,format='eps')
            except:
                print('\tFailed to save figure {}; will carry on anyways.'.format(thisFig))
                plt.close()

        if plot:
            return(miDict,figDict)
        else:
            return(miDict)

def consensus_tensor_from_msa(msa,dataDict,refSeqName=None, rescale=True, dimRedMode='mean',
dat_type=('data','units'), runInfo=None,outDir=None,plot=False):
    """
    * Description:

        Use a Bio.AlignIO MSA to create a consensus tensor from multiple data matrices,
optionally rescaling the data relative to 25th/75th percentile prior to dimension reduction.

    * Input:

        * `msa`

            A `Bio.Align.MultipleSeqAlignment` object or a path to an alignment file to be
loaded.

            Each sequence in the alignment should correspond to one MI matrix in miDict, and the
names should match.

            Why is this not automatically read from `miDict`? (TODO)

        * `dataDict`

            A dictionary of matrices (numpy.ndarray instances) keyed with a descriptive name that
matches only one sequence name in `msa`.
            `dataDict` is keyed as follows:

            * * `your_favorite_protein_A`: `dict`
                * `m`: `numpy.ndarray`

        * `refSeqName`

            The name of a sequence in `msa` to use as a reference for numbering.
            If specified, consensus matrices will be generated which map to the sequence of
`refSeqName`.

```

- * ``rescale``
A boolean which specifies whether or not to rescale MI data to 25th/75th percentile of reference data prior to dimension reduction; defaults to ``True``. This is definitely a good idea for coordinate MI, but possibly not for dihedral MI.
- * ``dat_type``
``('data_type','data_units')``
- * ``dimRedMode``
``{mean, iso}``
A string specifying the dimension reduction mode; if ``mean``, matrices will simply be averaged. If equal to ``iso``, the isomap method will be used.
- * ``outDir``
Path to directory in which to save calculated MI tensors.
- * ``plot``
A boolean; if true, draw plots.
- * **Output:**
 - * ``M``
A dictionary with data matrices keyed as follows:
 - * ``full``
A 3-dimensional consensus tensor with gapped positions for a given matrix set to ``np.nan``.
 - * ``consensus``
A 3-dimensional consensus tensor like ``full``, but with gaps propagated to all matrices. In other words, if one sequence has a gap in the MSA, the equivalent position in all matrices will be set to ``np.nan``.
 - * ``consensus_nogap``
A 3-dimensional consensus tensor like ``consensus``, but with all ``np.nan``-containing rows and columns removed.
 - * ``compressed``
A 2-dimensional consensus matrix created from ``consensus``, where positions with gaps have been set to ``np.nan``.
 - * ``consensus_compressed``
Like ``compressed``, but with rows and columns with values equal to ``np.nan`` removed.
 - * ``sequence_compressed``
Like ``consensus_compressed``, but mapped to the sequence in the MSA denoted by ``refSeqName``.
 - * ``consensus_nogap_rescale``
Like ``consensus_nogap``, but with all slices through the third dimension scaled relative to the first.
 - * ``consensus_compressed_rescale``

```

        Like `consensus_compressed`, but created from `consensus_nogap_rescale`.
    * `lookup`
        A dictionary of dictionaries, with residue number lookup info for the different
        consensus data matrices calculated:
    * `full`
        A dictionary of dictionaries, where each item specifies residue numbering for a
        given protein in the MSA with respect to `M['full']`.
    * `your_favorite_protein_A`: `list`
        A list of length equal to the length of `msa`, where each entry corresponds
        to a position in that alignment.
        Entries can either be residue numbers (i.e., `int`s) or gaps (i.e., `str`,
        `_`),
    * `consensus`
        A dictionary of dictionaries, where each item specifies residue numbering for a
        given protein in the MSA with respect to `M['consensus']`.
        Like `full`, but with gaps propagated throughout the alignment.
    * `your_favorite_protein_A`: `list`
        A list of length equal to the length of `msa`, where each entry corresponds
        to a position in that alignment.
        Entries can either be residue numbers (i.e., `int`s) or gaps (i.e., `str`,
        `_`),
    * `consensus_compressed`
        A dictionary of dictionaries, where each item specifies residue numbering for a
        given protein in the MSA with respect to `M['consensus_compressed']`.
        Like `full` and `consensus`, but with all gapped positions removed.
    * `your_favorite_protein_A`: `list`
        A list of length equal to the length of the 100% conserved subset of `msa`,
        where each entry corresponds to a position in that subalignment.
        Entries can only be residue numbers (i.e., `int`s).
    ...

def save_view_fig(outDir, file_name):
    if outDir is None:
        plt.show()
    elif os.path.isdir(outDir):
        try:
plt.savefig(os.path.join(outDir, file_name), transparent=True, close=True, verbose=True)
        except:
            print('Failed to save figure; displaying instead.')
            plt.show()
        plt.close()

    if type(dat_type) is not tuple or len(dat_type) != 2:
        print('Invalid data type tuple specified; using default (\'data\', \'units\').')
        dat_type, dat_units = ('data', 'units')
    else:
        dat_type, dat_units = [str(s) for s in dat_type]

    print('Loading MSA...') # Use Bio.Align to load an MSA, named msa, in one of several
supported formats
    if type(msa) == dict:
        msaTitle, msa=list(msa.items())[0]
    else:

```

```

msaTitle = 'msa'
if not isinstance(msa,Align.MultipleSeqAlignment): # Get the MSA all loaded up
if os.path.isfile(msa):
    try:
        msa=AlignIO.read(msa,'fasta')
    except ValueError:
        try:
            msa=AlignIO.read(msa,'clustal')
        except ValueError:
            try:
                msa=AlignIO.read(msa,'stockholm')
            except ValueError:
                try:
                    msa=AlignIO.read(msa,'phylip')
                except:
                    raise ValueError('Could not load supplied MSA! Is it FASTA, Clustal,
Stockholm, or PHYLIP formatted?')
            else:
                raise ValueError('Supplied multiple sequence alignment could not be loaded or
parsed!')
        for s in msa:
            if len(s) != len(msa[0]):
                raise ValueError('All sequences in the supplied alignment should be of the same
length!')

    print('Checking {} dictionary...'.format(dat_type))
    if type(dataDict) != dict:
        raise ValueError('dataDict must be a dictionary!')
    if dataDict == {}:
        print('Empty dictionary provided, exiting!')
        return(None,None)
    for k,v in dataDict.items():
        if not isinstance(v,np.ndarray):
            raise ValueError('dataDict values must be numpy arrays with nResis x nResis
dimensions!')

    print('Constructing alignment-indexed {} tensor...'.format(dat_type))
    alnLength = len(msa[0]) # Length of sequences in the alignment, probably including gaps
    dataDictLen = len(dataDict.keys()) # The number of entries in the data dictionary for
different proteins
    dataDims = [alnLength,alnLength,dataDictLen]
    M_full = np.zeros(dataDims)*np.nan # The consensus data matrix. After merging, any x or y
slice with an NaN will be deleted, as NaN indicates a position which could not be aligned.
    M_full_lookup = {}
    prot_order = []
    for dat_idx,dat_tup in enumerate(dataDict.items()):
        dat_id,M = dat_tup
        matches = [(i,seq.id) for i,seq in enumerate(msa) if dat_id.split('_')[0] in seq.id and
dat_id.split('_')[1] in seq.id] # this of tuples (i, seq), where i is the index of sequence seq
in the MSA.
        if len(matches) == 1: # There should only be one sequence per matrix, so this should
always be a list with only one tuple. If len == 0 or len > 1, throw an error below.
            matches=matches[0]
            datResis = list(range(len(M)))
            if len(M) != sum([r.isalnum() for r in msa[matches[0]]]):
                s=0
                for r in msa[matches[0]]:
                    if r.isalnum():
                        s+=1
                print('{}\t{}\t{}'.format(r,r.isalnum(),s))
                raise ValueError('Sequence {} matches {} matrix label {}, but matrix has
different dimensions than ungapped sequence ({} vs. {},
respectively)!.format(msa[matches[0]].id,dat_type,dat_id,sum([r.isalnum() for r in
msa[matches[0]]],len(M)))
            datResisMap = []
            datResisIdx = 0
            for r in msa[matches[0]]: # Build list datResisMap from datResis to use for lookup
table
                if r.isalnum():

```

```

        datResisMap.append(datResis[datResisIdx])
        datResisIdx+=1
    else:
        datResisMap.append('-')
        M_full_lookup[dat_id] = datResisMap
        print('\tFound {} matrix {} in MSA position {} with sequence name
{}...'.format(dat_type,dat_id,matches[0],matches[1]))
        prot_order.append((dat_idx,dat_id))
        for pair in it.product(enumerate(msa[matches[0]]),enumerate(msa[matches[0]])):
            pairList = [item for sublist in pair for item in sublist]
            if pairList[1].isalnum() and pairList[3].isalnum():
                M_full[pairList[0],pairList[2],dat_idx] =
M[datResisMap[pairList[0]],datResisMap[pairList[2]]]
            elif len(matches) == 0:
                raise ValueError('Could not find {} matrix name {} in MSA sequence
names!'.format(dat_type,dat_id))
            elif len(matches) > 1:
                print(matches)
                raise ValueError('Multiple instances of {} matrix name {} found in MSA sequence
names; only one sequence per matrix name should be present!'.format(dat_type,dat_id))

print('Reducing alignment-indexed {} tensor to consensus {} tensor...'.format(dat_type))
M_consensus_lookup = M_full_lookup
for kTup in it.product(M_consensus_lookup.keys(),M_consensus_lookup.keys()):
    if kTup[0] != kTup[1]:
        idxPair = zip(M_consensus_lookup[kTup[0]],M_consensus_lookup[kTup[1]])
        for i,idx in enumerate(idxPair):
            if idx[1] == '-':
                M_consensus_lookup[kTup[0]][i] = '-'
M_consensus = M_full.copy()
for l,m in it.product(range(dataDictLen),range(alnLength)): # For each matrix, iterate over
rows looking for NaNs
    if sum([np.isnan(v) for v in M_full[m,:,l].flatten()]) == alnLength:
        #print('\tFound full list of NaNs at row {} of {},
masking.'.format(m,list(dataDict.keys())[l]))
        M_consensus[m,:,:] = np.nan
        M_consensus[:,m,:] = np.nan

print('Compressing consensus {} tensor into two dimensions using
{}...'.format(dat_type,dimRedMode))
M_consensus_compressed_lookup = {}
for k,v in M_consensus_lookup.items():
    M_consensus_compressed_lookup[k]=[r for r in v if r != '-']
    consCompLen = len(list(M_consensus_compressed_lookup.values())[0])

M_consensus_nogap=np.ma.compressed(np.ma.fix_invalid(M_consensus)).reshape([consCompLen,consCompL
en,dataDictLen])

if rescale: # Transform data based on 25th/75th percentiles of a reference matrix.
    M_consensus_nogap_rescale = np.zeros_like(M_consensus_nogap)
    ref_idx =0
    ref_dat =
M_consensus_nogap[:,:,ref_idx][np.triu_indices_from(M_consensus_nogap[:,:,ref_idx],1)] # Get
reference data as upper triangle of first matrix
    ref_dat_perc = (np.percentile(ref_dat,25),np.percentile(ref_dat,75)) # Compute 25th/75th
percentiles
    for i in range(dataDims[2]):
        if i != ref_idx: # If we're not at the reference matrix, transform this matrix to
look like the reference
            print('\nRescaling [,:,{}]'.format(i))
            this_dat =
M_consensus_nogap[:,:,i][np.triu_indices_from(M_consensus_nogap[:,:,i],1)]
            this_dat_perc = (np.percentile(this_dat,25),np.percentile(this_dat,75))
            a = np.array([[this_dat_perc[0],1],[this_dat_perc[1],1]])
            b = np.array(ref_dat_perc)
            x = np.linalg.solve(a,b) # Solve for scale and translation params
            if not np.allclose(np.dot(a,x),b):
                raise ValueError('Problem solving for scaling parameters!')
            print('Input:\na: {} \nb: {} \nOutput:\nx: {}'.format(a,b,x))

```

```

        M_consensus_nogap_rescale[:, :, i] = M_consensus_nogap[:, :, i]*x[0]+x[1] # Transform
y = mx + b, where we just solved for m and b
    else: # If we're at the reference, just copy the data
        print('\nSkipping reference set...')
        M_consensus_nogap_rescale[:, :, i] = M_consensus_nogap[:, :, i]
print('\nDone!\n')
if plot:
    figA, axA=plt.subplots(1, dataDims[2], figsize=(dataDims[2]*10, 10))
    for i in range(dataDims[2]):

nR, binsR, patchesR=axA[i].hist(M_consensus_nogap[:, :, i][np.triu_indices_from(M_consensus_nogap[:, :, i], 1)], range=[0, np.percentile(M_consensus_nogap, 98)], bins=int(dataDims[1]/2), normed=True, alpha=0.5, histtype='stepfilled', edgecolor='none', facecolor='b', label='raw')

nT, binsT, patchesT=axA[i].hist(M_consensus_nogap_rescale[:, :, i][np.triu_indices_from(M_consensus_nogap_rescale[:, :, i], 1)], range=[0, np.percentile(M_consensus_nogap, 98)], bins=int(dataDims[1]/2), normed=True, alpha=0.5, histtype='stepfilled', edgecolor='none', facecolor='g', label='transform')
        #plt.legend()
        if i == ref_idx:
            axA[i].set_title('[:, :, {}] (ref)'.format(i))
        else:
            axA[i].set_title('[:, :, {}]'.format(i))
            axA[i].set_xlabel('{} ({} )'.format(dat_type, dat_units))
            axA[i].set_ylabel('Probability density')
            axA[i].grid(True)
            save_view_fig(outDir, '{}_{}_transform_hist.eps'.format(runInfo, dat_type))

if dimRedMode == 'mean':
    M_consensus_compressed=np.mean(M_consensus_nogap, 2)
    M_compressed=np.nanmean(M_full, 2)
    if rescale:
        M_consensus_rescale_compressed=np.mean(M_consensus_nogap_rescale, 2)
elif dimRedMode == 'iso':
    ...
else:
    print(dimRedMode)
    raise ValueError('Invalid dimension reduction mode specified.')

if refSeqName is not None: # If requested, create a version of M_compressed with values at
all non-gapped positions in a reference sequence.
    M_compressed_seq_temp = M_compressed.copy()
    matches = [seq for seq in msa if refSeqName.split('_')[0] in seq.id]
    if len(matches) > 1:
        print('Multiple reference sequences ({} ) match input string, using sequence
{}.'.format([m.id for m in matches], matches[0].id))
        print('Creating {} matrix mapped to reference sequence
{}...'.format(dat_type, matches[0].id))
        seqCompress = True
        for i, r in enumerate(matches[0].seq):
            if r == '-':
                M_compressed_seq_temp[:, i] = np.nan
                M_compressed_seq_temp[:, i] = np.nan
        seqLen = len([r for r in matches[0].seq if r != '-'])
        M_compressed_seq =
np.ma.compressed(np.ma.fix_invalid(M_compressed_seq_temp)).reshape([seqLen, seqLen])
    else:
        print(refSeqName.split('_')[0])
        print([seq.id for seq in msa])
        print('Couldn't find reference sequence to create sequence-compressed {} matrix,
continuing.'.format(dat_type))
        seqCompress = False
else:
    seqCompress = False

if plot:
    print('Plotting...')
    figA = {}
    figA['fig'], figA['ax']=plt.subplots(nrows=2, ncols=dataDictLen, squeeze=False)
    vmax=np.percentile(M_consensus[~np.isnan(M_consensus)], 99)

```

```

    for p in range(dataDictLen):
        q=figA['ax'][0,p].matshow(M_full[:, :, p], cmap=plt.cm.afmhot, vmax=vmax)
        figA['ax'][0,p].set_title('{}: full {}'.format(list(dataDict.keys())[p], dat_type))
        figA['ax'][0,p].set_xlabel('Alignment idx i')
        figA['ax'][0,p].set_ylabel('Alignment idx j')
        figA['div_q_'+str(p)] = make_axes_locatable(figA['ax'][0,p])
        figA['cax_q'+str(p)] = figA['div_q_'+str(p)].append_axes('right', size='5%', pad=0.05)
        figA['cbar_q'+str(p)] = plt.colorbar(q, cax=figA['cax_q'+str(p)], label='{}'.format(dat_type, dat_units))
        s=figA['ax'][1,p].matshow(M_consensus[:, :, p], cmap=plt.cm.afmhot, vmax=vmax)
        figA['ax'][1,p].set_title('{}: consensus {}'.format(list(dataDict.keys())[p],
dat_type))
        figA['ax'][1,p].set_xlabel('Alignment idx i')
        figA['ax'][1,p].set_ylabel('Alignment idx j')
        figA['div_s_'+str(p)] = make_axes_locatable(figA['ax'][1,p])
        figA['cax_s'+str(p)] = figA['div_s_'+str(p)].append_axes('right', size='5%', pad=0.05)
        figA['cbar_s'+str(p)] = plt.colorbar(s, cax=figA['cax_s'+str(p)], label='{}'.format(dat_type, dat_units))
        save_view_fig(outDir, 'consensus_{}_{}_full_{}.eps'.format(msaTitle, runInfo, dat_type))

    figB, axB = plt.subplots(1)
    p=axB.matshow(M_consensus_compressed, cmap=plt.cm.afmhot)
    axB.set_title('Compressed consensus {} matrix {}'.format(dat_type, dimRedMode))
    axB.set_xlabel('Alignment idx i')
    axB.set_ylabel('Alignment idx j')
    divB = make_axes_locatable(axB)
    caxB = divB.append_axes('right', size='5%', pad=0.05)
    cbarB = plt.colorbar(p, cax=caxB, label='{}'.format(dat_type, dat_units))

    save_view_fig(outDir, 'consensus_{}_{}_compressed_{}.eps'.format(msaTitle, runInfo, dat_type))

    figC, axC = plt.subplots(1)
    p=axC.matshow(M_compressed, cmap=plt.cm.afmhot)
    axC.set_title('Compressed {} matrix {}'.format(dat_type, dimRedMode))
    axC.set_xlabel('Alignment idx i')
    axC.set_ylabel('Alignment idx j')
    divC = make_axes_locatable(axC)
    caxC = divC.append_axes('right', size='5%', pad=0.05)
    cbarC = plt.colorbar(p, cax=caxC, label='{}'.format(dat_type, dat_units))

    save_view_fig(outDir, 'consensus_{}_{}_full_compressed_{}.eps'.format(msaTitle, runInfo, dat_type))

    if seqCompress:
        figD, axD = plt.subplots(1)
        p=axD.matshow(M_compressed_seq, cmap=plt.cm.afmhot)
        axD.set_title('Sequence compressed {} matrix {}'.format(dat_type, dimRedMode))
        axD.set_xlabel('Alignment idx i')
        axD.set_ylabel('Alignment idx j')
        divD = make_axes_locatable(axD)
        caxD = divD.append_axes('right', size='5%', pad=0.05)
        cbarD = plt.colorbar(p, cax=caxD, label='{}'.format(dat_type, dat_units))

    save_view_fig(outDir, 'consensus_{}_{}_sequence_compressed_{}.eps'.format(msaTitle, runInfo, dat_type))

    print('Finishing up...')
    lookup =
{'full':M_full_lookup, 'consensus':M_consensus_lookup, 'consensus_compressed':M_consensus_compressed_lookup, 'full_order':prot_order}
    M =
{'full':M_full, 'consensus':M_consensus, 'consensus_nogap':M_consensus_nogap, 'compressed':M_compressed, 'consensus_compressed':M_consensus_compressed}
    if seqCompress:
        M.update({'sequence_compressed':M_compressed_seq})
    if rescale:
M.update({'consensus_nogap_rescale':M_consensus_nogap_rescale, 'consensus_compressed_rescale':M_consensus_compressed_rescale})
    return(M, lookup)

```

```

def noise_analysis(M, resiList=[], diag_suppress=1, plot=False, runName='run', outDir=None):
    """
    * Description:

        Compare any number of two-dimensional matrices of equal dimensions by difference
        analysis.

    * Input:

        * `M`

            A three-dimensional matrix, where each slice in the first two dimensions corresponds
            to a given matrix for comparison.

        * `resiList`

            A list of integers corresponding to the indexing of `M` with respect to some protein.

        * `diag_suppress`

            An integer which specifies how many diagonals away from the primary diagonal to
            ignore during analysis.

        * `plot`

            A boolean specifying whether or not to plot the data.

        * `runName`

            A descriptive string.

        * `outDir`

            A directory in which to save any output.
            Required to save figures if plotting is enabled.

    * Output:

        * `N`

            Difference matrix, where the mean of `M` along the third dimension has been
            subtracted from each matrix in the first two dimensions of `M`.

        * `mu_input`

            Mean of `M` computed along the third dimension.

        * `std_input`

            Standard deviation of `M` computed along the third dimension.

        * `mu_diff`

            Mean of `N` computed along the third dimension.

        * `std_diff`

            Standard deviation of `N` computed along the third dimension.
    """
    if len(M.shape) != 3:
        raise ValueError('Input matrix M must be three dimensional.')

    n_mats = M.shape[2]
    M_mean = np.mean(M, axis=2)
    N = np.zeros_like(M)
    P = np.zeros((len(np.triu_indices_from(M[:, :, 0])[0]), n_mats))
    for m in range(0, n_mats):
        N[:, :, m] = M[:, :, m] - M_mean

```

```

mu_input = np.mean(M)
sigma_input = np.std(M)
mu_diff = np.mean(N)
sigma_diff = np.std(N)
print('\nInput matrix: {: 0.3f} +/- {: 0.3f} bits\nDifference matrix: {: 0.3f} +/- {: 0.3f}
bits\n'.format(mu_input,sigma_input,mu_diff,sigma_diff))
if diag_suppress is not None:
    P = N[np.triu_indices_from(N[:, :, 0],diag_suppress)]
else:
    P = N[np.triu_indices_from(N[:, :, 0])]

if plot:
    figA,axA=plt.subplots(nrows=2,ncols=1,squeeze=False,figsize=(25,15))

nA,bA,pA=axA[0][0].hist([M[np.triu_indices_from(M[:, :, 0],1)].flatten()],alpha=0.5,bins=M.shape[0]
,label=['MI', 'Difference MI'])
    axA[0][0].legend()
    axA[0][0].set_title('{: MI across datasets'.format(runName))
    axA[0][0].set_xlabel('Difference MI (bits)')
    axA[0][0].set_ylabel('Counts')
    nB,bB,pB=axA[1][0].hist([P[:,i] for i in
range(n_mats)],alpha=0.5,bins=M.shape[0],label=['Matrix {}'.format(j) for j in range(n_mats)])
    axA[1][0].legend()
    axA[1][0].set_title('{: mean difference histograms'.format(runName))
    axA[1][0].set_xlabel('Difference MI (bits)')
    axA[1][0].set_ylabel('Counts')
    if outDir is not None:
        try:
plt.savefig(os.path.join(outDir,'{}_difference_histograms.eps'.format(runName)),transparent=True,
close=True,verbose=True,format='eps')
        except:
            print('Failed to save figure; displaying instead.')
            plt.show()
    return(N,mu_input,sigma_input,mu_diff,sigma_diff)

def
matrix_statistics(M, fitCrop=0.01, perclist=[0.90,0.95,0.99], fitfn='lognorm', plot=True, outDir=None,
runName=None):
    """
    * Description:

    Calculate a variety of statistical properties for some symmetric MI matrix `M`.

    * Input:

    * `M`

    A symmetric MI matrix of type `numpy.ndarray`.

    * `fitCrop`

    A float specifying the amount of data to ignore on the positive side of the MI
distrubution from `M` during fitting.

    * `perclist`

    A list of floats specifying the percentiles at which to threshold the data.

    * `fitfn`

    `('norm', 'lognorm', 'gamma')`

    The type of distribution to fit to the MI distribution from `M`.

    * `plot`

    Boolean specifying whether or not to create various plots.

```

```

* `outdir`
    If provided, plots will be output to this directory.

* `runName`
    A descriptive title.

* Output:

* `percOut`:
    A list of tuples, where each entry `(percentile, mi_at_percentile)` gives the mutual
information in `M` at a given percentile in `percList`.

* `M_perc`
    A three-dimensional matrix, where each slice in the first two dimensions corresponds
to `M` thresholded at a given percentile in `percList`.
'''
if runName is None:
    runName = 'run'
if type(percList) != list:
    percList=[0.90,0.95,0.99]
if len(np.shape(M)) == 2:
    dims = 2
    M_depth = 1
    utM = M[np.triu_indices_from(M,1)] # Store upper triangle of M without diagonal as vector
utM
    # if diagNorm:
    #     diag = np.mean(np.diag(M))
    #     utM = utM/diag
elif len(np.shape(M)) == 3:
    dims = 3
    M_depth = np.shape(M)[2]
    utM = M[np.triu_indices_from(M[:, :, 0], 1)]
    # if diagNorm:
    #     for i in range(M_depth):
    #         diag = np.diag(M[:, :, i]) # Get diagonal
    #         diag = diag[~np.isnan(diag)] # Remove NaNs
    #         utM[:, i] = utM[:, i]/np.mean(diag) # Divide by diagonal mean
    utM = utM.flatten()
    utM=utM[~np.isnan(utM)]
else:
    raise ValueError('Input MI matrix M must be either 2- or 3-dimensional!')

if type(fitCrop) != tuple:
    if fitCrop == 0:
        utM_crop = np.sort(utM)
    else:
        utM_crop = np.sort(utM)[round(fitCrop*len(utM)):-1*round(fitCrop*len(utM))]

if fitfn == 'norm':
    mle = norm.fit(utM_crop) # Fit lognormal distribution to data
    freeze = norm(*mle) # Freeze distribution with params from fit
    perc=freeze.ppf(percList) # Get MI at each percentile
elif fitfn == 'lognorm':
    mle = lognorm.fit(utM_crop) # Fit lognormal distribution to data
    freeze = lognorm(*mle) # Freeze distribution with params from fit
    perc=freeze.ppf(percList) # Get MI at each percentile
elif fitfn == 'gamma':
    mle = gamma.fit(utM_crop) # Fit lognormal distribution to data
    freeze = gamma(*mle) # Freeze distribution with params from fit
    perc=freeze.ppf(percList) # Get MI at each percentile
else:
    raise ValueError('Unsupported fit function requested.')

M_perc = []
n_perc = len(percList)

```

```

for i in range(n_perc):
    #M_perc.append(np.clip(M,ln_perc[i],np.max(M),min=0)) # Threshold M at each percentile i
in perc
    M_perc.append(M.copy())
    M_perc[-1][M_perc[-1] < perc[i]] = 0

print('Fit {} distribution with (shape, loc, scale) = {}'.format(fitfn,mle))
print('PPF {} percent confidence intervals at {}'.format([p*100 for p in percList],perc))

if plot:
    figA=plt.figure()
    axA = figA.add_subplot(1,1,1)
    n,bins,patches=axA.hist(utM,500,normed=True,alpha=0.5,histtype='stepfilled')
    x = np.linspace(utM.min(),utM.max(),num=10*len(utM))
    axA.plot(x,freeze.pdf(x),c='r',linewidth=3,alpha=0.5)
    if fitfn == 'norm':
        axA.annotate('crop = {: 0.3f}\nloc = {: 0.3f}\nscale = {:
0.3f}\n'.format(fitCrop,*mle),xy=(perc[1],.8*max(n)),horizontalalignment='left')
    elif fitfn == 'lognorm':
        axA.annotate('crop = {: 0.3f}\nshape = {: 0.3f}\nloc = {: 0.3f}\nscale = {:
0.3f}'.format(fitCrop,*mle),xy=(perc[1],.8*max(n)),horizontalalignment='left')
    axA.annotate('95% at {:
0.3f}'.format(perc[1]),xy=(perc[1],freeze.pdf(perc[1])+0.05*perc[1]),xytext=(perc[1],.6*max(n)),
arrowprops=dict(facecolor='black',shrink=0.05,width=1,frac=0.04,headwidth=8),horizontalalignment=
'left')

    axA.set_xlim((utM.min(),utM.max()))
    #axA.set_xlim((utM.min(),0.3))
    axA.set_title('MI distribution, {} fit'.format(fitfn))
    axA.set_xlabel('MI (bits)')
    axA.set_ylabel('Probability density')
    if outDir is not None:
        try:

plt.savefig(os.path.join(outDir,'{}_mi_histogram.eps'.format(runName)),transparent=True,format='e
ps')
        except:
            print('Failed to save MI histogram; will carry on anyways.')
    else:
        plt.show()
    figB,axB = plt.subplots(nrows=M_depth,ncols=len(perc),squeeze=False)
    vmax = np.nanmax(np.triu(M_perc[-1],1))
    for i,a in enumerate(axB.flatten()):
        j=i%n_perc # percentile index
        if dims == 2:
            p = a.matshow(M_perc[j],vmax=vmax,cmap=plt.cm.afmhot)
            a.set_xlabel('residue i')
            a.set_ylabel('residue j')
            #print('Plotting matrix {}'.format(j))
        elif dims == 3:
            if i == 0:
                k = 0
            elif j == 0:
                k += 1
            p = a.matshow(M_perc[j][:,:,k],vmax=vmax,cmap=plt.cm.afmhot)
            if j == 0:
                a.set_ylabel('Matrix {}\nresidue j'.format(k))
            if k == M_depth - 1:
                a.set_xlabel('residue i')
            #print('Plotting matrix {}, {}'.format(j,k))
        if i < n_perc:
            a.set_title('{}%: {: 0.3f}'.format(int(percList[i]*100),perc[i]))
    if outDir is not None:
        try:

plt.savefig(os.path.join(outDir,'{}_mi_thresholds.eps'.format(runName)),transparent=True,format='
eps')
        except:
            print('Failed to save thresholded MI matrix plots; will carry on anyways.')

```

```

    else:
        plt.show()
        plt.close()

    return(list(zip(percList,perc)),M_perc)

def matrix_eigendecomposition(M, R=None, preprocess='meansubtract', eval_perc=0.95,
    rotate_basis=False, resiliList=[], plot=False, runName='run', outDir=None, tailFitCrop=None,
    shuffle_iter=1000, add_back_mean=False):
    """
    * Description:
        Performs eigendecomposition on some symmetric, two-dimensional matrix `M`.

    * Input:
        * `M`
            A symmetric, two-dimensional `numpy.ndarray`.
        * `R`
            A two- or three-dimensional random analogue of `M`. R.shape()[:, :, 0] must equal
M.shape().
        * `preprocess`
            `{'meansubtract', 'skpnorm', 'skpss'}`
            A method for preprocessing `M` prior to decomposition:
            * `meansubtract`
                Subtract the mean of `M` from `M`.
            * `skpnorm`
                Use `sklearn.preprocessing.normalize` to preprocess the data.
            * `skpss`
                Use `sklearn.preprocessing.StandardScaler` to preprocess the data.
        * `eval_perc`
            Float percentile for determining significant eigenvalues; defaults to 0.95.
        * `rotate_basis`
            If true, use FastICA (`sklearn.decomposition.FastICA`) to more fully orthogonalize
significant eigenvectors.
        * `resiliList`
            A list of integers corresponding to the indexing of `M` with respect to some protein.
        * `plot`
            Boolean specifying whether or not to create various plots.
        * `runName`
            A descriptive title.
        * `tailFitCrop`
            Used to omit large eigenvalues from eigenspectrum prior to fitting lognormal
distribution.
            For example, if processing a 20 x 20 matrix, eigendecomposition will yield 20

```

eigenvalues.

If `tailFitCrop` is set to 15, a lognormal distribution will be fit to the eigenspectrum corresponding to the least significant 15 eigenvalues.

This is useful to improve the quality of the fit in cases where there are a few extremely significant eigenvalues that would otherwise skew the distribution.

```
* Output:
* `evecs_sort`
    A `numpy.ndarray` of eigenvectors sorted by eigenvalue with dimensions matching `M`.
* `evals_sort`
    A sorted list of eigenvalues.
* `sig_evecs_ln`
    A `numpy.ndarray` of eigenvectors sorted by eigenvalue with  $m \times m$  dimensions, where  $m$ 
is the rank of the input matrix and  $n$  is the number of significant eigenvalues identified.
* `sig_evecs_shuffle`
    A `numpy.ndarray` of eigenvectors sorted by eigenvalue with  $m \times m$  dimensions, where  $m$ 
is the rank of the input matrix and  $n$  is the number of significant eigenvalues identified.
* `E_sig`
    A three-dimensional `numpy.ndarray` matrix with dimensions  $m \times m \times m$ , where  $m$  is the
rank of the input matrix.
    Each slice in the third dimension corresponds to the reconstruction of the data with
an increasing number of "top" eigenvectors, from most to least significant.
    As such, E[:, :, -1] is expected to be equal to M.
* `E_insig`
    A three-dimensional `numpy.ndarray` matrix with dimensions  $m \times m \times m$ , where  $m$  is the
rank of the input matrix.
    Each slice in the third dimension corresponds to the reconstruction of the data with
an increasing number of "bottom" eigenvectors, from most to least significant.
    As such, E[:, :, 0] is expected to be equal to M.
...
# Setup
# if (plot is True or outDir is not None) and runName is None:
#     runName = 'run'

if not isinstance(M, np.ndarray):
    raise ValueError('M must be a numpy array.')
elif len(M.shape) != 2:
    raise ValueError('Input matrix M must be two-dimensional.')
M_mean = np.mean(M)
if preprocess == 'meansubtract':
    D = M - M_mean
elif preprocess == 'skpnorm':
    D = normalize(M)
elif preprocess == 'skpss':
    D = StandardScaler().fit_transform(M)
elif preprocess is None:
    print('Skipping preprocessing step.')
    D = M.copy()
else:
    raise ValueError('Invalid preprocessing mode specified.')

print('\nCalculating eigenvalues and eigenvectors for D...', end=' ')
try:
    evals, evecs = np.linalg.eigh(D)
    # u, s, v = np.linalg.svd(M.T)
except:
    print('failed!')
```

```

    raise
print('\nChecking eigenvectors...')
for ev in evecs:
    np.testing.assert_array_almost_equal(1.0,np.linalg.norm(ev))
print('Sorting eigenvectors by eigenvalues...')
evals_abs=np.abs(evals)
idx = evals_abs.argsort()[::-1] # Sort from greatest to least
evals_abs_sort=evals_abs[idx]
evals_sort = evals[idx]
evecs_sort = evecs[:,idx]

print('Fitting eigenvalue spectrum with scipy.stats.lognorm...')
if tailFitCrop is None:
    eval_ln = lognorm.fit(evals_abs_sort) # Fit lognormal distribution to data
else:
    eval_ln = lognorm.fit(evals_abs_sort[:-tailFitCrop])
ln_freeze = lognorm(*eval_ln) # Freeze distribution with params from fit
ln_perc = ln_freeze.ppf(eval_perc) # Get MI at each percentile
n_sig_evals_ln = sum(evals_abs_sort>ln_perc)
sig_evecs_ln = evecs_sort[:,n_sig_evals_ln].copy()
print('\nEigenvalues:\n{}\n\nEigenvectors:\n{}\n'.format(evals,evecs))
print('{} significant eigenvalue(s) greater than CDF cutoff of{: 0.2f} at
{}%.\n'.format(n_sig_evals_ln,ln_perc,int(eval_perc*100)))

if R is not None:
    print('Generating random matrix model based on provided data...',end=' ')
    if len(R.shape) == 2:
        n_rand_mats = 1
    elif len(R.shape) == 3:
        n_rand_mats = R.shape[2]
    print('Found {} random matrices.'.format(n_rand_mats))
    if n_rand_mats == 1:
        ...
    else:
        rand_evals_mat = np.zeros((R.shape[0], n_rand_mats))
        for m_idx in range(n_rand_mats):
            rand_evals,rand_evecs = np.linalg.eigh(R[:, :,m_idx])
            rand_evals_abs = np.abs(rand_evals)
            idx = rand_evals_abs.argsort()[::-1]
            rand_evals_abs_sort = rand_evals_abs[idx]
            rand_evals_sort = rand_evals[idx]
            rand_evals_mat[:,m_idx] = rand_evals_sort
        #plt.matshow(rand_evals_mat, cmap=plt.cm.afmhot)
        #plt.show()
        all_evals = np.zeros((rand_evals_mat.shape[0], n_rand_mats+1))
        all_evals[:,1:n_rand_mats+1] = rand_evals_mat - rand_evals_mat.mean()
        all_evals[:,0] = evals_sort - evals_sort.mean()
    print('Comparing to eigendecompositions of input matrix shuffled {}
times...'.format(shuffle_iter), end=' ')
    shuffle_evals = np.zeros((D.shape[0],shuffle_iter))
    triu_nodiag = np.triu_indices_from(D,1)
    tril_nodiag = np.tril_indices_from(D,-1)
    S = D.copy()
    for i in range(shuffle_iter): # Shuffle matrix S (preserving diagonal) shuffle_iter times and
calculate eigvals for each
        s = S[triu_nodiag]
        np.random.shuffle(s)
        S[triu_nodiag] = s
        S[tril_nodiag] = S.T[tril_nodiag]
        shuffle_evals[:,i], _ = np.linalg.eigh(S)
        shuffle_evals[:,i] = np.sort(np.abs(shuffle_evals[:,i]))[::-1]
    # shuffle_cutoff = shuffle_evals.max()
    # n_sig_evals_shuffle = np.sum(evals_abs_sort>shuffle_cutoff)
    n_sig_evals_shuffle = 0
    for i,comp in enumerate(evals_abs_sort > shuffle_evals.max(axis=1)):
        if comp == True:
            n_sig_evals_shuffle += 1
        else:
            shuffle_cutoff = shuffle_evals.max(axis=1)[i]

```

```

        break
    sig_evecs_shuffle = evecs_sort[:, :n_sig_evals_shuffle].copy()
    print('Found {0} significant eigenvalues greater than shuffle cutoff of {1:.2f}; will assume
{0} modes throughout remaining analysis.\n'.format(n_sig_evals_shuffle, shuffle_cutoff))

    if rotate_basis and n_sig_evals_shuffle > 1:
        print('Rotating significant eigenvectors (based on shuffle cutoff) with ICA...')
        ica=FastICA(n_components=n_sig_evals_shuffle,max_iter=100000)
        sig_evecs_shuffle=ica.fit(sig_evecs_shuffle).transform(sig_evecs_shuffle)

    print('Reconstructing data...')
    nEle = D.shape[0]
    E_sig = np.zeros([nEle,nEle,nEle])
    E_insig = np.zeros([nEle,nEle,nEle])
    for i in range(nEle):
        sig_evecs_temp = np.zeros_like(evecs_sort)
        sig_evecs_temp[:, :i+1]=evecs_sort[:, :i+1]
        sig_evals_temp=np.zeros_like(evals_sort)
        sig_evals_temp[i+1] = evals_sort[i+1]
        sig_evals_temp=np.diag(sig_evals_temp)
        E_sig[:, :, i] = np.dot(sig_evecs_temp,np.dot(sig_evals_temp,sig_evecs_temp.T))
        insig_evecs_temp = np.zeros_like(evecs_sort)
        insig_evecs_temp[:, i+1:]=evecs_sort[:, i+1:]
        insig_evals_temp=np.zeros_like(evals_sort)
        insig_evals_temp[i+1:] = evals_sort[i+1:]
        insig_evals_temp=np.diag(insig_evals_temp)
        E_insig[:, :, i] = np.dot(insig_evecs_temp,np.dot(insig_evals_temp,insig_evecs_temp.T))
        if add_back_mean:
            E_sig[:, :, i] += M_mean
            E_insig[:, :, i] += M_mean
    if add_back_mean:
        D += M_mean

    if plot:
        print('Plotting...')
        if rotate_basis:
            figC,axC=plt.subplots(1,2,figsize=(20,30))
            bounds = cmap_bounds(sig_evecs_shuffle)
            axC[0].matshow(sig_evecs_shuffle,cmap=plt.cm.RdBu_r,vmin=bounds[0],vmax=bounds[1])
            axC[0].set_title('Significant eigenvectors')
            axC[1].matshow(sig_evecs_shuffle,cmap=plt.cm.RdBu_r)
            axC[1].set_title('Significant eigenvectors, ICA transform')
            for i in (0,1):
                axC[i].set_xlabel('Eigenvector')
                axC[i].set_ylabel('Residue')
            if resiliList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axC[0].get_yticks()[::-1]]
                k = [resiliList[n] for n in j]
                axC[0].set_yticklabels(k)
                axC[1].set_yticklabels(k)
            if outDir is not None:
                try:
                    plt.savefig(os.path.join(outDir,runName+'_rotate-
basis.eps'),transparent=True,format='eps')
                    plt.close()
                except:
                    print('Failed to save figure!')
            else:
                plt.show()
        if R is not None:
            figD,axD=plt.subplots(1,2,figsize=(25,15))
            q=axD[0].matshow(all_evals, cmap=plt.cm.afmhot, vmin=np.percentile(all_evals, 2),
vmax=np.percentile(all_evals,98))
            if resiliList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axD[0].get_yticks()[::-1]]
                k = [resiliList[n] for n in j]
                axD[0].set_yticklabels(k)
            divD = make_axes_locatable(axD[0])
            caxD = divD.append_axes('right',size='5%',pad=0.05)

```

```

cbarD = plt.colorbar(q,cax=caxD,label='Eigenvalue magnitude')
axD[0].set_xlabel('Instance\nUnshuffled (idx 0), shuffled (idx 1)')
axD[0].set_ylabel('Position')
axD[0].set_title('All eigenvalues, sorted')
axD[1].hist(all_evals,color=['r']+n_rand_mats*['k'],alpha=.8)
axD[1].grid()
axD[1].set_xlabel('Mean-centered MI')
axD[1].set_ylabel('Counts')
axD[1].set_title('Distributions, unshuffled (red) and shuffled (black)')
if outDir is not None:
    try:
        plt.savefig(os.path.join(outDir,runName+'_vs-random-matrix-
model.eps'),transparent=True,format='eps')
        plt.close()
    except:
        print('Failed to save figure!')
else:
    plt.show()

figA,axA=plt.subplots(1,1,figsize=(1,1))
x = np.linspace(0,evals_abs.max(),num=10*len(evals_abs))
axA.plot(x,ln_freeze.pdf(x),c='r',linewidth=3,alpha=0.5)

n,b,p=axA.hist(evals_abs,bins=int(len(evals_abs)),alpha=1,edgecolor=None,normed=True,color='k')
axA.annotate('Lognormal ECDF cutoff,\n{:.0%} at
{:.2f}'.format(eval_perc,ln_perc),xy=(ln_perc,ln_freeze.pdf(ln_perc)+0.6*ln_freeze.pdf(ln_perc))
,xytext=(ln_perc,.8*max(n)),
arrowprops=dict(facecolor='black',shrink=0.05,width=1,frac=0.04,headwidth=8),horizontalalignment=
'left')
axA.annotate('Shuffle cutoff\nat
{:.2f}'.format(shuffle_cutoff),xy=(shuffle_cutoff,ln_freeze.pdf(shuffle_cutoff)+0.2*ln_freeze.pdf
(shuffle_cutoff)),xytext=(shuffle_cutoff,.6*max(n)),
arrowprops=dict(facecolor='black',shrink=0.05,width=1,frac=0.04,headwidth=8),horizontalalignment=
'left')
axA.set_title('Eigenvalue distribution, lognormal fit')
axA.set_xlabel('abs(eigenvalue)')
axA.set_ylabel('Probability density')
axA.grid()
#axA.set_ylim((0,1))
if outDir is not None:
    try:
        plt.savefig(os.path.join(outDir,runName+'_decomp-
hist.eps'),transparent=True,format='eps')
        plt.close()
    except:
        print('Failed to save figure!')
else:
    plt.show()

figF, axF = plt.subplots(1,1,figsize=(20,20))
bounds = cmap_bounds(evecs_sort)
p = axF.matshow(evecs_sort,cmap=plt.cm.RdBu_r,vmin=bounds[0],vmax=bounds[1])
axF.set_xlabel('Eigenvector index\n(eigenvalue)')
axF.set_ylabel('Residue')
axF.set_title('Eigenvectors sorted by absolute value of eigenvalues\n')
j = [int(n) for n in axF.get_yticks()[:-1]]
k = ['{}\n{: 0.1f}'.format(n,evals_sort[n]) for n in j]
axF.set_xticklabels(k)
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axF.get_xticks()[:-1]]
    k = [resiList[n] for n in j]
    axF.set_yticklabels(k)
div = make_axes_locatable(axF)
cax = div.append_axes('right',size='5%',pad=0.05)
cbar = plt.colorbar(p,cax=cax,label='Eigenvector magnitude')
if outDir is not None:
    try:
        plt.savefig(os.path.join(outDir,runName+'_decomp-vec-
mat.eps'),transparent=True,format='eps')

```

```

        plt.close()
    except:
        print('Failed to save figure!')
else:
    plt.show()

    mat_plot_tup = ((D, 'Input'), (E_sig[:, :, -1], 'Full reconstruction'),
(E_sig[:, :, n_sig_evals_ln-1], 'E_sig_lognorm[:, :, {}]'.format(n_sig_evals_ln-1)),
(E_insig[:, :, n_sig_evals_ln-1], 'E_insig_lognorm[:, :, {}]'.format(n_sig_evals_ln-1)),
(E_sig[:, :, n_sig_evals_shuffle-1], 'E_sig_shuffle[:, :, {}]'.format(n_sig_evals_shuffle-1)),
(E_insig[:, :, n_sig_evals_shuffle-1], 'E_insig_shuffle[:, :, {}]'.format(n_sig_evals_shuffle-1)))
    n_rows = sum(divmod(len(mat_plot_tup), 2))
    figBDict = {}
    figBDict['fig'], figBDict['ax'] =
plt.subplots(n_rows, 2, figsize=(15, len(mat_plot_tup)*3), squeeze=True)
    if add_back_mean:
        cmap = plt.cm.afmhot
        #bounds = (np.percentile(np.triu(D, 5), 1), np.percentile(np.triu(D, 5), 99))
        bounds = (D.min(), D.max())
    else:
        cmap = plt.cm.RdBu_r
        bounds = cmap_bounds(np.triu(D, 5))
    for i, mat in enumerate(mat_plot_tup):
        p = figBDict['ax'][i//2][i%2].matshow(mat[0], cmap=cmap, vmin=bounds[0], vmax=bounds[1])
        figBDict['ax'][i//2][i%2].set_xlabel('Residue i')
        figBDict['ax'][i//2][i%2].set_ylabel('Residue j')
        if resiliList != []: # Relabel matrix axes with proper residue indices
            j = [int(n) for n in figBDict['ax'][i//2][i%2].get_xticks()[:-1]]
            k = [resiliList[n] for n in j]
            [figBDict['ax'][i//2][i%2].set_xticklabels(k) for i in (0, 1, 2)]
            [figBDict['ax'][i//2][i%2].set_yticklabels(k) for i in (0, 1, 2)]
            figBDict['ax'][i//2][i%2].set_title(mat[1])
            figBDict['div{}'.format(i)] = make_axes_locatable(figBDict['ax'][i//2][i%2])
            figBDict['cax{}'.format(i)] =
figBDict['div{}'.format(i)].append_axes('right', size='5%', pad=0.05)
            figBDict['cbar{}'.format(i)] =
plt.colorbar(p, cax=figBDict['cax{}'.format(i)], label='Units from mean')
            figBDict['fig'].suptitle('{}: Data reconstruction\nLognormal cutoff, {:.0%} at
{:.0.2f}\nShuffle cutoff, {:.0.2f}'.format(runName, eval_perc, ln_perc, shuffle_cutoff), fontsize=20)

    if outDir is not None:
        try:
plt.savefig(os.path.join(outDir, runName+'_components.eps'), transparent=True, format='eps')
            plt.close()
        except:
            print('Failed to save figure!')
    else:
        plt.show()
        figE, axE = plt.subplots(1, 3, figsize=(25, 15)) # ax[0] shows matrix of random eigenvalues
over shuffle_iter repetitions, ax[1] shows plot version compared to unshuffled matrix eigvals,
ax[2] shows histogram version of ax[1]

dat_bounds = (np.concatenate((shuffle_evals.flatten(), evals_abs_sort.flatten())).min(), np.concatena
te((shuffle_evals.flatten(), evals_abs_sort.flatten())).max())
    if shuffle_iter <= 100:
        q = axE[0].matshow(shuffle_evals, cmap=plt.cm.afmhot)
        axE[0].set_xlabel('Shuffle index')
    else:
        q = axE[0].matshow(shuffle_evals[:, :100], cmap=plt.cm.afmhot)
        axE[0].set_xlabel('Shuffle index, truncated to first 100 instances')
    axE[0].set_ylabel('Eigenvalue index')
    if resiliList != []: # Relabel matrix axes with proper residue indices
        j = [int(n) for n in axE[0].get_yticks()[:-1]]
        k = [resiliList[n] for n in j]
        axE[0].set_yticklabels(k)
    div = make_axes_locatable(axE[0])
    cax = div.append_axes('right', size='5%', pad=0.05)

```

```

cbar = plt.colorbar(q,cax=cax,label='Absolute value of eigenvalue magnitude')
# Lines
[axE[1].plot(list(range(D.shape[0])), shuffle_evals[:,i],'k-',alpha=0.05) for i in
range(shuffle_iter)]
axE[1].plot(list(range(D.shape[0])),evals_abs_sort,'r.-')
axE[1].set_xlim((-1,D.shape[0]))
axE[1].set_ylim((dat_bounds[0]-1, dat_bounds[1]+1))
axE[1].set_xlabel('Eigenvalue index')
axE[1].set_ylabel('Absolute value of eigenvalue magnitude')
axE[1].grid()

bin_bounds=np.linspace(int(dat_bounds[0]-
abs(0.2*dat_bounds[0])),int(dat_bounds[1]+0.2*abs(dat_bounds[1])),int(D.shape[0]/5))

axE[2].hist(shuffle_evals.flatten(),bins=bin_bounds,color='k',normed=True,label='shuffled')

axE[2].hist(evals_abs_sort.flatten(),bins=bin_bounds,color='r',normed=True,alpha=0.8,label='unshuffled')
axE[2].set_xlabel('Absolute value of eigenvalue magnitude')
axE[2].set_ylabel('Probability density')
axE[2].grid()
figE.suptitle('{}\nEigenvalue analysis, unshuffled matrix (red) vs. {} shuffled matrices
(black)'.format(runName,shuffle_iter),fontsize=20)
if outDir is not None:
    try:
        plt.savefig(os.path.join(outDir,runName+'_vs-shuffled-data-
matrix.eps'),transparent=True,format='eps')
        plt.close()
    except:
        print('Failed to save figure!')
else:
    plt.show()

figG,axG=plt.subplots(1,1,figsize=(10,10))

axG.hist([evals_abs_sort.flatten(),shuffle_evals.flatten()],bins=bin_bounds,color=['k','r'],edgec
olor=None,normed=True,alpha=0.6, label=['input','shuffled'])
axG.annotate('Shuffle cutoff\nat
{:0.2f}'.format(shuffle_cutoff),xy=(shuffle_cutoff,ln_freeze.pdf(shuffle_cutoff)+0.2*ln_freeze.pd
f(shuffle_cutoff)),xytext=(shuffle_cutoff,.6*max(n)),
arrowprops=dict(facecolor='black',shrink=0.05,width=1,frac=0.04,headwidth=8),horizontalalignment=
'left')
axG.set_xlabel('Absolute value of eigenvalue magnitude')
axG.set_ylabel('Probability')
#axG.grid()
figG.suptitle('{}\nEigenvalue analysis, unshuffled matrix (black) vs. {} shuffled
matrices (red)'.format(runName,shuffle_iter),fontsize=20)
if outDir is not None:
    try:
        plt.savefig(os.path.join(outDir,runName+'_vs-shuffled-
hist.eps'),transparent=True,format='eps')
        plt.close()
    except:
        print('Failed to save figure!')
else:
    plt.show()

return(evecs_sort, evals_sort, n_sig_evals_ln, sig_evecs_ln, n_sig_evals_shuffle,
sig_evecs_shuffle, E_sig, E_insig)

def
matrix_infomapping(M,threshold=0,binarize=False,resiList=[],diag_suppress=None,infomap_iterations
=1000,plot=False,outDir=None):
'''
* Description:

Perform community detection on thresholded input matrix `M` using `infomap`.
This requires the compiled `infomap` executable in the user's path and a customized

```

version of the `infomap.py` script distributed with it.
For more information, see <http://www.mapequation.org/code.html>.

```
* Input:
* `M`
    A symmetric, two-dimensional numpy.ndarray.
* `threshold`
    A float or list of floats greater than the minimum and less than the maximum of M.
    For some threshold t in threshold, infomap will be run on the matrix M[M<t] =
0`.
* `binarize`
    A boolean; when true, infomap is run for a given threshold t on binarized M,
i.e., M[M<t] = 0` and M[M>t] = 1`.
* `resilist`
    A list of integers corresponding to the indexing of M with respect to some protein.
* `diag_supress`
    If None, diagonal of M will not be suppressed prior to running infomap.
    If an integer, elements out to the diag_supressth diagonal will be zeroed.
* `infomap_iterations`
    Integer number of iterations that infomap is allowed to run.
* `plot`
    Boolean specifying whether or not to create various plots.
* `outDir`
    If provided, infomap stuff and plots will be output to this directory.
    If None, infomap stuff will be dumped into current working directory.
* Output, len(threshold) == 1`:
* `GDict`
    Dictionary with networkx graph objects created from infomap objects keyed by
floats from threshold.
* `treeDict`
    Dictionary with infomap.HierarchicalNetwork objects keyed by floats from
threshold.
* Output, len(threshold) > 1`:
    As above, but with additional output:
* `thresholdingStats`
    List of tuples, where each tuple corresponds to an infomap run with elements
(threshold, codeLength, number_of_communities).
'''
# Some code taken from example-networkx.py from Infomap distribution!
# Check inputs
if type(threshold) != list:
    threshold=[threshold]
if len(M.shape) > 2 or M.shape[0]!=M.shape[1]:
    raise ValueError('Supplied matrix must be two dimensional and square!')
```

```

if resiList != []:
    if type(resiList) == list:
        if sum([type(i) == int for i in resiList]) < len(resiList):
            raise ValueError('resiList must contain integers!')
        else:
            raise ValueError('resiList must be a list!')
    if resiList != [] and len(resiList) != M.shape[0]:
        raise ValueError('Length of supplied residue index list doesn\'t match length of input
matrix!')
    if binarize is True and min(threshold) < M.min():
        raise ValueError('Binarization is impossible with threshold less than the minimum of
input matrix M.')

if diag_suppress is not None and type(diag_suppress) == int:
    try:
        N = np.triu(M,diag_suppress) + np.tril(M,-diag_suppress)
    except:
        print('Failed to zero diagonal(s), check diag_crop and try again!')
        raise
else:
    N=M.copy()
    np.fill_diagonal(N,0)

# Create and process network
GDict={}
treeDict={}
for t in threshold:
    G = nx.from_numpy_matrix(N) # Create a networkx graph from the supplied numpy matrix
    if resiList != []:
        for n in G.nodes_iter():
            G.node[n]['residue'] = resiList[n]
    print('Initializing infomap...\n')
    conf = infomap.init('--code-rate 1 -uz -N{}'.format(infomap.iterations))
    net = infomap.Network(conf)
    tree = infomap.HierarchicalNetwork(conf)
    print('\nBuilding network from matrix...')
    edge_remove = []
    for i,j in G.edges_iter(): # Iterate over the graph
        w = G[i][j]['weight']
        if w>t:
            if binarize:
                net.addLink(int(i),int(j)) # Add unweighted edge between i and j
            else:
                net.addLink(int(i),int(j),float(w)) # Add edge between i and j with weight w
        else:
            edge_remove.append((i,j))
    [G.remove_edge(*e) for e in edge_remove] # Delete edges with weights below threshold
    print('Finalizing and clustering network with infomap...')
    net.finalizeAndCheckNetwork(True,nx.number_of_nodes(G))
    infomap.run(net,tree)
    print('Mapping to networkx graph...')
    communities = {}
    node_depth = {}
    for leaf in tree.leafIter():
        communities[leaf.originalLeafIndex] = leaf.parentNode.parentIndex
        node_depth[leaf.originalLeafIndex] = leaf.depth()
    nx.set_node_attributes(G,'community',communities)
    nx.set_node_attributes(G,'depth',node_depth)
    G.graph['n_communities'] = tree.numTopModules()
    G.graph['codelength'] = tree.codelength()
    G.graph['max_depth'] = tree.maxDepth()
    G.graph['binarized'] = False
    G.graph['threshold'] = threshold
    G.graph['infomap_iterations'] = infomap.iterations
    GDict[t] = G
    treeDict[t] = tree
if outDir is not None:
    tree.writeHumanReadableTree(os.path.join(outDir,'network_{}.tree'.format(t)))
    tree.writeMap(os.path.join(outDir,'network_{}.map'.format(t)))

```

```

    if plot:
        print('Drawing graph...')
        plt.figure()
        cmapDark = colors.ListedColormap(['#1f78b4', '#33a02c', '#e31a1c', '#ff7f00',
'#6a3d9a'], 'indexed', G.graph['n_communities'])
        cmapLight = colors.ListedColormap(['#a6cee3', '#b2df8a', '#fb9a99', '#fdbf6f',
'#cab2d6'], 'indexed', G.graph['n_communities'])
        comms=[v for k,v in list(nx.get_node_attributes(G, 'community').items())]
        pos=nx.spring_layout(G)
        nx.draw_networkx_edges(G,pos)
        if resiliList == []:
            nodeCollection =
nx.draw_networkx_nodes(G,pos=pos,node_color=comms,cmap=cmapLight)
        else:
            nodeCollection =
nx.draw_networkx_nodes(G,pos,with_labels=False,node_size=600,alpha=0.7,node_color=comms,cmap=cmap
Light)
            node_labels=nx.get_node_attributes(G,'residue')
            nx.draw_networkx_labels(G,pos,labels = node_labels)
            darkColors=[cmapDark(v) for v in comms]
            nodeCollection.set_edgcolor(darkColors)
            if len(threshold) < 10:
                plt.show()
                #plt.close()
            if outDir is not None:
                print('Saving graph...')

plt.savefig(os.path.join(outDir,'graph_{}.eps'.format(t)),transparent=True,format='eps')
    if G.graph['n_communities'] > 1:
        cmap = plt.cm.afmhot
        cmap.set_bad('k',.3)
        figA,axA=plt.subplots(1,G.graph['n_communities'],figsize=(45,15))
        comms = np.array(list(nx.get_node_attributes(G,'community').values()))
        for c in range(G.graph['n_communities']):
            idx = (comms != c)
            m=N.copy()
            m[idx,:]= np.nan
            m[:,idx]= np.nan
            #m[m<threshold] = 0
            axA[c].matshow(m,cmap=cmap)
            axA[c].set_xlabel('residue i')
            axA[c].set_ylabel('residue j')
            axA[c].set_title('Community {}'.format(c))
            if resiliList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axA[c].get_xticks()[:-1]]
                x = [resiliList[n] for n in j]
                axA[c].set_xticklabels(x)
                axA[c].set_yticklabels(x)
            if outDir is not None:
                print('Saving clustered matrix figure...')

plt.savefig(os.path.join(outDir,'matrix_clusters_{}.eps'.format(t)),transparent=True,format='eps'
)

    if len(threshold) > 10 and plot:
        plt.close('all')

    if len(threshold) > 1:
        Gkeys = list(GDict.keys())
        Gkeys.sort()
        traj_codelength = [GDict[Gkeys[i]].graph['codelength'] for i in range(len(Gkeys))]
        traj_ncomms = [GDict[Gkeys[i]].graph['n_communities'] for i in range(len(Gkeys))]
        if plot:
            figB,axB=plt.subplots(1,2,figsize=(15,10))
            axB[0].scatter(Gkeys,traj_codelength,alpha=0.5,edgecolor='none')
            axB[0].set_xlabel('Threshold (bits)')
            axB[0].set_ylabel('Codelength (bits)')
            axB[0].grid(True)
            axB[1].scatter(Gkeys,traj_ncomms,alpha=0.5,edgecolor='none')
            axB[1].set_ylim((0,max(traj_ncomms)+1))

```

```

    axB[1].set_xlabel('Threshold (bits)')
    axB[1].set_ylabel('Number of communities')
    axB[1].set_xlim((0,max(traj_ncomms)+1))
    axB[1].grid(True)
    if outDir is not None:
        print('Saving threshold analysis plots...')

plt.savefig(os.path.join(outDir,'thresholding_{}.eps'.format(t)),transparent=True,format='eps',op
timize=False)
    return(GDict,treeDict,list(zip(Gkeys,traj_codelength,traj_ncomms)))
else:
    return(GDict,treeDict)
print('Done!')

def
nonnegative_matrix_factorization(M,n_components=3,optimize=False,resiList=[],diag_suppress=None,m
ode='symnmf-nndsvd',runName='run',plot=False,outDir=None):
    """
    * Description:

        Perform non-negative matrix factorization on input `M` for rank `n_components` using
        `sklearn.decomposition.ProjectiveGradientNMF`.
        This implementation of the algorithm (in `sklearn`) is not constrained for symmetric
        matrices, so the `SymNMF` approach should probably be used instead.

    * Input:

    * `M`

        A two-dimensional `numpy.ndarray` object.

    * `n_components`

        Rank of factorization to perform.

    * `optimize`

        If `True`, will perform NMF up to rank equal to that of the input matrix and report
        the reconstruction error at each step.
        Note that there is no actual optimization at this step.

    * `resiList`

        A list of integers corresponding to the indexing of `M` with respect to some protein.

    * `diag_suppress`

        If `None`, diagonal of `M` will not be suppressed prior to running `infomap`.
        If an integer, elements out to the `diag_suppress`th diagonal will be zeroed.

    * `mode`

        `{'symnmf', 'symnmf-nndsvd', 'pgnmf-random', 'pgnmf-nndsvd'}`

        The method for performing NMF.

    * `symnmf`

        Perform symmetric non-negative matrix factorization with `SymNMF.py`.

    * `symnmf-nndsvd`

        Perform symmetric non-negative matrix factorization with `SymNMF.py`.
        Initialize protocol with matrix calculated with NNDSVD.

    * `symnmf-nndsvida`

        Perform symmetric non-negative matrix factorization with `SymNMF.py`.
        Initialize protocol with matrix calculated with NNDSVD; zeros will be replaced

```

with a small number.

```
*  `pgnmf-random``
    Perform projected gradient non-negative matrix factorization using
`sklearn.decomposition.ProjectedGradientNMF`.
    Initialize with random matrix.

*  `pgnmf-nndsvd``
    Perform projected gradient non-negative matrix factorization using
`sklearn.decomposition.ProjectedGradientNMF`.
    Initialize using SVD.

*  `runName`
    A descriptive title.

*  `plot`
    Boolean specifying whether or not to create various plots.

*  `outDir`
    If provided, `infomap` plots will be output to this directory.

*  Output:

*  `W`
    Components, with dimensions n x k.

*  `H`
    Components, with dimensions k x n.
...
def get_optimum_rank(obj,diff_perc = 0.05):
    """
    """
    def exp_decay(x, a, b, c):
        """
        y=e^
        """
        return(a * np.exp(-b * x) + c)

    rank_range = np.arange(1,len(obj)+1)
    obj = np.array(obj)
    opt,cov = curve_fit(exp_decay,rank_range,obj)
    fit = [exp_decay(x,*opt) for x in rank_range]
    for i,val in enumerate(obj):
        rank_estimate = i + 1
        if np.abs((val - opt[-1])) / opt[-1] < diff_perc:
            break
    print('Estimated rank based on exponential decay model (5%) is
    {}. '.format(rank_estimate))
    return(rank_range,fit,rank_estimate)

def randomize_matrix(M):
    """
    Returns a symmetric, randomized copy of M with diagonal preserved.
    """
    M_temp = M.copy()
    vec = M_temp[np.triu_indices_from(M_temp)]
    np.random.shuffle(vec)
    M_temp[np.triu_indices_from(M_temp)] = vec
    M_random = np.triu(M_temp,1) + (np.eye(M_temp.shape[0]) * np.diag(M_temp)) +
    np.tril(M_temp.T,-1) # Keep diag equal to mean of matrix
    plt.matshow(M_random)
```

```

    return(M_random)

if len(M.shape) != 2 or M.shape[0] != M.shape[1]:
    raise ValueError('Input matrix must be two-dimensional.')
n_components = int(n_components)
if n_components > M.shape[0] or n_components < 1:
    raise ValueError('n_components must be at least 1 and no greater than the rank of the
input matrix.')
if diag_suppress is not None and type(diag_suppress) == int:
    try:
        N = np.triu(M,diag_suppress) + np.tril(M,-diag_suppress)
    except:
        print('Failed to zero diagonal(s), check diag_crop and try again!')
        raise
    # print(np.sum(N==0))
    # N[N==0]=np.mean(M)
else:
    N=M.copy()
mode = mode.split('-')
if mode[0] == 'symnmf':
    print('Performing NMF with SymNMF.')
    if optimize:
        print('Attempting to optimize rank...')
        #ncpu = mp.cpu_count()
        #if ncpu > 1:
        #    ncpu -= 1
        #pool = mp.Pool(processes=ncpu)
        #results = [pool.apply_async(symnmf_newton,args=(A, k)) for A, k in
zip(it.repeat(N,N.shape[0] - 1), list(range(1, N.shape[0])))])
        #opt_list = [p.get() for p in results]
        #pool.close()
        #pool.join()
        N_random = randomize_matrix(N)
        obj=[]
        try:
            [obj.append((symnmf_newton(N,i,max_iter=10000)[2],
symnmf_newton(N_random,i,max_iter=10000)[2])) for i in range(1,2*int(N.shape[0]/3))] # Run out to
2/3 of maximum possible rank
        except RuntimeError:
            print('Failed')
            rank_range,fit,rank_estimate = get_optimum_rank([o[0] for o in obj])
            _,fit_random,rank_estimate_random = get_optimum_rank([o[1] for o in obj])
        if plot:
            figA,axA=plt.subplots(1,1,figsize=(15,15))
            axA.plot(np.arange(1,len(obj)+1),[o[0] for o in
obj],marker='o',color='k',label='observed')
            axA.plot(np.arange(1,len(obj)+1),[o[1] for o in
obj],marker='o',color='b',label='random')
            axA.plot(rank_range,fit,color='r',label='fit, observed')
            axA.plot(rank_range,fit_random,color='m',label='fit, random')
            axA.set_xlabel('Factorization rank')
            axA.set_ylabel('Reconstruction error')
            axA.grid()
            axA.legend()
            n_components = rank_estimate
        if len(mode) > 1:
            print('Initializing protocol with method {}'.format(mode[1]))
            Hinit = mode[1]
        else:
            mode.append('random')
            Hinit = None
        H_start,H,i,obj = symnmf_newton(N,n_components,Hinit = Hinit, max_iter = 1000000, tol =
0.0001, sigma = 0.1, beta = 0.1, debug_lvl = 0, rand_mat = None)
        print('SymNMF converged after {} cycles. Frobenius norm of matrix difference between the
training data and reconstructed data from fit produced by model: {}'.format(i,obj))
        if plot:
            figA,axA=plt.subplots(1,5,figsize=(50,20))
            figA.suptitle('Symmetric non-negative matrix factorization of {}, rank {} (mode {}-
{} )'.format(runName,n_components,mode[0],mode[1] ),fontsize=20)

```

```

vmin = np.percentile(N,1)
vmax = np.percentile(N,99)
imC=axA[0].matshow(N, cmap=plt.cm.afmhot, vmin=M.min(), vmax=M.max())
axA[0].set_title('Input matrix A')
axA[0].set_ylabel('Residue i')
axA[0].set_xlabel('Residue j')
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axA[0].get_yticks()[:-1]]
    k = [resiList[n] for n in j]
    axA[0].set_xticklabels(k)
    axA[0].set_yticklabels(k)
divC = make_axes_locatable(axA[0])
caxC = divC.append_axes('right',size='10%',pad=0.05)
cbarC = plt.colorbar(imC,cax=caxC)
imD=axA[1].matshow(H_start,cmap=plt.cm.afmhot)
axA[1].set_title('Initialization matrix H_init, rank {}'.format(n_components))
axA[1].set_ylabel('Position')
axA[1].set_xlabel('Component')
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axA[1].get_yticks()[:-1]]
    k = [resiList[n] for n in j]
    axA[1].set_yticklabels(k)
divD = make_axes_locatable(axA[1])
caxD = divD.append_axes('right',size='50%',pad=0.05)
cbarD = plt.colorbar(imD,cax=caxD)
imA=axA[2].matshow(H,cmap=plt.cm.afmhot)
axA[2].set_title('Factorization matrix H, rank {}'.format(n_components))
axA[2].set_ylabel('Position')
axA[2].set_xlabel('Component')
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axA[2].get_yticks()[:-1]]
    k = [resiList[n] for n in j]
    axA[2].set_yticklabels(k)
divA = make_axes_locatable(axA[2])
caxA = divA.append_axes('right',size='50%',pad=0.05)
cbarA = plt.colorbar(imA,cax=caxA)
imB=axA[3].matshow(np.dot(H,H.T),cmap=plt.cm.afmhot, vmin=M.min(), vmax=M.max())
axA[3].set_title('dot(H,H.T)')
axA[3].set_ylabel('Residue i')
axA[3].set_xlabel('Residue j')
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axA[3].get_yticks()[:-1]]
    k = [resiList[n] for n in j]
    axA[3].set_xticklabels(k)
    axA[3].set_yticklabels(k)
divB = make_axes_locatable(axA[3])
caxB = divB.append_axes('right',size='10%',pad=0.05)
cbarB = plt.colorbar(imB,cax=caxB)
imE=axA[4].matshow(M-np.dot(H,H.T),cmap=plt.cm.afmhot)
axA[4].set_title('M-dot(H,H.T)')
axA[4].set_ylabel('Residue i')
axA[4].set_xlabel('Residue j')
if resiList != []: # Relabel matrix axes with proper residue indices
    j = [int(n) for n in axA[4].get_yticks()[:-1]]
    k = [resiList[n] for n in j]
    axA[4].set_xticklabels(k)
    axA[4].set_yticklabels(k)
divE = make_axes_locatable(axA[4])
caxE = divE.append_axes('right',size='10%',pad=0.05)
cbarE = plt.colorbar(imE,cax=caxE)
if outDir is not None:
    try:
plt.savefig(os.path.join(outDir,runName+'_symnmf_{}_rank{}.eps'.format(mode[1],n_components)),tra
nsparent=True,format='eps')
    except:
        print('Failed to save figure!')
    else:
        plt.show()

```

```

plt.close()
return(H,i,obj)

elif mode[0] == 'pgnmf' and mode[1] in ('random','nndsvd'):
    init=mode[1]
    print('Performing NMF with ProjectedGradientNMF with {} initialization.'.format(init))
    if optimize:
        print('Attempting to optimize rank...')
        opt_list=[]
        for i in range(1,N.shape[0]):
            model = ProjectedGradientNMF(n_components=i, init=init,
random_state=0,max_iter=10000,nls_max_iter=100000)
            fit = model.fit(N)
            opt_list.append(fit.reconstruction_err_)
        if plot:
            figA,axA=plt.subplots(1,1,figsize=(15,15))
            axA.plot(opt_list,marker='o')
            axA.set_xlabel('Factorization rank')
            axA.set_ylabel('Reconstruction error')
        return
        model = ProjectedGradientNMF(n_components=n_components, init=init,
random_state=0,max_iter=10000,nls_max_iter=100000)
        fit = model.fit(N)
        W = model.fit_transform(N)
        H = model.components_
        print('Frobenius norm of matrix difference between the training data and reconstructed
data from fit produced by model: {}'.format(fit.reconstruction_err_))
        if plot:
            figA,axA=plt.subplots(1,2,figsize=(20,30))
            figA.suptitle('Non-negative matrix factorization of {}, rank {}, {} initialization
procedure'.format(runName,n_components,init),fontsize=20)
            imA=axA[0].matshow(W,cmap=plt.cm.afmhot)
            axA[0].set_title('Factorization matrix W'.format(n_components,init))
            axA[0].set_ylabel('Position')
            axA[0].set_xlabel('Component')
            divA = make_axes_locatable(axA[0])
            caxA = divA.append_axes('right',size='50%',pad=0.05)
            cbarA = plt.colorbar(imA,cax=caxA)
            imB=axA[1].matshow(H.T,cmap=plt.cm.afmhot)
            axA[1].set_title('Transposed coefficient matrix H'.format(n_components,init))
            axA[1].set_ylabel('Position')
            axA[1].set_xlabel('Component')
            divB = make_axes_locatable(axA[1])
            caxB = divB.append_axes('right',size='50%',pad=0.05)
            cbarB = plt.colorbar(imB,cax=caxB)
            plt.subplots_adjust(top=0.85)
            if resiliList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axA[0].get_yticks()[::-1]]
                y = [resiliList[n] for n in j]
                axA[0].set_yticklabels(y)
                axA[1].set_yticklabels(y)
            if outDir is not None:
                plt.savefig(os.path.join(outDir,'{}_nmf-
comps_{}.eps'.format(runName,n_components)),transparent=True,format='eps')
            figB,axB=plt.subplots(1,2,figsize=(30,20))
            N_copy=N.copy()
            np.fill_diagonal(N_copy,np.nan)
            axB[0].matshow(N_copy,cmap=plt.cm.afmhot)
            axB[0].set_title('Input matrix')
            axB[0].set_xlabel('Residue i')
            axB[0].set_ylabel('Residue j')
            axB[1].matshow(W.dot(H),cmap=plt.cm.afmhot)
            axB[1].set_title('Reconstructed input matrix (W.dot(H))')
            axB[1].set_xlabel('Residue i')
            axB[1].set_ylabel('Residue j')
            if resiliList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axB[0].get_yticks()[::-1]]
                y = [resiliList[n] for n in j]
                axB[0].set_xticklabels(y)

```

```

        axB[0].set_yticklabels(y)
        axB[1].set_xticklabels(y)
        axB[1].set_yticklabels(y)
    if outDir is not None:
        plt.savefig(os.path.join(outDir, '{}_nmf-
recon_{}.eps'.format(runName,n_components)),transparent=True,format='eps')
        figC,axC=plt.subplots(1,1,figsize=(30,20))
        p=axC.hist([W[:,i] for i in
range(W.shape[1])],bins=int(W.shape[0]/4),alpha=0.5,label=['Component {}'.format(k) for k in
range(n_components)])
        axC.legend()
        axC.set_title('NMF component histograms')
        axC.set_ylabel('Counts')
        axC.set_xlabel('Mutual information (bits)')
        if outDir is not None:
            try:
                plt.savefig(os.path.join(outDir, '{}_nmf-
hists_{}.eps'.format(runName,n_components)),transparent=True,format='eps')
            except:
                print('Failed to save NMF histograms, moving on...')
        else:
            plt.show()
            plt.close()

    return(W,H)

def component_comparison(M,mode='hist'):
    """
    """
    if mode.lower() == 'hist':
        figA,axA=plt.subplots(1,1,figsize=(30,15))
        plt.hist([M[:,i] for i in range(M.shape[1])])

    elif mode.lower() == 'ecdf':
        sort_list=[np.sort(M[:,i]) for i in range(M.shape[1])]
        yvals_list=[np.arange(len(sort_list[i]))/float(len(sort_list[i])) for i in
range(M.shape[1])]
        figA,axA=plt.subplots(1,1,figsize=(30,15))
        [plt.plot(dat[0],dat[1],'-o',label=str(i)) for i,dat in
enumerate(zip(sort_list,yvals_list))]
        plt.legend()
        plt.show()
    return

def sklearn_perform_manifold_learning(M,preproc=False):
    depth = M.shape[-1]
    M_flat = np.zeros((np.triu_indices_from(M[:, :, 0])[0].shape[0],depth))
    for i in range(depth):
        #M_rescale[:, :, i] = (M_rescale[:, :, i]-M_rescale[:, :, i].min())/(M_rescale[:, :, i].max()-
M_rescale[:, :, i].min())
        M_flat[:,i]=M[:, :, i][np.triu_indices_from(M[:, :, i])]
        print('Created flattened array with shape {}'.format(M_flat.shape))
        # Isomap
        X=manifold.Isomap(15,n_components=1).fit_transform(M_flat)
        reduced={'isomap':np.zeros_like(M[:, :, 0])}
        reduced['isomap'][np.triu_indices_from(reduced['isomap'])]=X[:,0]
        temp=reduced['isomap'].copy().T
        np.fill_diagonal(temp,0)
        reduced['isomap']=reduced['isomap']+temp
        figA,axA=plt.subplots(1,2,figsize=(25,15))
        M_mean=np.mean(M,axis=2)
        np.fill_diagonal(M_mean,np.nan)
        pa=axA[0].matshow(M_mean,cmap=plt.cm.afmhot)
        pb=axA[1].matshow(reduced['isomap'],cmap=plt.cm.afmhot)
        plt.colorbar(pb)
    return(reduced)

```

```

def temp_fast_ica(D,n_components,preprocess='meansubtract',resiList=[],plot=True,outDir=None):

    D_mean = D.copy()
    D_mean[np.isnan(D_mean)] = np.nanmean(D_mean)

    # Create m x n matrix with vectorized upper triangle for a given protein in column n with
    alignment position in row m
    nSets = np.shape(D_mean)[-1]
    nObs = len(D_mean[np.triu_indices_from(D_mean[:, :, 0])])
    nPosns = np.shape(D)[0]
    M = np.zeros([nObs,nSets])
    for p in range(nSets):
        np.fill_diagonal(D_mean[:, :, p], 0)
        M[:, p] = D_mean[:, :, p][np.triu_indices_from(D_mean[:, :, p])]

    if preprocess is not None:
        if preprocess == 'skpss':
            N = StandardScaler().fit_transform(M)
        elif preprocess == 'skpnorm':
            N = normalize(M)
        elif preprocess == 'meansubtract':
            N = M - np.mean(M)
        else:
            raise ValueError('Invalid preprocessing mode specified.')
    else:
        N = M.copy()
    print('Calculated two-dimensional data matrix M from input tensor with dimensions {}; M has
    dimensions {}'.format(D.shape, M.shape))

    ica = FastICA(n_components=n_components)
    P = ica.fit(N).transform(N)
    P /= P.std(axis=0)

    Q = np.zeros_like(D)
    V = [np.linalg.norm(np.mean(D_mean, axis=2), axis=1)]
    if plot:
        figA, axA = plt.subplots(nrows=1, ncols=n_components, squeeze=False, figsize=(30, 15))
        vmax = 2 * np.mean(P)
    for p in range(n_components):
        Q[:, :, p][np.triu_indices_from(Q[:, :, p])] = P[:, p]
        Q[:, :, p] = Q[:, :, p] + np.triu(Q[:, :, p], 1).T
        V.append(np.linalg.norm(Q[:, :, p], axis=1))
        if plot:
            sp = axA[0][p].matshow(Q[:, :, p], cmap=plt.cm.RdBu_r)
            axA[0][p].set_title('Component {}'.format(p))
            if resiList != []: # Relabel matrix axes with proper residue indices
                j = [int(n) for n in axA[0][p].get_xticks()[:-1]]
                x = [resiList[n] for n in j]
                axA[0][p].set_xticklabels(x)
                axA[0][p].set_yticklabels(x)

    if plot:
        if outDir is None:
            plt.show()
        if outDir is not None:
            try:
                plt.savefig(os.path.join(outDir, runName + '_pca.eps'), transparent=True, format='eps')
            except:
                print('Failed to save figure {}; will carry on anyways.'.format(thisFig))
    return(ica, P, Q)

def
compare_component_to_list(V, membership_list, resi_list=[], log=False, plot=False, outDir=None, runName
='run'):
    ...
    * Description:

    Assesses the distribution of per-residue values in vector `V` relative to some subset of
    residues.

```

```

* Input:
* `V`
    A vector or matrix of per-residue values.
* `membership_list`
    A list of residue indices considered members of some subset.
* `resi_list`
    An explicit list of residue indices corresponding to each entry in `V`.
    If provided, the entries of `membership_list` should follow `resi_list`.
    If not provided, indexing will be relative to enumeration of `V`.
* `log`
    If `True`, histograms will be drawn on log scale.
* `plot`
    Boolean specifying whether or not to create various plots.
* `outDir`
    If provided, plots will be output to this directory.
* `runName`
    A descriptive title.

* Output:
* `V_in`
    Elements of `V` with indices corresponding to `membership_list`.
* `V_out`
    Elements of `V` with indices not corresponding to `membership_list`.
* `perc_sub`
    Percent
...
if resi_list == []:
    resi_list = list(range(V.shape[0]))
elif len(resi_list) != V.shape[0]:
    raise ValueError('Provided resi_list is not equal in length to V!')
binary_list = [0]*len(resi_list)
for k,i in enumerate(membership_list):
    if i in resi_list:
        binary_list[resi_list.index(i)] = 1
    else:
        print('Element {} corresponding to position {} of membership_list not found in
resi_list, skipping...'.format(k,i))
binary_list=np.array(binary_list)
if V.shape[0] == V.shape[1]:
    V_temp=np.triu(V)+np.tril(V)
    np.fill_diagonal(V_temp,np.nan)
    V_in = V_temp[binary_list==1,:].flatten()
    V_in = V_in[~np.isnan(V_in)]
    V_out = V_temp[binary_list==0,:].flatten()
    V_out = V_out[~np.isnan(V_out)]
    V_temp=V_temp.flatten()
    V_temp = V_temp[~np.isnan(V_temp)]
else:

```

```

    V_temp = V.copy()
    V_in = V[binary_list==1]
    V_out = V[binary_list==0]
    h_full,_ = np.histogram(V_temp,range=(np.min(V_temp),np.max(V_temp)),bins=int(V.shape[0]/2))
    h_sub,_ = np.histogram(V_in,range=(np.min(V_temp),np.max(V_temp)),bins=int(V.shape[0]/2))
    perc_sub = (h_sub/h_full)
    print('\n(subset counts, full counts) per bin:')
    for i in range(int(V.shape[0]/2)):
        print(h_sub[i],h_full[i])
    if plot:
        figA,axA=plt.subplots(1,1,figsize=(30,20))

n,b,p=axA.hist([V_temp,V_in],alpha=0.5,normed=False,label=['all','subset'],bins=int(V.shape[0]/2)
,histtype='bar',log=log)
    axA.set_xlabel('Component value')
    axA.set_ylabel('Counts')
    axA.set_title('Comparison histograms, {}'.format(runName))
    plt.legend()
    if outDir is not None:

plt.savefig(os.path.join(outDir,'{}_compare_hists.eps'.format(runName)),transparent=True,format='
eps')
    figB,axB=plt.subplots(1,1,figsize=(30,20))
    perc_sub[np.isnan(perc_sub)] = 0
    axB.bar(range(len(perc_sub)),perc_sub,alpha=0.5)
    axB.set_xlabel('Bin')
    axB.set_ylabel('h_sub/h_full')
    axB.set_title('Histogram fractions, {}'.format(runName))
    if outDir is not None:
        plt.savefig(os.path.join(outDir,'{}_compare_hists-
frac.eps'.format(runName)),transparent=True,format='eps')
    return(V_in,V_out,perc_sub)

def data_to_structure(pdb, M, msa_seq, outDir, resiList=None, chain='A', runName='run',
pymol_percentile_list=[90], absolute_scale=True, pymol_spectrum='black red orange yellow white',
verbose=False):
    """
    * Description:

    A generic function to map data in `M` to a protein structure in `pdb`.
    Creates a new PDB file for each column of `M` where B-factors have been changed to values
of a given column of `M`.
    Also creates a text file containing the commands needed to visualize patterns in those
columns.

    * Input:

    * `pdb`

    An absolute path to a PDB file.

    * `M`

    A `numpy.ndarray` object; either a vector with length equal to the number of residues
in the structure in `pdb` or length equal to the length of `resiList`.
    Can also be a matrix, where each column is a different vector to be mapped.

    * `msa_seq`

    A sequence string corresponding to the elements of `M`.
    For example, `TKNYKQ`.

    * `outDir`

    If provided, plots will be output to this directory.

    * `resiList`

    Optional list of residue indices matching `msa_seq` and `M`.

```

```

* `chain`
    The chain in the structure from `pdb` that data will be mapped to.

* `runName`
    A descriptive title.

* `pymol_percentile_list`
    Draw spheres for data in this percentile.

* `verbose`
    More details.

* Output:

* `pdbList`
    ...
    A list of paths to created PDB files.
...
if type(pymol_percentile_list) in (float, int) and (pymol_percentile_list > 0 and
pymol_percentile_list <= 100):
    pymol_percentile_list = [pymol_percentile_list]
elif type(pymol_percentile_list) is list:
    check = [type(perc) for perc in pymol_percentile_list if type(perc) not in (int,float)]
    if len(check) > 0:
        raise ValueError('One or more elements of pymol_percentile_list are not of type int
or float!')
    else:
        raise ValueError('Invalid pymol_percentile_list value provided.')
if type(M) != list: # If M is a matrix, convert it to a list of vectors with the first being
the mean vector for the matrix and the following ones being sequential slices
    if isinstance(M,np.ndarray):
        if len(M.shape) > 1:
            V = []
            try:
                [V.append(M[:,i]) for i in range(min(np.shape(M)))]
            except:
                raise
        else:
            V = [list(M)]
    else:
        raise ValueError('M is neither a list of vectors nor a matrix.')
else: # If M is a list of vectors, just move right ahead.
    V = M.copy()
if absolute_scale:
    scale = [0,0]
    for v in V:
        if v.min() < scale[0]:
            scale[0] = v.min()
        if v.max() > scale[1]:
            scale[1] = v.max()
    print('Minimum and maximum over all components is {:.4f} and {:.4f},
respectively.'.format(*scale))
p=PDB.PDBParser()
try:
    structure = p.get_structure(os.path.basename(pdb),pdb)
except:
    raise
if resiList is None:
    resiList = [resi.id[1] for resi in list(structure[0][chain].get_residues())]
lookup = []
i=0
for r in msa_seq:
    if r == '-':
        lookup.append('-')

```

```

        else:
            lookup.append(i)
            i+=1
pdbList = []
pymol_obj_list = []
try:
    os.mkdir(os.path.join(outDir,runName))
except:
    pass
n_comps = len(V)-1
with open(os.path.join(outDir,runName,'{}_pymol-commands.pml'.format(runName)), 'w') as
c_file:
    for j,v in enumerate(V):
        b = v.copy()
        if absolute_scale:
            b = (b-scale[0])/(scale[1]-scale[0])*100
        else:
            b = (b-np.min(b))/(np.max(b)-np.min(b))*100 # Scale v relative to min/max over
range from 0 to 100
            print('Vector {}: (1, 10, 90, 99)th percentile = {:.42f}, {:.42f}, {:.42f},
{:.42f}'.format(j,np.percentile(b,1),np.percentile(b,10), np.percentile(b,90),
np.percentile(b,99)))
            for atom in structure[0][chain].get_atoms():
                try:
                    rIdx = resiList.index(atom.get_parent().id[1])
                    alnIdx = lookup.index(rIdx)
                    atom.set_bfactor(b[alnIdx])
                    if verbose:
                        print('\tFound residue {} in resiList, setting {} atom B-factor to
{:.42f}'.format(atom.get_parent().id[1],atom.id, b[alnIdx]))
                except ValueError:
                    #print('Residue {} not in resiList, setting atom B-factor to
zero.'.format(atom.get_parent().id[1]))
                    atom.set_bfactor(-.01)
                except IndexError:
                    print('Indexing error... (rIdx, alnIdx, atomID, resiName) = ({} , {} , {} ,
{} )'.format(rIdx,alnIdx,atom.id,atom.get_parent().id[1]))
                    raise
                io = PDB.PDBIO()
                io.set_structure(structure)
                thisPDBBase = '{}_{}_comp-{}-
{}'.format(runName,os.path.splitext(os.path.basename(pdb))[0], j, n_comps)
                thisPDBPath = os.path.join(outDir,runName,'{}.pdb'.format(thisPDBBase))
                pdbList.append(thisPDBPath)
                io.save(thisPDBPath)
                del(io)
                c_file.write('load {}\n'.format(thisPDBPath))
                for perc in pymol_percentile_list:
                    pymol_obj_list.append('{}_{}_{}_{}'.format(runName, j, n_comps, perc))
                    if perc > 50:
                        c_file.write('create {}_grp, {} and b > {:.42f}\n'.format(pymol_obj_list[-1],
thisPDBBase, np.percentile(b,perc))
                    else:
                        c_file.write('create {}_grp, {} and b < {:.42f} and b >
0\n'.format(pymol_obj_list[-1], thisPDBBase, np.percentile(b,perc))
                        c_file.write('show spheres, {}\n'.format(pymol_obj_list[-1]))
                        c_file.write('show surface, {}\n'.format(pymol_obj_list[-1]))
                        #c_file.write('color grey60, b<0')
                        c_file.write('spectrum b, {}, all, 0, 100\n'.format(pymol_spectrum))
                        c_file.write('hide lines\n')
                        c_file.write('show cartoon\n')
                        c_file.write('set transparency, 0.5\n')
                        c_file.write('set surface_quality, 1\n')
                        c_file.write('remove hydrogens\n')
                        c_file.write('remove solvent\n')
            return(pdbList)

def
matrix_sklearn_clustering(D, resiList=[],diag_suppress=None,method='agg',n_clusters=2,plot=False,o

```

```

utDir=None,runName='hc'):
    """
    *   Description:
        Uses `sklearn.clustering` methods to perform hierarchical clustering on a distance matrix
        D.

    *   Input:
        *   `D`
            A matrix.

        *   `resilist`
            Optional list of residue indices as integers.

        *   `diag_suppress`
            If `None`, diagonal of `M` will not be suppressed prior to clustering.
            If an integer, elements out to the `diag_suppress`th diagonal will be zeroed prior to
            clustering.

        *   `method`
            `{agg, aff, sc}`
            The `sklearn.cluster` method to use for clustering.
            If equal to `agg`, uses Ward's method of agglomerative clustering.
            If equal to `aff`, uses affinity propagation; this is the default.
            If equal to `sc`, uses spectral clustering (note: still need to implement conversion
            to affinity matrix prior to using this).

        *   `n_clusters`
            Number of clusters to create; only of use for `agg` method.

        *   `plot`
            `{False, True}`
            If true, plots the results of clustering as a series of matrices corresponding to the
            clusters followed by histograms of mutual information for each cluster.
            These plots are useful in determining the total number of clusters.

        *   `outDir`
            Path to a directory in which to save plots.

        *   `runName`
            A descriptive title.

    *   Output:
        *   `C`
            Standard object output by `sklearn.cluster` methods.
    """
    # If requested, zero diagonal/off-diagonal elements
    if diag_suppress is not None and type(diag_suppress) == int:
        try:
            E = np.triu(D,diag_suppress) + np.tril(D,-diag_suppress)
        except:
            print('Failed to zero diagonal(s), check diag_crop and try again!')
            raise

```

```

else:
    E = D.copy()
    # Perform the clustering
    if method=='agg':
        print('\nClustering with agglomerative clustering using scikit-learn...')
        C = AgglomerativeClustering(n_clusters=n_clusters,linkage='ward').fit(E)
    if method=='aff':
        print('\nClustering with affinity propagation using scikit-learn...')
        C = AffinityPropagation().fit(E)
    if method=='sc': # Need to transform to affinity matrix...?
        # Convert D to A here
        print('\nUsing spectral clustering from scikit-learn...')
        C = SpectralClustering().fit(E)
    # Analyze the clustering results
    groups=np.unique(C.labels_)
    E_temp = E.copy()
    if diag_suppress is not None and type(diag_suppress) == int:
        E_temp[np.tril_indices_from(E_temp)]=np.nan # Ignore diagonal
    else:
        E_temp[np.tril_indices_from(E_temp,-1)]=np.nan # Include diagonal
    E_mean = np.nanmean(E_temp)
    E_sum = np.nansum(E_temp)
    sorted_idx = [x for (y,x) in sorted(zip(C.labels_,list(range(len(E)))))]
    labels_sort=sorted(C.labels_)
    E_sort=E[sorted_idx,:].copy()
    E_sort=E_sort[:,sorted_idx]
    if diag_suppress is not None and type(diag_suppress) == int:
        np.fill_diagonal(E_sort,np.nan)
    d_list = []
    print('\nOverall:\nsum(mi) = {:.0.2f} bits\nmean(mi) = {:.0.2f} bits\n'.format(E_sum,E_mean))
    if plot:
        ticks_loc = [0]
        ticks_lab = [0]
        for i in range(1,len(labels_sort)):
            if labels_sort[i] != labels_sort[i-1]:
                ticks_loc.append(i)
                ticks_lab.append(labels_sort[i])
        cmap = plt.cm.afmhot
        cmap.set_bad('k',.3)
        #cmap.set_bad(alpha=0)
        figA,axA=plt.subplots(nrows=1,ncols=1,figsize=(15,15))
        vmin=np.percentile(E_sort,5)
        vmax=np.percentile(E_sort,95)
        imA=axA.matshow(E_sort,cmap=cmap,vmin=vmin,vmax=vmax)
        axA.set_xticks(ticks_loc)
        axA.set_xticklabels(ticks_lab)
        axA.set_yticks(ticks_loc)
        axA.set_yticklabels(ticks_lab)
        axA.set_xlabel('residue cluster index')
        axA.set_ylabel('residue cluster index')
        axA.set_title('Input matrix, clustered, {} groups'.format(len(groups)))
        divA = make_axes_locatable(axA)
        caxA = divA.append_axes('right',size='5%',pad=0.05)
        cbarA = plt.colorbar(imA,cax=caxA)
        figB,axB=plt.subplots(nrows=1,ncols=n_clusters,squeeze=False,figsize=(45,15))
    for grp in list(groups):
        d=E_temp.copy()
        d[C.labels_ != grp,:] = np.nan
        d[:,C.labels_ != grp] = np.nan
        d_sum = np.nansum(d)
        d_mean = np.nanmean(d)
        d_list.append(d)
        print('Cluster {}: \nsum(mi_grp) = {:.0.2f} bits\nmean(mi_grp) = {:.0.2f}
bits\n'.format(grp,d_sum,d_mean))
        if plot:
            ax_grp = axB[0][grp].matshow(d,cmap=cmap,vmin=vmin,vmax=vmax)
            axB[0][grp].set_xlabel('residue i')
            axB[0][grp].set_ylabel('residue j')
            axB[0][grp].set_title('Cluster {}'.format(grp))

```

```

    if resiList != []: # Relabel matrix axes with proper residue indices
        j = [int(n) for n in axB[0][grp].get_xticks()[::-1]]
        k = [resiList[n] for n in j]
        axB[0][grp].set_xticklabels(k)
        axB[0][grp].set_yticklabels(k)
    if plot:
        figC,axC=plt.subplots(nrows=1,ncols=1,squeeze=False,figsize=(45,15))
        bins=np.linspace(np.nanmin(E_temp),np.nanmax(E_temp),round(len(E)/2))
        ax_hist = axC[0][0].hist([d[~np.isnan(d)] for d in
d_list],bins=bins,alpha=0.6,log=False,edgecolor=None,label=['Cluster {}'.format(g) for g in
list(groups)])
        axC[0][0].set_xlabel('Mutual information (bits)')
        axC[0][0].set_ylabel('Counts')
        axC[0][0].set_title('Cluster histograms')
        plt.grid()
        plt.legend()
        plt.show()
    if outDir is not None:
        try:
            figA.savefig(os.path.join(outDir,runName+'_cluster-
mat_sort_{}.eps'.format(n_clusters)),transparent=True,close=True,verbose=True)
            figB.savefig(os.path.join(outDir,runName+'_cluster-
mat_split_{}.eps'.format(n_clusters)),transparent=True,close=True,verbose=True)
            figC.savefig(os.path.join(outDir,runName+'_cluster-
hist_{}.eps'.format(n_clusters)),transparent=True,close=True,verbose=True)
        except:
            print('Failed to save plots!')
    return(C)

def sklearn_clustering_to_pymol(C,outDir,resiList=[],runName=None):
    """
    * Description:

    Converts the output of a given `sklearn.cluster` function to a dictionary of residue
    groups and writes `PyMOL` commands needed for structural assessment.

    * Input:

    * `C`

    Standard dictionary output by `sklearn.cluster` methods.

    * `outDir`

    Path to directory in which to write text files containing `PyMOL` commands.

    * `resiList`

    Optional list of residue indices as integers.

    * `runName`

    A descriptive title.

    * Output:

    * `groupDict`

    Dictionary for which each key/value pair is a group index and list of residue indices
    for that group, respectively.
    """
    if os.path.isdir(outDir):
        if type(resiList) == list and resiList != [] and len(resiList) == len(C.labels_):
            groupIter = zip(C.labels_,resiList)
        else:
            print('No residue labels...')
            print(type(resiList))
            print(len(resiList), len(C.labels_))
            groupIter = zip(C.labels_,range(len(C.labels_)))

```

```

groupDict = {}
for group in groupIter:
    if group[0] not in groupDict.keys():
        groupDict[group[0]] = []
        groupDict[group[0]].append(group[1])
    colors =
['red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'slate', 'firebrick', 'chartreuse']
    with open(os.path.join(outDir, '{}_clusters_{}.txt'.format(runName, C.n_clusters)), 'w') as
f:
    for group in groupDict.items():
        f.write('create grp{}, resi {} and chain A\n'.format(group[0], '+'.join([str(k)
for k in group[1]])))
        f.write('color {}, grp{}\n'.format(colors[group[0]%len(colors)], group[0]))
        f.write('show surface, grp{}\n'.format(group[0]))
    else:
        raise OSError('Invalid output directory provided!')
    return(groupDict)

def dendrogram_to_pymol(Z, level, resiLabels, outFile):
    """
    THIS DOESNT WORK PROPERLY, DO NOT USE UNTIL FIXED
    Functions add_node and label_tree similar to those by Max Leiserson, see
    https://gist.github.com/mdml/7537455
    """
    def add_node(node, parent ):
        # First create the new node and append it to its parent's children
        newNode = dict( node_id=node.id, children=[] )
        parent['children'].append( newNode )
        # Recursively add the current node's children
        if node.left:
            add_node(node.left, newNode)
        if node.right:
            add_node(node.right, newNode)

    def label_tree(n):
        if len(n['children']) == 0: # If the node is a leaf, then we have its name
            leafNames = [resiLabels[n['node_id']]]
        else: # If not, flatten all the leaves in the node's subtree
            leafNames = []
            for child in n['children']:
                leafNames = leafNames + label_tree(child)
        del(n['node_id'])
        #n['name'] = name = '-'.join(sorted(map(str, leafNames)))
        n['name'] = name = leafNames
        return(leafNames)

    def get_children_at_level(n, l, level, groupList):
        if l != level:
            if len(n['children']) > 0: # Keep going until we get to the level we want
                for sub in n['children']:
                    l+=1
                    get_children_at_level(sub, l, level, groupList)
            else: # If we hit a dead end, add singleton
                pass
                # groupList.append(n['name'])
        elif l == level:
            for sub in n['children']:
                groupList.append(sub['name'])

    T = hierarchy.to_tree(Z, rd=False)
    R = hierarchy.dendrogram(Z, labels=resiLabels, get_leaves=True, no_plot=True)
    dendro = dict(children=[], name='root')

    add_node(T, dendro)
    label_tree(dendro['children'][0])

    allGroupList = []
    groupList = []
    get_children_at_level(dendro, 0, level, groupList)

```

```

level = 0
while len([sub for sub in groupList if len(sub) > 1]) > 0:
    groupList = []
    get_children_at_level(dendro,0,level,groupList)
    glFlat = [val for sub in groupList for val in sub]
    print('{} residues in groups at level {}'.format(len(glFlat),level))
    allGroupList.append([sub for sub in groupList if len(sub) > 1])
    level += 1

if os.path.isdir(outFile) == False:
    colors =
['red','green','blue','cyan','magenta','yellow','slate','firebrick','chartreuse']
    with open(outFile,'w') as f:
        for i,treeLevel in enumerate(allGroupList):
            for j,group in enumerate(treeLevel):
                f.write('create grp{}-{}, resi {}\n'.format(i,j,'+'.join([str(k) for k in
group])))
                f.write('color {}, grp{}-{}\n'.format(colors[i%len(colors)],i,j))
                f.write('show surface, grp{}-{}\n'.format(i,j))
    return(dendro,allGroupList)

def sort_matrix_by_dendrogram(MI,Z):
    """
    Sorts a matrix by dendrogram output.
    """
    R = hierarchy.dendrogram(Z,no_plot=True)
    MI_sort = MI[:,R['ivl']][R['ivl']]
    return(MI_sort)

def dendrogram_to_groups_by_color(R,bugfix=False,outDir=None):
    """
    Get a list of lists, where each sublist is a set of leaves grouped by coloring.
    """
    clusters=zip(R['color_list'],R['ivl'])
    groupList = []
    if bugfix:
        startFlag = True
        for i,leaf in enumerate(clusters):
            if startFlag:
                groupList.append([leaf[1]])
                prevC = leaf[0]
                startFlag = False
            else:
                if leaf[0] != prevC:
                    groupList[-1].append(leaf[1])
                    prevC = leaf[0]
                    startFlag = True
                elif leaf[0] == prevC:
                    groupList[-1].append(leaf[1])
    else:
        prevC = 'x'
        for i,leaf in enumerate(clusters):
            if leaf[0] != prevC:
                groupList.append([leaf[1]])
                prevC = leaf[0]
            elif leaf[0] == prevC:
                groupList[-1].append(leaf[1])

    if outDir is not None:
        if os.path.isdir(outDir):
            colors =
['red','green','blue','cyan','magenta','yellow','slate','firebrick','chartreuse']
            with open(os.path.join(outDir,'clusters.txt'),'w') as f:
                for i,group in enumerate(groupList):
                    f.write('create grp{}, resi {}\n'.format(i,'+'.join([str(k) for k in
group])))
                    f.write('color {}, grp{}\n'.format(colors[i%len(colors)],i))
                    f.write('show surface, grp{}\n'.format(i))
    return(groupList)

```

```

def plot_all_matrices(miDict, figDict=None, outDir=None, matList = ['m', 'ms', 'r', 'd', 'ds']):
    """
    * Description:

    Plot all MI, D, and Ds matrices for all datasets for all data types.
    This thing is kind of a mess.

    * Input:

    * `miDict`

    A dictionary which stores MI data, etc.
    All matrices in subdictionaries keyed with `matrices` will be plotted.
    `miDict` has the following basic structure:

    * `proteins`: `dict`

        * `your_favorite_protein_A`: `dict`

            * `sets`: `dict`

                * `ds1_1`: `dict`

                    * `data`: `dict`

                        * `dihedrals`: `dict`

                            * `matrices`: `dict`

                                All `numpy.ndarray`s keyed beyond this level will be
plotted.

                                * `coordinates`: `dict`

                                    * `matrices`: `dict`

                                        All `numpy.ndarray`s keyed beyond this level will be
plotted.

                                * `figDict`

                                If provided, new figures will be added to the dictionary `figDict`.

                                * `outDir`

                                If provided, plots will be saved to the path specified.

    * Output:

    * `figDict`

    ... A dictionary that keys all figures created.
    """

def mat_to_txt(this_set, resi_indices, prot, dataType, dataSet, mat, outDir):
    triu_idx = np.triu_indices_from(this_set)
    #n_ele = this_set.shape[0]
    n_ele = len(triu_idx[0])
    flat = np.zeros((n_ele,5))
    for i in range(n_ele):
        j,k = triu_idx[0][i], triu_idx[1][i]
        flat[i,:] = [j, k, resi_indices[j], resi_indices[k], this_set[j,k]]
    flat = flat[np.argsort(flat[:,4])[:-1]]
    try:
        with open(os.path.join(outDir, '{}-{}-{}-{}_sorted.txt'.format(prot, dataType,
dataSet[0], mat)), 'w') as f:
            f.write('# {} {} {} {} \n\tj\ttrj\tval\n'.format(prot, dataType, dataSet[0]
,mat))

```

```

        [f.write('{:.0f}\t{:.0f}\t{:.0f}\t{:.0f}\t{:.6f}\n'.format(i,j,k,l,m)) for
i,j,k,l,m in flat]
    except:
        print('Failed to write matrix data to text file!')
        raise

    if figDict is None or type(figDict) != dict:
        figDict = {}
    for prot in sorted(miDict['proteins'].keys()):
        for dataType in ('dihedrals', 'coordinates'):
            for mat in matList:
                if mat not in ('m', 'ms', 'r', 'd', 'ds'):
                    continue
                thisFig = 'indivMats_{}_{_}'.format(prot, dataType, mat)
                print('Plotting {}'.format(thisFig))
                n_plots = len([s for s in miDict['proteins'][prot]['sets'].keys() if s !=
'merge'])
                figDict[thisFig] = {}
                figDict[thisFig]['fig'], figDict[thisFig]['ax'] =
plt.subplots(nrows=1, ncols=n_plots, squeeze=False, figsize=(n_plots*10, 10))
                #meanVmax =
np.mean([miDict['proteins'][prot]['sets'][dataSet]['data'][dataType]['matrices'][mat] for dataSet
in miDict['proteins'][prot]['sets'].keys() if mat in
miDict['proteins'][prot]['sets'][dataSet]['data'][dataType]['matrices'].keys()])
                if mat in ('m', 'r'):
                    cmap=plt.cm.afmhot
                else:
                    cmap=plt.cm.afmhot_r
                for i, dataSet in enumerate(sorted(miDict['proteins'][prot]['sets'].items())):
                    if dataSet[0] == 'merge':
                        continue
                    try:
                        this_set = dataSet[1]['data'][dataType]['matrices'][mat]
                    except KeyError:
                        print('Data type {} for matrix {} not found,
skipping.'.format(dataType, mat))
                        plt.close('all')
                        key_error = True
                        continue
                    key_error = False
                    if i == 0 and mat != 'r':
                        if mat in ('m', 'ms'):
                            vmin = 0
                        else:
                            vmin = np.percentile(this_set[np.triu_indices_from(this_set, 5)], 1)
                            vmax = np.percentile(this_set[np.triu_indices_from(this_set, 5)], 99)
                    elif i == 0 and mat == 'r':
                        vmin = 0
                        vmax = 1

                    # vmin=0
                    # vmax=1
                    #print(i, figDict[thisFig]['ax'][0][i])
                    #print(figDict[thisFig]['ax'][0][i])

p=figDict[thisFig]['ax'][0][i].matshow(this_set, cmap=cmap, vmin=vmin, vmax=vmax)

                try:
                    j = [int(n) for n in figDict[thisFig]['ax'][0][i].get_xticks()[:-1]]
                    k = [miDict['proteins'][prot]['resiIndices'][n] for n in j]
                    figDict[thisFig]['ax'][0][i].set_xticklabels(k)
                    figDict[thisFig]['ax'][0][i].set_yticklabels(k)
                except:
                    print('Failed to relabel axes ticks, skipping. Check
miDict[\protein\][prot][\set\][dataSet][\resiIndices\] and try again.')
                    figDict[thisFig]['ax'][0][i].set_title('{}-{}-{}-{}'.format(prot, dataType,
dataSet[0], mat))
                    figDict[thisFig]['ax'][0][i].set_xlabel('residue i')
                    figDict[thisFig]['ax'][0][i].set_ylabel('residue j')

```

```

        figDict[thisFig][dataSet[0]+'_div_p'] =
make_axes_locatable(figDict[thisFig]['ax'][0][i])
        figDict[thisFig][dataSet[0]+'cax_p'] =
figDict[thisFig][dataSet[0]+'_div_p'].append_axes('right',size='5%',pad=0.05)
        figDict[thisFig][dataSet[0]+'cbar_p'] =
plt.colorbar(p,cax=figDict[thisFig][dataSet[0]+'cax_p'],label='Distance (Angstrom)')
        if outDir is not None:
            mat_to_txt(this_set, miDict['proteins'][prot]['resiIndices'], prot,
dataType, dataSet, mat, outDir)

        if outDir is None and not key_error:
            plt.show()
        elif outDir is not None and not key_error:
            try:

plt.savefig(os.path.join(outDir,thisFig+'.eps'),transparent=True,format='eps')
            plt.close('all')
        except:
            print('Failed to save figure {}; will carry on anyways.'.format(thisFig))

    if 'distance' in dataSet[1]['data'].keys() and 'contacts' in dataSet[1]['data'].keys():
        thisFig = 'indivMats_{}_dist-contacts'.format(prot)
        print('Plotting {...}'.format(thisFig))
        n_plots = len([s for s in miDict['proteins'][prot]['sets'].keys() if s != 'merge'])
        figDict[thisFig] = {}
        figDict[thisFig]['fig'],figDict[thisFig]['ax'] =
plt.subplots(nrows=2,ncols=n_plots,squeeze=False,figsize=(n_plots*10,20))
        for i,dataSet in enumerate(sorted(miDict['proteins'][prot]['sets'].items())):
            if dataSet[0] == 'merge':
                continue
            try:
                this_set_cont = dataSet[1]['data']['contacts']['matrices']['m']
                this_set_dist = dataSet[1]['data']['distance']['matrices']['m']
            except KeyError:
                print('Failed to find contact map data for set {} of {},
skipping.'.format(dataSet[0],thisFig))
                print(dataSet[1]['data']['contacts'].keys())
                print(dataSet[1]['data']['contacts']['matrices'].keys())
                raise
            p=figDict[thisFig]['ax'][0][i].matshow(this_set_dist,cmap=plt.cm.afmhot_r)
            q=figDict[thisFig]['ax'][1][i].matshow(this_set_cont,cmap=plt.cm.afmhot)
            try:
                j = [int(n) for n in figDict[thisFig]['ax'][0][i].get_xticks()[::-1]]
                k = [miDict['proteins'][prot]['resiIndices'][n] for n in j]
                figDict[thisFig]['ax'][0][i].set_xticklabels(k)
                figDict[thisFig]['ax'][0][i].set_yticklabels(k)
                figDict[thisFig][dataSet[0]+'_div_p'] =
make_axes_locatable(figDict[thisFig]['ax'][0][i])
                figDict[thisFig][dataSet[0]+'cax_p'] =
figDict[thisFig][dataSet[0]+'_div_p'].append_axes('right',size='5%',pad=0.05)
                figDict[thisFig][dataSet[0]+'cbar_p'] =
plt.colorbar(p,cax=figDict[thisFig][dataSet[0]+'cax_p'],label='Distance (Angstrom)')

                figDict[thisFig]['ax'][1][i].set_xticklabels(k)
                figDict[thisFig]['ax'][1][i].set_yticklabels(k)
                figDict[thisFig][dataSet[0]+'_div_q'] =
make_axes_locatable(figDict[thisFig]['ax'][1][i])
                figDict[thisFig][dataSet[0]+'cax_q'] =
figDict[thisFig][dataSet[0]+'_div_q'].append_axes('right',size='5%',pad=0.05)
                figDict[thisFig][dataSet[0]+'cbar_q'] =
plt.colorbar(q,cax=figDict[thisFig][dataSet[0]+'cax_q'],label='Contact fraction')

            except:
                print('Failed to relabel axes ticks, skipping. Check
miDict[\'protein\'][prot][\'set\'][dataSet[0][\'resiIndices\']] and try again.')
                figDict[thisFig]['ax'][0][i].set_title('{ }-{}-{}-{}'.format(prot, 'contacts',
dataSet[0] , 'd_raw'))
                figDict[thisFig]['ax'][0][i].set_xlabel('residue i')
                figDict[thisFig]['ax'][0][i].set_ylabel('residue j')

```

```

        figDict[thisFig]['ax'][1][i].set_title('{}-{}-{}-{}'.format(prot, 'contacts',
dataSet[0] , 'd_bin'))
        figDict[thisFig]['ax'][1][i].set_xlabel('residue i')
        figDict[thisFig]['ax'][1][i].set_ylabel('residue j')
        if outDir is None:
            plt.show()
        elif outDir is not None:
            try:
plt.savefig(os.path.join(outDir,thisFig+'.eps'),transparent=True,format='eps')
            plt.close('all')
        except:
            print('Failed to save figure {}; will carry on anyways.'.format(thisFig))
plt.close('all')
return(figDict)

def quick_mat_plot(M,ax_title='',fig_title='', x_lab='x',y_lab='y',cmap=plt.cm.afmhot,
resi_list=[], cbar_label='units', v=(), outDir=None, fName='plot', plt_format='eps'):
    cmap=copy.copy(cmap)
    cmap.set_bad([0.75,0.75,0.75],1.)
    fig,ax = plt.subplots(nrows=1,ncols=1, figsize=(10,10))
    if (type(v) is list or type(v) is tuple) and len(v) == 2:
        p = ax.matshow(M,cmap=cmap, vmin=v[0], vmax=v[1])
    else:
        p = ax.matshow(M,cmap=cmap)
    if type(resi_list) is list and len(resi_list) > 0:
        j = [int(n) for n in ax.get_xticks()[:-1]]
        k = [resi_list[n] for n in j]
        ax.set_xticklabels(k,fontsize=18)
        ax.set_yticklabels(k,fontsize=18)
    else:
        ax.set_xticklabels([int(i) for i in ax.get_xticks()],fontsize=18)
        ax.set_yticklabels([int(i) for i in ax.get_yticks()],fontsize=18)
    ax.set_title(ax_title,fontsize=18)
    ax.set_xlabel(x_lab,fontsize=18)
    ax.set_ylabel(y_lab,fontsize=18)
    div = make_axes_locatable(ax)
    cax = div.append_axes('right',size='5%',pad=0.05)
    cbar = plt.colorbar(p,cax=cax,label=cbar_label)
    fig.suptitle(fig_title,fontsize=28)
    if outDir is None:
        plt.show()
    elif outDir is not None:
        try:

plt.savefig(os.path.join(outDir,'{}.{}'.format(fName,plt_format)),transparent=True,format=plt_for
mat)
        plt.close()
    except:
        raise

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='A lot of stuff goes here.')
    parser.add_argument('miPickle', type=str, help='a pickled dictionary containing MI matrices,
keyed appropriately')
    parser.add_argument('--pdbDir', type=str,default=os.getcwd(), help='a directory containing
PDB files with names matching MSA in the miDict found in miPickle.')
    parser.add_argument('--outDir',type=str, default=os.getcwd(), help='a directory into which to
dump things')
    parser.add_argument('--runName',type=str,default='ermi',help='an optional name to prepend to
output for the run')
    parser.add_argument('--refSeqName',type=str,default=None,help='optional reference sequence
name for which to create MI and distance matrices')
    parser.add_argument('-o','--old',action='store_true',default=False, help='run analysis v1')
    parser.add_argument('-p','--plot',action='store_true',default=False, help='draw plots')
    parser.add_argument('-n','--norm',action='store_true',default=False,help='normalize MI
matrices (L1)')
    args = parser.parse_args()

```

```

# 0: Setup
try:
    fmt = '{:=^'+str(os.popen('stty size', 'r').read().split()[1])+}'
except:
    fmt = '{:=^30}'
print('\n'+fmt.format(' ANALYZE_ERMI.PY ')+'\n')
if args.runName == '':
    args.runName = 'run'
if os.path.isfile(args.miPickle): # load the dictionary. Expects miDict with structure
miDict['proteins'][proteinString]['data'][dataTypeString]['matrices'][matrixName] = matrix and
miDict['proteins'][proteinString]['resiIndices'] = listOfIntegers
miDict = pickle.load(open(args.miPickle,'rb'))
else:
    raise ValueError('Invalid path to miPickle!')
if not isinstance(miDict,dict):
    raise ValueError('miDict from miPickle must be a valid dictionary!')
args.outDir = os.path.join(args.outDir,args.runName+'_output')
try:
    os.mkdir(args.outDir)
except:
    pass
pdb_file_list = glob.glob(os.path.join(args.pdbDir,'*.pdb'))

if args.old:
    # 1: Calculate distance matrices, plot each step
    print('\nCalculating distance matrices...\n')
    for prot in miDict['proteins'].keys():
        for dSet in miDict['proteins'][prot]['sets'].keys():
            for dtype in miDict['proteins'][prot]['sets'][dSet]['data'].keys():
                if dtype not in ('dihedrals','coordinates'):
                    continue
                theseMats = miDict['proteins'][prot]['sets'][dSet]['data'][dtype]['matrices']
                if 'm' in theseMats.keys():
                    print('\n\nFound MI matrix for {} {}, {}'.format(prot,dSet,dtype))
                    if dtype.lower() in ('angles','dihedrals'): # Matrix for dihedral
*a*ngles

miDict['proteins'][prot]['sets'][dSet]['data'][dtype]['matrices']['d'],miDict['proteins'][prot]['sets'][dSet]['data'][dtype]['matrices']['ds']=MI_to_D(theseMats['m'],runName='{}-{}-{}-{}').format(args.runName,prot,dSet,dtype),plot=args.plot,outDir=args.outDir)
                    elif dtype.lower() in ('coordinates'): # Matrix for cartesian
*c*oordinates

miDict['proteins'][prot]['sets'][dSet]['data'][dtype]['matrices']['d'],miDict['proteins'][prot]['sets'][dSet]['data'][dtype]['matrices']['ds']=MI_to_D(theseMats['m'],normMode='full',runName='{}-{}-{}-{}').format(args.runName,prot,dSet,dtype),plot=args.plot,outDir=args.outDir)
                    else:
                        raise ValueError('')
                else:
                    print('Could not find MI matrix for dataset {} of {} {},
skipping.'.format(dSet,prot,dtype))

    # 2: Plot all of each type of matrix together for a given protein:
    print('\nPlotting all individual matrices...\n')
    if args.plot:
        figDict = plot_all_matrices(miDict,outDir=args.outDir)

    # 3: Create average matrices across datasets
    print('\nCalculating merged matrices...\n')
    if args.plot:
        miDict,figDict =
calculate_merged_mi_corr_matrices(miDict,plot=args.plot,outDir=args.outDir,figDict=figDict,norm=arg
args.norm)
        else:
            miDict =
calculate_merged_mi_corr_matrices(miDict,plot=args.plot,outDir=args.outDir,norm=args.norm)

    # 4: If alignment is available, calculate consensus MI and correlation coefficient
matrices and corresponding distance matrix

```

```

# This can and needs to be dramatically simplified
print('\nCalculating consensus matrices...\n')
for msaTitle,msa in miDict['consensus']['alignments'].items():
    tempMIDict = {'dihedrals':{},'coordinates':{}}
    tempRDict = {'dihedrals':{},'coordinates':{}}
    tempDSDict = {'dihedrals':{},'coordinates':{}}
    miDict['consensus']['data'] = {msaTitle:{'dihedrals':{},'coordinates':{}}}
    for prot in miDict['proteins'].keys():
        if 'merge' in miDict['proteins'][prot]['sets'].keys():
            for dType in tempMIDict.keys():
tempMIDict[dType]['{}_{}'.format(prot,dType)]=miDict['proteins'][prot]['sets']['merge']['data'][d
Type]['matrices']['mi_mean']

tempDSDict[dType]['{}_{}'.format(prot,dType)]=miDict['proteins'][prot]['sets']['merge']['data'][d
Type]['matrices']['ds_mean']

tempRDict[dType]['{}_{}'.format(prot,dType)]=miDict['proteins'][prot]['sets']['merge']['data'][dT
ype]['matrices']['corcoef_mean']

        for dType in tempMIDict.keys():
            if dType not in ('coordinates','dihedrals'):
                continue
                print('\nCalculating consensus MI matrix for {} data...'.format(dType))
                mi, mi_lookup, _ =
consensus_mi_tensor_from_msa({msaTitle:msa},tempMIDict[dType],dimRedMode='mean',refSeqName=args.r
efSeqName,runInfo=dType,outDir=args.outDir,plot=args.plot)
                ds, ds_lookup, _ =
consensus_mi_tensor_from_msa({msaTitle:msa},tempDSDict[dType],dimRedMode='mean',refSeqName=args.r
efSeqName,runInfo=dType,outDir=args.outDir,plot=args.plot)
                corcoef, corcoef_lookup, _ =
consensus_mi_tensor_from_msa({msaTitle:msa},tempRDict[dType],dimRedMode='mean',refSeqName=args.r
efSeqName,runInfo=dType,outDir=args.outDir,plot=args.plot)
                miDict['consensus']['data'][msaTitle][dType]['mi_{}_full'.format(msaTitle)]=mi
                miDict['consensus']['data'][msaTitle][dType]['ds_{}_full'.format(msaTitle)]=ds

miDict['consensus']['data'][msaTitle][dType]['corcoef_{}_full'.format(msaTitle)]=corcoef
                if dType == 'dihedrals':
                    mis,d,ds =
MI_to_D(mi['consensus_compressed'],runName='{}_consensus_{}_{}'.format(args.runName,msaTitle,dTyp
e),plot=args.plot,outDir=args.outDir)
                    if 'sequence_compressed' in mi.keys():
                        mis,d_seq,ds_seq =
MI_to_D(mi['sequence_compressed'],runName='{}_sequence_{}_{}'.format(args.runName,msaTitle,dType)
,plot=args.plot,outDir=args.outDir)
                    elif dType == 'coordinates':
                        mis,d,ds =
MI_to_D(mi['consensus_compressed'],runName='{}_consensus_{}_{}'.format(args.runName,msaTitle,dTyp
e),normMode='full',plot=args.plot,outDir=args.outDir)
                        if 'sequence_compressed' in mi.keys():
                            mis_seq,d_seq,ds_seq =
MI_to_D(mi['sequence_compressed'],runName='{}_sequence_{}_{}'.format(args.runName,msaTitle,dType)
,normMode='full',plot=args.plot,outDir=args.outDir)
                            if 'sequence_compressed' in mi.keys():
                                #
                                miDict['consensus']['data'][msaTitle][dType]['mis_{}_sequence'.format(msaTitle)]={'sequence_compr
essed':mis_seq}

                                miDict['consensus']['data'][msaTitle][dType]['d_{}_sequence'.format(msaTitle)]={'sequence_compres
sed':d_seq}

                                miDict['consensus']['data'][msaTitle][dType]['ds_{}_sequence'.format(msaTitle)]={'sequence_compre
ssed':ds_seq}

                                miDict['consensus']['data'][msaTitle][dType]['d_{}_consensus'.format(msaTitle)]={'consensus_compr
essed':d}

                                miDict['consensus']['data'][msaTitle][dType]['ds_{}_consensus'.format(msaTitle)]={'consensus_comp
ressed':ds}

```

```

miDict['consensus']['data'][msaTitle][dtype]['lookup_{}_consensus'.format(msaTitle)]=mi_lookup

else:
    # 1: Plot all individual matrices:
    if args.plot:
        print('Plotting all individual matrices...\n')
        temp_dir = os.path.join(args.outDir, 'indivMats')
        try:
            os.mkdir(temp_dir)
        except:
            pass
        figDict = plot_all_matrices(miDict, outDir=temp_dir, matList=['m', 'r',
'm_harm', 'd_bin', 'd_raw'])
        plt.close('all')

    # 2: Create average matrices across datasets
    print('\nCalculating merged MI matrices and contact graphs...')
    if args.plot:
        temp_dir = os.path.join(args.outDir, 'mergeMats')
        try:
            os.mkdir(temp_dir)
        except:
            pass
        miDict, figDict =
calculate_merged_mi_corr_matrices(miDict, plot=args.plot, outDir=temp_dir, figDict=figDict, norm=args
.norm)
        miDict, figDict =
calculate_merged_cm_matrices(miDict, plot=args.plot, outDir=temp_dir, figDict=figDict)
        plt.close('all')
    else:
        miDict =
calculate_merged_mi_corr_matrices(miDict, plot=args.plot, outDir=args.outDir, norm=args.norm)
        miDict = calculate_merged_cm_matrices(miDict, plot=args.plot, outDir=args.outDir)
    # 3: Scale mean matrices, create difference matrices for full and harmonic restraint
matrices
    print('\nScaling diagonals of mean matrices...')
    for prot, dtype in it.product(miDict['proteins'].items(), ('dihedrals', 'coordinates')):
        temp_dir = os.path.join(args.outDir, 'diagRescale')
        try:
            os.mkdir(temp_dir)
        except:
            pass
        prot[1]['sets']['merge']['data'][dtype]['matrices']['mi_mean_rescale'] =
rescale_matrix_diag(prot[1]['sets']['merge']['data'][dtype]['matrices']['mi_mean'],
plot=args.plot, runName='{}_{}_mi'.format(prot[0], dtype), outDir=temp_dir)
        try:
            prot[1]['sets']['merge']['data'][dtype]['matrices']['corcoef_mean_rescale'] =
rescale_matrix_diag(prot[1]['sets']['merge']['data'][dtype]['matrices']['corcoef_mean'],
plot=args.plot, runName='{}_{}_corcoef'.format(prot[0], dtype), outDir=temp_dir)
        except KeyError:
            continue
    if args.plot:
        plt.close('all')

    # 4: Create consensus matrices if there is an MSA in miDict
    for msa_name, msa in miDict['consensus']['alignments'].items():
        print('\nCollecting data for consensus matrix calculation using MSA
{}...'.format(msa_name))
        if not isinstance(msa, Align.MultipleSeqAlignment):
            print('Entry {} is not a multiple sequence alignment,
skipping!'.format(msa_name))
            continue
        temp_dir = os.path.join(args.outDir, 'consensus_{}'.format(msa_name))
        try:
            os.mkdir(temp_dir)
        except:
            pass
        tempDict = {'dihedrals': {}, 'coordinates': {}, 'contacts': {}, 'distance': {}}
        miDict['consensus']['data'] =

```

```

{msa_name: {'dihedrals': {}, 'coordinates': {}, 'contacts': {}, 'distance': {}}}
for prot in miDict['proteins'].keys():
    if 'merge' not in miDict['proteins'][prot]['sets'].keys():
        print('No averaged data found for protein {}, moving on...'.format(prot))
        continue # Move on if there is no averaged matrix...
    for dType in tempDict.keys():
        if dType in ('coordinates', 'dihedrals'):
            tempDict[dType][ '{}_{}_mi'.format(prot, dType) ] =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices']['mi_mean_rescale']
            try:
                tempDict[dType][ '{}_{}_r'.format(prot, dType) ] =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices']['corcoef_mean_rescale']
            except:
                continue
            try:
                tempDict[dType][ '{}_{}_mi_harm'.format(prot, dType) ] =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices']['mi_harm_mean_rescale']
            except:
                continue
            # tempDict[dType][ '{}_{}_mi_harm'.format(prot, dType) ] =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices']['mi_harm_mean_rescale']
            elif dType in ('distance', 'contacts'):
                tempDict[dType][ '{}_{}'.format(prot, dType) ] =
miDict['proteins'][prot]['sets']['merge']['data'][dType]['matrices']['m_mean']
            else:
                continue
            pickle.dump({'tempDict': tempDict}, open('/Users/kiwhite/Desktop/temp.pkl', 'wb'))
            for dType in tempDict.keys():
                print('Calculating consensus matrix for {} data...'.format(dType))
                if dType in ('coordinates', 'dihedrals'):
                    print(tempDict[dType].items())
                    mi, mi_lookup = consensus_tensor_from_msa({msa_name: msa}, {k: v for k, v in
tempDict[dType].items() if k.split('_')[-1] == 'mi'},
                    dat_type=('{}_MI'.format(dType), 'bits'), dimRedMode='mean', rescale=True,
                    refSeqName=args.refSeqName, runInfo=dType, outDir=temp_dir, plot=args.plot)
                    try:
                        mi_harm, mi_harm_lookup = consensus_tensor_from_msa({msa_name: msa}, {k: v
for k, v in tempDict[dType].items() if k.split('_')[-1] == 'harm'},
                    dat_type=('{}_MIharm'.format(dType), 'bits'), dimRedMode='mean', rescale=True,
                    refSeqName=args.refSeqName, runInfo=dType, outDir=temp_dir, plot=args.plot)
                    except KeyError:
                        continue
                    try:
                        r, r_lookup = consensus_tensor_from_msa({msa_name: msa}, {k: v for k, v in
tempDict[dType].items() if k.split('_')[-1] == 'r'},
                    dat_type=('{}_corcoef'.format(dType), 'corr'), dimRedMode='mean', rescale=True,
                    refSeqName=args.refSeqName, runInfo=dType, outDir=temp_dir, plot=args.plot)
                    except KeyError:
                        continue
                    miDict['consensus']['data'][msa_name][dType][ 'm_{}_full'.format(msa_name) ] =
mi
                    miDict['consensus']['data'][msa_name][dType][ 'm_harm_{}_full'.format(msa_name) ] = mi_harm
                    miDict['consensus']['data'][msa_name][dType][ 'r_{}_full'.format(msa_name) ] =
r
                    miDict['consensus']['data'][msa_name][dType][ 'lookup_{}_consensus'.format(msa_name) ] = mi_lookup
                    elif dType in ('distance', 'contacts'):
                        d_labels = {'distance': ('distance', 'Angstroms'), 'contacts': ('contact',
'probability over MSA')}
                        d, d_lookup = consensus_tensor_from_msa({msa_name: msa}, tempDict[dType],
                        dat_type=d_labels[dType], dimRedMode='mean', rescale=False, refSeqName=args.refSeqName,
                        runInfo=dType, outDir=temp_dir, plot=args.plot)
                        miDict['consensus']['data'][msa_name][dType][ 'm_{}_full'.format(msa_name) ] =
d
                    miDict['consensus']['data'][msa_name][dType][ 'lookup_{}_consensus'.format(msa_name) ] = d_lookup

            if args.plot:

```

```

        plt.close('all')
pickle.dump(miDict, open('/Users/kiwhite/Desktop/temp.pkl', 'wb'))
# 5: Look at matrix correlations
temp_dir = os.path.join(args.outDir, 'cc_{}'.format(msa_name))
try:
    os.mkdir(temp_dir)
except:
    pass
cc_dict, cc_mat = matrix_correlation_coef(miDict, compare_dict = {}, n_permutations=100,
msa_id=msa_name, seq_id_dict = {}, runName=args.runName, plot=args.plot, outDir=temp_dir)

# 6: Look at matrix eigendecomposition, infer dimensionality of collective feature, and
perform SymNMF with NNSVD initialization assuming that many factors.
print('\nPerforming eigendecomposition and SymNMF on all matrices...')
temp_dir_eig = os.path.join(args.outDir, 'eigendecomp')
try:
    os.mkdir(temp_dir_eig)
except:
    pass
temp_dir_symnmf = os.path.join(args.outDir, 'symnmf')
try:
    os.mkdir(temp_dir_symnmf)
except:
    pass
for protID,protDict in miDict['proteins'].items():
    to_structure = False
    try:
        this_pdb = [p for p in pdb_file_list if protID in p][0]
    except IndexError:
        print('\tPDB file for {} not found; will not be able to map eigenvectors or factors
to structure for {} data.'.format(protID,dType))
    else:
        msa_seq = []
        for msa in miDict['consensus']['alignments'].values():
            try:
                msa_seq.append([str(s) for s in msa if protID in s.id][0])
            except IndexError:
                pass
        if len(msa_seq) == 0:
            print('\tCould not find sequence in any supplied MSA corresponding to {}; this is
required to map {} data to structure. Skipping.'.format(protID,dType))
        else:
            if len(msa_seq) > 1:
                print('\tMultiple sequences found for {}, using
{}'.format(protID,msa_seq[0]))
                msa_seq = msa_seq[0]
                to_structure = True
            for dType,dDict in protDict['sets']['merge']['data'].items():
                print('\tDecomposing {} data for {}...'.format(dType, protID))
                if dType in ('coordinates', 'dihedrals'):
                    for mat_id in ('mi_mean_rescale', 'corcoef_mean_rescale'):
                        this_run =
                        '{}_{}_{}_{}'.format(args.runName,protID,dType,mat_id.split('_')[0])
                        try:
                            _, _, n_sig_evals_ln, sig_evecs_ln, n_sig_evals_shuffle,
sig_evecs_shuffle, _, _ = matrix_eigendecomposition(dDict['matrices'][mat_id],
resiList=protDict['resiIndices'], plot=True, runName=this_run, outDir=temp_dir_eig,
shuffle_iter=1000)
                        except KeyError:
                            continue
                        if n_sig_evals_shuffle > 0:
                            symnmf_factors, _, _ =
nonnegative_matrix_factorization(np.abs(dDict['matrices']['mi_mean_rescale']),n_components=n_sig_
evals_shuffle, resiList=protDict['resiIndices'],mode='symnmf-
nnsvd',runName=this_run,plot=True,outDir=temp_dir_symnmf)
                            if to_structure:
                                data_to_structure(this_pdb, sig_evecs_shuffle, msa_seq, temp_dir_eig,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],

```

```

pymol_spectrum='blue white red')
        data_to_structure(this_pdb, symnmf_factors, msa_seq, temp_dir_symnmf,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],
pymol_spectrum='black red orange yellow white')
        elif n_sig_evals_ln > 0:
            symnmf_factors, _, _ =
nonnegative_matrix_factorization(np.abs(dDict['matrices']['mi_mean_rescale']),n_components=n_sig_
evals_ln,resiList=protDict['resiIndices'],mode='symnmf-
nndsvd',runName=this_run,plot=True,outDir=temp_dir_symnmf)
            if to_structure:
                data_to_structure(this_pdb, sig_evecs_ln, msa_seq, temp_dir_eig,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],
pymol_spectrum='blue white red')
                data_to_structure(this_pdb, symnmf_factors, msa_seq, temp_dir_symnmf,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],
pymol_spectrum='black red orange yellow white')
            else:
                print('Found no significant eigenmodes for {} {},
skipping.'.format(protID, dType))
            elif dType == 'distance':
                this_run = '{}_{}_{}'.format(args.runName,protID,dType)
                _, _, n_sig_evals_ln, sig_evecs_ln, n_sig_evals_shuffle, sig_evecs_shuffle, _, _
= matrix_eigendecomposition(dDict['matrices']['m_mean'], resiList=protDict['resiIndices'],
plot=True, runName=this_run, outDir=temp_dir_eig, shuffle_iter=1000)
                symnmf_factors, _, _ =
nonnegative_matrix_factorization(dDict['matrices']['m_mean'],n_components=n_sig_evals_shuffle,res
iList=protDict['resiIndices'],mode='symnmf-
nndsvd',runName=this_run,plot=True,outDir=temp_dir_symnmf)
                if to_structure:
                    data_to_structure(this_pdb, sig_evecs_shuffle, msa_seq, temp_dir_eig,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],
pymol_spectrum='blue white red')
                    data_to_structure(this_pdb, symnmf_factors, msa_seq, temp_dir_symnmf,
resiList=protDict['resiIndices'], chain='A', runName=this_run, pymol_percentile_list=[90],
pymol_spectrum='black red orange yellow white')
                for msaID, msaDict in miDict['consensus']['data'].items():
                    for dType,dDict in msaDict.items():
                        print('\tDecomposing {} data for {}...'.format(dType, '{} MSA
consensus'.format(msaID)))
                        if dType in ('coordinates','dihedrals'):
                            _, _, n_sig_evals_ln, sig_evecs_ln, n_sig_evals_shuffle, sig_evecs_shuffle, _, _
= matrix_eigendecomposition(dDict['m_{}_full'.format(msaID)]['consensus_compressed_rescale'],
resiList=[], plot=True, runName='{}_{}_{}'.format(args.runName,msaID,dType),
outDir=temp_dir_eig, shuffle_iter=1000)
                            if n_sig_evals_shuffle > 0:
                                symnmf_factors, _, _ =
nonnegative_matrix_factorization(np.abs(dDict['m_{}_full'.format(msaID)]['consensus_compressed_re
scale']),n_components=n_sig_evals_shuffle,mode='symnmf-
nndsvd',runName='{}_{}_{}'.format(args.runName,msaID,dType),plot=True,outDir=temp_dir_symnmf)
                                elif n_sig_evals_ln > 0:
                                    symnmf_factors, _, _ =
nonnegative_matrix_factorization(np.abs(dDict['m_{}_full'.format(msaID)]['consensus_compressed_re
scale']),n_components=n_sig_evals_ln,mode='symnmf-
nndsvd',runName='{}_{}_{}'.format(args.runName,msaID,dType),plot=True,outDir=temp_dir_symnmf)
                                else:
                                    print('Found no significant eigenmodes for {} consensus {},
skipping.'.format(msaID, dType))
                            elif dType == 'distance':
                                _, _, n_sig_evals_ln, sig_evecs_ln, n_sig_evals_shuffle, sig_evecs_shuffle, _, _
= matrix_eigendecomposition(dDict['m_{}_full'.format(msaID)]['consensus_compressed'],
resiList=[], plot=True, runName='{}_{}_{}'.format(args.runName,msaID,dType), outDir=temp_dir_eig,
shuffle_iter=1000)
                                if n_sig_evals_shuffle > 0:
                                    symnmf_factors, _, _ =
nonnegative_matrix_factorization(dDict['m_{}_full'.format(msaID)]['consensus_compressed'],n_compo
nents=n_sig_evals_shuffle,mode='symnmf-
nndsvd',runName='{}_{}_{}'.format(args.runName,msaID,dType),plot=True,outDir=temp_dir_symnmf)
                                elif n_sig_evals_ln > 0:
                                    symnmf_factors, _, _ =

```

```

nonnegative_matrix_factorization(dDict['m-{}_full'.format(msaID)]['consensus_compressed'],n_compo
nents=n_sig_evals_ln,mode='symnmf-
nndsvd',runName='{}_{}_{}'.format(args.runName,msaID,dType),plot=True,outDir=temp_dir_symnmf)
    else:
        print('Found no significant eigenmodes for {} consensus {},
skipping.'.format(msaID, dType))

    # X: Save all output
    print('\nPickling...\n')
    with open(os.path.join(args.outDir,'{}_analyze_ermi_output.pkl'.format(args.runName)),'wb')
as outPickle:
        pickle.dump({'miDict':miDict,'ccDict':cc_dict},outPickle)
    print('Done!')

```

1.2.10 multiconf_utilities.py

Utility functions used by different programs in §1.2.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

import os, pickle, glob
import numpy as np
from Bio import PDB
from scipy.stats import shapiro

try:
    from itertools import zip_longest
except ImportError:
    try:
        from itertools import izip_longest as zip_longest
    except:
        raise

def create_ensemble_list_deprecated(ensemble):
    """
    Creates an ensemble of Bio.PDB structures from one or more PDB files. No alignment option
    available.
    input:
        ensemble: Path to a PDB file or a directory containing multiple PDB files.
    Output:
        ensembleList: A list of one or more Bio.PDB structure objects.
    """
    ensembleList = []
    parser = PDB.PDBParser()
    if os.path.isdir(ensemble):
        for pdb in glob.iglob(os.path.join(ensemble,'*.pdb')):
            print('Harvesting models from from {}'.format(pdb))
            ensembleList.append(parser.get_structure(os.path.basename(pdb),pdb))
    elif os.path.isfile(ensemble):
        print('Harvesting models from from {}'.format(ensemble))
        ensembleList.append(parser.get_structure(os.path.basename(ensemble),ensemble))
    return(ensembleList)

def create_ensemble_list(ensemble,superimpose=True,supersele='all',ref=None, in_memory=False,
dumpPDB=False, outDir=None, verbose=False):
    """
    Creates an ensemble of Bio.PDB structures from one or more PDB files. Rather than storing the
    loaded structures in memory, saves them as pickles instead and returns list of pickles.
    Probably the best approach for more than a few ensembles. This is the function used by
    __main__.

```

```

Input:
    ensemble: Path to a PDB file or a directory containing multiple PDB files.
    superimpose: If true, output structures will be aligned to the first model of the first
PDB file specified using Bio.PDB.Superimposer
    supersele: A string specifying which residues to use for superimposing. If '\all\', use
all residues. Otherwise, provide semicolon-separated ranges x:x+n, where x and x+n are consistent
with PDB residue indexing.
    outDir: An optional string specifying location for pickling.
    verbose: If true, say more.
Output:
    ensembleList: A list of one or more paths to pickles containing Bio.PDB structure
objects.
'''
def get_atoms(biopdb_obj, supersele_list, model_id=0, refMode=False, atom_label=['CA']):
    '''
    Get CA atoms consistent with selection list from a structure.
    '''
    if not isinstance(biopdb_obj, PDB.Model.Model):
        try:
            model = biopdb_obj[model_id]
        except KeyError:
            print('Failed to get structure {} from {}'.format(model_id, biopdb_obj.id))
            raise
    else:
        model = biopdb_obj
        atoms = []
        resis = []
        for chain in model:
            for res in chain:
                if supersele_list == ['all'] or any([res.id[1] in range(*supersele_list[i]) for i
in range(len(supersele_list))]):
                    try:
                        [atoms.append(res[a]) for a in atom_label]
                        resis.append(res.id[1])
                    except KeyError:
                        continue
        if refMode and atoms == []:
            raise ValueError('Cannot get atoms from {}! Check model indexing and selection
string.'.format(biopdb_obj))
        return(atoms, resis)

parser = PDB.PDBParser()

if superimpose:
    m=' and superimposing '
    if supersele == 'all':
        m += 'all Calphas'
        supersele_list = ['all']
    else:
        supersele_list=[]
        for range_str in supersele.split(';'):
            if ',' in range_str: # Ignore chain selection
                range_str = range_str.split(',')[1]
            if ':' in range_str: # Residue range
                supersele_list.append([int(i) for i in range_str.split(':')])
                supersele_list[-1][1] += 1
            else: # It's hopefully just an integer
                supersele_list.append([int(range_str)])
        m += 'a subset of Calphas specified by supersele_list {}'.format(supersele_list)
else:
    m=' '

if superimpose and ref is None:
    gotRef = False
elif superimpose and ref is not None:
    if type(ref) is str and os.path.isfile(ref):
        print('Will superimpose everything with first model of structure from
{}'.format(ref))
        refStructure = parser.get_structure(os.path.basename(ref), ref)

```

```

elif isinstance(ref,PDB.Structure.Structure):
    print('Will superimpose everything with first model of structure from
    {}.format(ref.id))
    refStructure = ref
else:
    raise ValueError('Supplied reference {} is not valid!'.forma(ref))
refModel = refStructure[0]
refAtoms, refResis = get_atoms(refModel, supersele_list, refMode=True)
gotRef = True
if isinstance(ensemble, list):
    if not all([os.path.isfile(e) for e in ensemble]):
        raise ValueError('If ensemble is provided as a list, all elements must be valid file
paths.')
    else:
        ens = ensemble
elif os.path.isfile(ensemble):
    ens = (ensemble,)
elif os.path.isdir(ensemble):
    ens = glob.glob(os.path.join(ensemble, '*.pdb'))
else:
    raise ValueError('Invalid PDB file or directory of files specified.')
ensembleList = []
for i,pdb in enumerate(ens):
    pdbPath,pdbFile = os.path.split(pdb)
    pdbBase = os.path.splitext(pdbFile)[0]
    if not in_memory:
        if outDir is None:
            ensembleList.append(os.path.join(pdbPath, '{}.pk1'.format(pdbBase)))
        elif os.path.isdir(outDir):
            ensembleList.append(os.path.join(outDir, '{}.pk1'.format(pdbBase)))
    print('Loading {}...'.format(pdb))
    thisStructure = parser.get_structure(os.path.basename(pdb),pdb)
    print('Harvesting{}for models from from {}...'.format(m,os.path.basename(pdb)))
    if superimpose and not gotRef: # If we're on the first model and a reference structure
was not supplied, create a CA atom list from it to use as reference
        print('Superposing everything with first model of structure from {}'.format(pdb))
        refModel = thisStructure[0]
        refAtoms, refResis = get_atoms(refModel, supersele_list, refMode=True)
        gotRef = True
    if superimpose and gotRef: # If we're on a model other than the reference, align it to
the reference using PDB.Superimposer
        rms_list = []
        for thisModel in thisStructure:
            theseAtoms, theseResis = get_atoms(thisModel, supersele_list)
            if len(refAtoms) == len(theseAtoms):
                super_imposer = PDB.Superimposer()
                super_imposer.set_atoms(refAtoms,theseAtoms)
                if verbose:
                    print('Aligned model {} in {} to reference (RMS = {:.2f}
Angstroms)'.format(thisModel.id,os.path.basename(pdb),np.abs(super_imposer.rms)))
                rms_list.append((pdb,super_imposer.rms))
                super_imposer.apply(thisStructure[thisModel.id].get_atoms())
            else:
                raise ValueError('Number of CA atoms in model {} of current structure is {};
this is different from the number in the reference structure,
{}!\nREFERENCE:\n{}\n{}\n\nCURRENT\n{}\n{}\n'.format(thisModel.id, len(theseAtoms),
len(refAtoms), refStructure.id, refResis, thisStructure.id, theseResis))
        print('Finished superposition, with average RMS over ensemble of {:.4f} +/- {:.4f}
Angstroms.'.format(np.mean([item[1] for item in rms_list]),np.std([item[1] for item in
rms_list])))
    if dumpPDB:
        if outDir is None:
            dump_path = os.path.join(pdbPath, 'aligned_ensembles')
        else:
            dump_path = os.path.join(outDir, 'aligned_ensembles')
        if not os.path.exists(dump_path):
            os.mkdir(dump_path)
        io=PDB.PDBIO()
        io.set_structure(thisStructure)

```

```

pdb_out=os.path.join(dump_path, '{}_{}_'.format(pdbBase,i))
if superimpose:
    if supersele == 'all':
        pdb_out += 'superposed-all.pdb'
    else:
        pdb_out += 'superposed-subset.pdb'
else:
    pdb_out += 'raw.pdb'
print('Saving output PDB:\n{}'.format(pdb_out))
io.save(pdb_out)
del(io)
if not in_memory:
    print('Writing pickle {}... '.format(thisStructure),end='')
    with open(ensembleList[-1], 'wb') as f:
        pickle.dump(thisStructure,f)
    print('done.\n')
    del(thisStructure)
else:
    ensembleList.append(thisStructure)
if superimpose:
    #del(refStructure)
    return(ensembleList,rms_list)
else:
    return(ensembleList)

def filter_pdb_list_rfactor(pdb_list, factor='work',
crop_percentile=('auto',0.8),permissive=False,runName='run',outDir=None,verbose=False):
'''
* Description:

Reduce a list of PDB files such that only those with Rfree below an automatically or
user-specified percentile are kept.
If `crop_percentile` is `auto`, the subfunction `shapiro_auto_crop` will be used to
determine an optimal percentile.

* Input:

* `calculate_ermi`

A list of absolute paths to PDB files.

* `crop_percentile`

`(('auto',percentile_lower_bound), ('manual',crop_percentile))`

A tuple, where the first element is a string that specifies the filter mode and the
second specifies a percentile for that mode.
In `auto` mode, the crop percentile is determined automatically and is no lower than
the float `percentile_lower_bound`.
In `manual` mode, the crop percentile is simply equal to `crop_percentile`.
In both cases, `crop_percentile[1]` must be greater than 0.01 and less than 1.00.

* `permissive`

If `True`, will not throw error when Rfree is not found in a given PDB file.

* `outDir`

A directory in which to write a text file with info about which files were filtered.

* Output:

* `pdb_list`

Reduced list of absolute paths to PDB files.

* `pdb_info_list`

A list of lists, where each sublist has entries `[pdb_file, Rfree,

```

```

included_in_output]`.
'''

def shapiro_auto_crop(data, lower_bound=0.8, verbose=True):
    '''
    * Description:

    Returns the percentile that yields a threshold needed to create the most "normally"
    distributed subset of an array of values; this is assessed using the test statistic `W` from the
    Shapiro-Wilk test for normality.
    Data are taken from largest to smallest values.

    Justification for test over common alternatives:

    Razali, N., Wah, YB. 2011. Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov,
    Lilliefors and Anderson-Darling tests. Journal of Statistical Modeling and Analytics 2 (1) 21-33.

    * Input:

    * `data`

    A list or 1-dimensional `numpy` array of values.

    * `lower_bound`

    Decimal fraction which specifies the maximum amount of data to discard; at the
    default of 0.8, a maximum of 20% of data points will be discarded.

    * `verbose`

    Print information and statistics.

    * Output:

    * `crop_percentile`

    The integer-valued percentile below which to keep data.

    * `found_maxima`

    If `True`, the data were optimally normal with no exculsion of data or a local
    maxima in `W` was identified in the range specified by `lower_bound`.
    If `False`, the "most normal" subset of the data was the one for which as much
    data as possible was excluded.
    '''
    if verbose:
        print('\nDetermining truncation threshold for data using Shapiro-Wilk test for
normality...')
    if lower_bound <= 0. or lower_bound >= 1.0:
        raise ValueError('lower_bound must be greater than 0 and less than 1!')
    test_out = []
    w, p = shapiro(data)
    test_out.append((0, w, p))
    if verbose:
        print('For full dataset, W = {:.4f} and p-val = {}'.format(w, p))
    for i in range(1, 99):
        data_sub = data[data < np.percentile(data, 100-i)]
        if len(data_sub) < lower_bound * len(data) or len(data_sub) < 3:
            break
        w, p = shapiro(data_sub)
        if verbose:
            print('Discard %ile = {: >3}%, W = {:.4f}, Rfactor at %ile = {:.4f}, # models
retained = {}'.format(i, w, np.percentile(data, 100-i), len(data_sub)))
        test_out.append((i, w, p))
    W = [t[1] for t in test_out]
    max_idx = W.index(max(W))
    crop_percentile = 100 - test_out[max_idx][0]
    if max_idx == len(W) - 1:
        found_maxima = False

```

```

else:
    found_maxima = True
    if verbose:
        print('Optimal if worst {}% of data are discarded; W, p-val = {:.4f},
{:0.4f}'.format(100-crop_percentile,test_out[max_idx][1],test_out[max_idx][2]))
        if not found_maxima:
            print('Local maximum not found; it would be prudent to double-check Rfree
distribution.')
```

```

    print('\n')
    return(crop_percentile,found_maxima)

factor = factor.lower()
if not isinstance(factor, str) or factor not in ('free','work'):
    raise ValueError('factor must be a string equal to \'free\' or \'work\'!')
if len(pdb_list)<10:
    raise ValueError('pdb_list must have at least 10 entries.')
```

```

if outDir == None:
    outDir = os.getcwd()
pdb_info_list = []
for this_pdb in pdb_list:
    with open(this_pdb,'r') as f:
        for line in f:
            if 'R VALUE          (WORKING SET)' in line:
                pdb_info_list.append([this_pdb,float(line.strip(' ').split(' ')[-1])])
            if 'FREE R VALUE' in line:
                pdb_info_list[-1].append(float(line.strip(' ').split(' ')[-1]))
                break
            elif 'ATOM' in line[0:3]:
                if permissive:
                    print('Failed to find R factors in {}, moving on...'.format(this_pdb))
                else:
                    raise ValueError('Failed to find R factors in {}!'.format(this_pdb))
```

```

stats = {'free':np.array([r[2] for r in pdb_info_list]), 'work':np.array([r[1] for r in
pdb_info_list])}
#[print('R{:}\n{:}\n\n'.format(k,v)) for k,v in stats.items()]
if type(crop_percentile) != tuple:
    raise ValueError('crop_percentile must be a tuple with format (\'mode\',\'percentile\'),
where mode is a string in (\'auto\',\'manual\') and percentile is a float between 0.01 and
1.00!')
```

```

elif crop_percentile[1] < 0.01 or crop_percentile[1] > 1.00 or type(crop_percentile[1]) is
not float:
    raise ValueError('crop_percentile[1] is {} with type {}, but it must be a float between
0.01 and 1.0!'.format(crop_percentile[1],type(crop_percentile[1])))
elif crop_percentile[0] == 'auto':
    if verbose:
        print('Automatically reducing using Shapiro-Wilk test for normality with default
lower percentile bound.')
```

```

    crop_percentile, found_maxima =
shapiro_auto_crop(stats[factor],lower_bound=crop_percentile[1])
    auto = True
elif crop_percentile[0] == 'manual':
    crop_percentile = crop_percentile[1]
    crop_percentile = int(100*crop_percentile)
    auto = False
cutoff = np.percentile(stats[factor],crop_percentile) # Omit the worst (100 -
crop_percentile) % of ensembles.
if auto:
    if found_maxima:
        msg = 'Automatically determined threshold using Shapiro-Wilk test for normality.'
```

```

    else:
        msg = 'Shapiro-Wilk test for normality failed to yield a local minima over permitted
percentile search range.'
```

```

    msg = msg + '\n{} ensembles with R{} less than {:.4f} ({}th percentile)
retained.'.format(np.sum(stats[factor]<cutoff),factor,cutoff,int(crop_percentile))
    print(msg)
    r_work_keep = []
    r_free_keep = []
for i,entry in enumerate(pdb_info_list):
    if ((factor == 'free' and entry[2] > cutoff) or (factor == 'work' and entry[1] > cutoff))
```

```

and cutoff < 100:
    pdb_info_list[i].append(False)
    pdb_list.pop(pdb_list.index(entry[0]))
else:
    pdb_info_list[i].append(True)
    r_work_keep.append(pdb_info_list[i][1])
    r_free_keep.append(pdb_info_list[i][2])
r_stats = (np.mean(r_work_keep), np.std(r_work_keep), np.mean(r_free_keep), np.std(r_free_keep))
stats_msg = 'Super-ensemble Rwork = {:.4f} +/- {:.4f}\nSuper-ensemble Rfree = {:.4f} +/-
{:.4f}\n'.format(*r_stats)
print(stats_msg)
if outDir != None:
    with open(os.path.join(outDir, '{}_rfactor_analysis.txt'.format(runName)), 'w') as f:
        f.write(msg+'\n')
        f.write(stats_msg)
        for entry in pdb_info_list:
            f.write('{}\t{}\t{}\t{}\n'.format(*entry))
return(pdb_list, pdb_info_list)

def get_filtered_pdb_list(summary_file, keep_path=False):
    """
    Gets information from txt output of multiconf_utilities.filter_pdb_list_rfator
    """
    pdb_list_good = []
    pdb_list_bad = []
    with open(summary_file, 'r') as fi:
        for line in fi:
            line = line.rstrip().strip(' ').split()
            if line[-1] == 'True':
                if keep_path:
                    pdb_list_good.append(line[0])
            else:
                pdb_list_good.append(os.path.split(line[0])[1])
            elif line[-1] == 'False':
                if keep_path:
                    pdb_list_bad.append(line[0])
            else:
                pdb_list_bad.append(os.path.split(line[0])[1])
    return({'good':pdb_list_good, 'bad':pdb_list_bad})

def get_resi_coord_slice_for_prody_ensemble(resi_numbers, resi_selection, atom_names=None,
atom_selection='all', split_by_resi=True):
    """
    returns a slice or slices (as array or list of arrays) for the second index of a prody
    coordinate ensemble, i.e., ensemble.getCoordsets()[:,mat_slice,:]
    """
    if not isinstance(resi_selection, list):
        resi_selection=sorted([resi_selection])
    else:
        resi_selection=sorted(resi_selection)
    if len(resi_selection) == 0:
        return([])
    if atom_names is not None and isinstance(atom_selection, list) and not all([isinstance(l, list)
for l in atom_selection]):
        if any([isinstance(l, list) for l in atom_selection]):
            raise ValueError('atom_selection must be equal to \'all\', a single string (e.g.,
\'CA\') to apply to all residues in resi_selection, a list of strings (e.g., [\'CA\', \'CB\']) to
apply to all residues in resi_selection, or a list of lists of atom selections, where the main
list is equal in length to resi_selection and the sublists specify the atoms to retain for each
residue.')
        else:
            n_atoms = len(atom_selection)
            atom_selection *= len(resi_selection)
            atom_selection = [atom_selection[x:x+n_atoms] for x in
range(0, len(atom_selection), n_atoms)]
    elif isinstance(atom_selection, str) and atom_selection is not 'all':
        atom_selection = [atom_selection]*len(resi_selection)
    if atom_names is not None and atom_selection != 'all' and len(resi_selection) !=

```

```

len(atom_selection):
    raise ValueError('Length of resi_selection ({} must be equal to length of atom_names
({})! This can be made possible in the future if
needed...'.format(len(resi_selection),len(atom_selection)))
    if isinstance(atom_names,list) and len(resi_numbers) != len(atom_names):
        raise ValueError('Number of residue indices must equal the number of atom names!')
    if not all(r in resi_numbers for r in resi_selection):
        raise ValueError('Invalid residue selection specified!')
    if atom_names is None or atom_selection == 'all':
        mat_slice = [np.where(resi_numbers==sele)[0] for sele in resi_selection]
        if not split_by_resi:
            mat_slice = np.hstack(mat_slice)
        return(mat_slice)
    else:
        mat_slice=None
        for sele, atom_subsel in zip(resi_selection,atom_selection):
            if not isinstance(atom_subsel,list):
                atom_subsel=[atom_subsel]
            idx_set = np.where(resi_numbers==sele)[0]
            idx_set = idx_set[sorted(atoms_subsel))]
            if mat_slice is None:
                mat_slice=idx_set
                if split_by_resi:
                    mat_slice=[mat_slice]
            else:
                if split_by_resi:
                    mat_slice.append(idx_set)
                else:
                    mat_slice=np.append(mat_slice,idx_set)
        return(mat_slice)

def dict_merge(dct, merge_dct):
    """ Recursive dict merge. Inspired by :meth:`dict.update()` , instead of
    updating only top-level keys, dict_merge recurses down into dicts nested
    to an arbitrary depth, updating keys. The `merge_dct` is merged into
    `dct`. From https://gist.github.com/angstwad/bf22d1822c38a92ec0a9.
    :param dct: dict onto which the merge is executed
    :param merge_dct: dct merged into dct
    :return: None
    """
    for k, v in merge_dct.items():
        if (k in dct and isinstance(dct[k], dict) and isinstance(merge_dct[k], dict)):
            dict_merge(dct[k], merge_dct[k])
        else:
            dct[k] = merge_dct[k]

def parse_selection_string(sele_str, full_tuple=True):
    """
    Converts a selection string to a list of integers spanning that selection.
    For example, string '2:4,8:12' is converted list to [2, 3, 4, 8, 9, 10, 11, 12].
    """
    # chain = None
    resi_sublist=[]
    if '=' in sele_str:
        group_name, sele_str = sele_str.split('=')
    else:
        group_name = sele_str
    for range_str in sele_str.split(';'):
        chain = None
        if ',' in range_str:
            chain, range_str = range_str.split(',')
        if ':' in range_str:
            temp_range = [int(i) for i in range_str.split(':')]
            temp_range[1] += 1
            if full_tuple:
                resi_sublist.append(list([(chain,i) for i in range(*temp_range)]))
            else:
                resi_sublist.append(list(range(*temp_range)))

```

```

    else:
        if full_tuple:
            resi_sublist.append([(chain, int(range_str))])
        else:
            resi_sublist.append([int(range_str)])
    resi_sublist = [item for sublist in resi_sublist for item in sublist]
    return(group_name, resi_sublist)

def mat_to_txt(mat, index_labels, full_path, header=None, start_diagonal=0):
    """
    """
    triu_idx = np.triu_indices_from(mat, start_diagonal)
    n_ele = len(triu_idx[0])
    flat = np.zeros((n_ele, 3))
    for i in range(n_ele):
        j, k = triu_idx[0][i], triu_idx[1][i]
        flat[i, :] = [j, k, mat[j, k]]
    flat = flat[np.argsort(flat[:, 2])[:, :-1]]
    try:
        with open(os.path.join(full_path), 'w') as f:
            if header is not None:
                f.write('# {}\n'.format(header))
            f.write(' i\t j\t ri\t rj\t val\n')
            [f.write('{: 4.0f}\t{: 4.0f}\t{: >8}\t{: >8}\t{: 4.6f}\n'.format(i, j, index_labels[int(i)], index_labels[int(j)], m)) for i, j, m in flat]
    except:
        print('Failed to write matrix data to text file!')
        raise

def grouper(iterable, n, padvalue=None):
    """
    From http://stackoverflow.com/a/312644
    grouper('abcdefg', 3, 'x') --> ('a','b','c'), ('d','e','f'), ('g','x','x')
    """
    return zip_longest(*[iter(iterable)]*n, fillvalue=padvalue)

def symm_roll(M, roll):
    """
    """
    if type(roll) is int and roll != 0:
        M_roll = np.roll(np.roll(M, roll, axis=0), roll, axis=1)
        return(M_roll)
    elif type(roll) is not int:
        raise ValueError('roll must be an integer!')
    elif roll == 0:
        print('roll is zero, doing nothing...')
        return(M)

def triu_corr_coef(A, B, diag=1):
    """
    Returns the correlation coefficient between the upper triangles of two matrices.
    """
    if A.ndim > 1:
        At = A[np.triu_indices_from(A, diag)]
    else:
        At = A
    if B.ndim > 1:
        Bt = B[np.triu_indices_from(B, diag)]
    else:
        Bt = B
    try:
        corr_coef = np.corrcoef(At, Bt)[0, 1]
    except ValueError:
        print('At shape = {}\nBt shape = {}\n'.format(At.shape, Bt.shape))
        raise
    return(corr_coef)

def triu_cosine(A, B, diag=1):

```

```

'''
Returns the cosine of the angle between the upper triangles of two matrices.
'''
if A.ndim > 1:
    At = A[np.triu_indices_from(A,diag)]
else:
    At = A
if B.ndim > 1:
    Bt = B[np.triu_indices_from(B,diag)]
else:
    Bt = B
try:
    cosine=np.dot(At,Bt)/np.linalg.norm(At)/np.linalg.norm(Bt)
except:
    print('At shape = {}\nBt shape = {}'.format(At.shape,Bt.shape))
    raise
return(cosine)

def simple_rescale(v):
'''
Rescales data in vector v to fall between 0 and 1, with minimum at 0 and maximum at 1.
'''
return((v-v.min())/(v.max()-v.min()))

def cmap_bounds(M,perc=None):
'''
Keeps white at 0 in a map like RdBu.
'''
if perc is None or type(perc) not in (int, float) or perc < 0:
    bounds = [-max(np.abs(M.min()),np.abs(M.max()))]
else:
    bounds = [-max(np.abs(np.percentile(M,100-perc)),np.abs(np.percentile(M,perc)))]
bounds.append(-bounds[0])
return(bounds)

def sort_crop_eigs(evals, evecs, keep=10, abs_evals=True, verbose=False):
'''
'''
if not isinstance(evals,np.ndarray) or not isinstance(evecs,np.ndarray):
    raise ValueError('Eigenvalues and eigenvectors must be numpy ndarrays.')
if verbose:
    print('Input eigenvalues:\n{}'.format(evals))
if abs_evals:
    evals = np.abs(evals)
idx = evals.argsort()[::-1][:keep]
evals_sort = evals[idx]
evecs_sort = evecs[:,idx]
if verbose:
    print('Top {} eigenvalues:\n{}'.format(keep,evals_sort))
return(evals_sort, evecs_sort)

def calc_rms(M, evecs = False):
'''
By default, creates an n x n matrix of RMS values from a 3n x 3n matrix, where elements are
in order [a1, a2, a3, b1, b2, b3, ...].
If M is an eigenvector matrix, RMS is taken only down eigenvector elements.
'''
if len(M.shape) == 1:
    M_stack = np.dstack((M[::3],M[1::3],M[2::3]))
elif evecs:
    M_stack = np.dstack((M[::3,:],M[1::3,:],M[2::3,:]))
else:
    M_stack = np.dstack((M[::3,::3],M[1::3,1::3],M[2::3,2::3]))
M_rms = np.sqrt(np.sum(np.square(M_stack), axis=2))
return(M_rms)

def cov_to_corr(cov):
'''
Converts a covariance matrix to a correlation coefficient matrix.
'''

```

```

    ...
    corr = np.zeros_like(cov)*0
    for i,j in np.nditer(np.triu_indices_from(cov)):
        corr[i,j] = cov[i,j]/(np.sqrt(cov[i,i])*np.sqrt(cov[j,j]))
    corr += np.tril(corr.T,-1)
    return(corr)

def residual_variance(evals,abs_val=True):
    ...
    ...
    n_evals = evals.shape[0]
    res_var = np.ndarray(n_evals-1)
    if abs_val:
        evals = np.abs(evals)
    evals = evals[evals.argsort()[::-1]]
    for i,e in enumerate(evals):
        if i < n_evals - 1:
            res_var[i] = e/np.sum(evals[i:])
        else:
            return(res_var)

def kth_diag_indices(a, k):
    ...
    Get indices for the kth diagonal of a.

    Thought this would be a built-in, but found out that it wasn't... so code from Hans Then:
    http://stackoverflow.com/questions/10925671/numpy-k-th-diagonal-indices
    ...
    rows, cols = np.diag_indices_from(a)
    if k < 0:
        return(rows[:k], cols[-k:])
    elif k > 0:
        return(rows[k:], cols[:-k])
    else:
        return(rows, cols)

def decay_func(x,a,b):
    ...
    y=a/x+b
    ...
    return((a/(x))+b)

def exp_func(x,a,b,c):
    ...
    y=a*e^(b*x)+c
    ...
    return(a*np.exp(b*x)+c)

def exp_func_b(x,a):
    ...
    ...
    return(1-np.exp((x-1)/a))

def double_exp_func(x,a,b,c,d,e):
    ...
    ...
    return(a*np.exp(-b*x)+c*np.exp(-d*x)+e)

```

1.3 arduino_control_center_v2

1.3.1 arduino_control_center_v2.ino

The Arduino sketch which controls pump state, power supply voltage, etc. in response to commands sent over a network. A work in progress.

```

#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUdp.h>
#include <Wire.h>
#include <Adafruit_MCP4725.h>

// MCP4725 stuff
Adafruit_MCP4725 dac;

// Vctl monitor
#define N_SAMPLES 10

// Internet config
// box v1
// byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0x3E, 0xF3 };
// box v2
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0F, 0x2B, 0xA9 };
// IPAddress ip(129, 112, 108, 252); // UTSW
IPAddress ip(164, 54, 161, 89); // ANL BioCARS
// IPAddress ip(192, 168, 2, 2); // Local
unsigned int localPort = 8887; // local port to listen on

// buffers for receiving and sending data
char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold incoming packet,
char replyBuffer[UDP_TX_PACKET_MAX_SIZE]; // a string to send back

// An EthernetUDP instance
EthernetUDP Udp;

// Hardware control stuff
int PUMP = 3; // Pump control by PWM
int MINIPUMP = 5; // Pump control by PWM
int RESET = 8; // Reset
int INHIBIT = 7; // Inhibit
int ILED = 48;
int SLED = 49; // Device online?
int CLED = 50; // Comm loop
int PLED = 51; // Pump on?
int VLED = 52; // Voltage status
int ELED = 53; // Device error?

// Hardware variables
char pumpSpeed[3];
char pumpTime[5];
char voltComm[10];
uint32_t volt = 0;
//int sample_sum = 0;
//unsigned char sample_count = 0;

// Setup
void setup() {
  digitalWrite(RESET, HIGH);
  delay(200);
  Ethernet.begin(mac, ip);
  Udp.begin(localPort);
  Serial.begin(9600);
  Serial.print("\nCONTROL CENTER ONLINE @ ");
  Serial.println(Ethernet.localIP());
  pinMode(ILED, OUTPUT);
  pinMode(PLED, OUTPUT);
  pinMode(SLED, OUTPUT);
  pinMode(ELED, OUTPUT);
  pinMode(VLED, OUTPUT);
  pinMode(CLED, OUTPUT);
  pinMode(PUMP, OUTPUT);
  pinMode(MINIPUMP, OUTPUT);
  pinMode(INHIBIT, OUTPUT);
  pinMode(RESET, OUTPUT);
}

```

```

digitalWrite(ILED, LOW);
digitalWrite(SLED, HIGH);
digitalWrite(PLED, LOW);
digitalWrite(ELED, LOW);
digitalWrite(CLED, LOW);
int mcptest = analogRead(A0);
if (mcptest > 0)
{
  Serial.println("MCP4725 is active!");
  volt = mcptest * 4 + 3;
  digitalWrite(VLED, HIGH);
}
else
{
  digitalWrite(VLED, LOW);
}
digitalWrite(INHIBIT, LOW);
analogWrite(PUMP, 0);
analogWrite(MINIPUMP, 0);
// For Adafruit MCP4725A1 the address is 0x62 (default) or 0x63 (ADDR pin tied to VCC)
// For MCP4725A0 the address is 0x60 or 0x61
// For MCP4725A2 the address is 0x64 or 0x65
dac.begin(0x62);
// dac.setVoltage(0, false);
}

// Main
void loop()
{
  // if there's data available, read a packet
  int packetSize = Udp.parsePacket();
  if (packetSize)
  {
    digitalWrite(CLED, HIGH);
    Serial.println(' ');
    Serial.print("\nReceived packet of size ");
    Serial.println(packetSize);
    Serial.print("Sender: ");
    IPAddress remote = Udp.remoteIP();
    for (int i = 0; i < 4; i++)
    {
      Serial.print(remote[i], DEC);
      if (i < 3)
      {
        Serial.print(".");
      }
    }
    Serial.print(":");
    Serial.println(Udp.remotePort());

    // Read the packet into packetBuffer
    Udp.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
    Serial.print("UDP packet contents: ");
    Serial.println(packetBuffer);

    // Convert the message recieved in packetBuffer into a command
    int charIndex = 0;
    int varIndex = 0;
    bool miniComm = false;
    char thisCommand = ' ';
    int i;
    for ( i = 0; i < sizeof(packetBuffer) - 1; i++ )
    {
      if ( packetBuffer[i] == ',' )
      {
        charIndex = 0;
        varIndex++;
      }
      else if ( packetBuffer[i] == '\n' )

```

```

{
  charIndex = 0;
  varIndex = 0;
}
else
{
  if ( varIndex == 0 )
  {
    if ( packetBuffer[i] == 'x' ) // This kills everything.
    {
      thisCommand = 'x';
      break;
    }
    else if ( packetBuffer[i] == 'p' || packetBuffer[i] == 'P' ) // This is a pump command.
    {
      thisCommand = 'p';
      if ( packetBuffer[i] == 'p' )
      {
        miniComm = true;
      }
      varIndex++;
    }
    else if ( packetBuffer[i] == 'v' ) // This is a voltage control command.
    {
      thisCommand = 'v';
      varIndex++;
    }
    else if ( packetBuffer[i] == 's' ) // This is a request for device state.
    {
      thisCommand = 's';
      varIndex++;
    }
    else if ( packetBuffer[i] == 'r' ) // This is a reset message.
    {
      thisCommand = 'r';
    }
    else if ( packetBuffer[i] == 't' ) // This is a test message.
    {
      thisCommand = 't';
      break;
    }
    else if ( packetBuffer[i] == 'i' ) // Turn off inhibit
    {
      thisCommand = 'i';
      break;
    }
    else if ( packetBuffer[i] == 'I' ) // Turn on inhibit
    {
      thisCommand = 'I';
      break;
    }
    // New command cases here.
  }
  else {
    if ( thisCommand == 'p' )
    {
      if ( varIndex == 2 )
      {
        pumpSpeed[charIndex] = packetBuffer[i];
        charIndex++;
      }
      else if ( varIndex == 3 )
      {
        pumpTime[charIndex] = packetBuffer[i];
        charIndex++;
      }
    }
  }
  if ( thisCommand == 'v' )
  {

```

```

    if ( varIndex == 2 )
    {
        if ( packetBuffer[i] == 'n' )
        {
            voltComm[charIndex] = 'n';
            break;
        }
        else if ( packetBuffer[i] == 'x' )
        {
            voltComm[charIndex] = 'x';
            break;
        }
        else
        {
            charIndex++;
        }
    }
    else if ( varIndex == 3 )
    {
        voltComm[charIndex] = packetBuffer[i];
        charIndex++;
    }
    else
    {
        break;
    }
}
}
}
packetBuffer[i] = 0;
}
digitalWrite(CLED, LOW);
// Prepare messages array
char* messages[30];
messages[0] = "success";
int msgIdx = 1;

// Perform action based on received message
if ( thisCommand == 'p' )
{
    float s = atof(pumpSpeed);
    float t = atof(pumpTime);
    if ( s > 100.0 || s < 0.0 )
    {
        s = 100.0;
    }
    if ( s > 0 && t > 0 )
    {
        analogWrite(PUMP, 0);
        Serial.print("Running pump at ");
        Serial.print(s);
        Serial.print("% for ");
        Serial.print(t);
        Serial.println(" seconds.");
        digitalWrite(PLED, HIGH);
        if ( miniComm == false )
        {
            analogWrite(PUMP, (s / 100.0) * 205 + 50);
            delay(t * 1000);
            analogWrite(PUMP, 0);
        }
        else
        {
            analogWrite(MINIPUMP, (s / 100.0) * 205 + 50);
            delay(t * 1000);
            analogWrite(MINIPUMP, 0);
        }
    }

    digitalWrite(PLED, LOW);
}

```

```

    Serial.print("Now, ");
    messages[msgIdx++] = "PUMP_SPEED";
    messages[msgIdx++] = "int";
    messages[msgIdx++] = "\n";
}
else if ( s > 0 && t < 0.01 )
{
    Serial.print("Running pump at ");
    Serial.print(s);
    Serial.println("% continuously; send 'p,0' or 'x' to cancel.\nNow, ");
    digitalWrite(PLED, HIGH);
    if ( miniComm == false )
    {
        analogWrite(PUMP, (s / 100.0) * 205 + 50);
    }
    else
    {
        analogWrite(MINIPUMP, (s / 100.0) * 205 + 50);
    }
    messages[msgIdx++] = "PUMP_SPEED";
    messages[msgIdx++] = "int";
    messages[msgIdx++] = "\n";
}
else if ( s == 0 )
{
    Serial.println("Stopping pump.");
    t = 0;
    if ( miniComm == false )
    {
        analogWrite(PUMP, 0);
    }
    else
    {
        analogWrite(MINIPUMP, 0);
    }
    digitalWrite(PLED, LOW);
    messages[msgIdx++] = "PUMP_SPEED";
    messages[msgIdx++] = "int";
    messages[msgIdx++] = "\n";
}
memset(pumpSpeed, 0, sizeof(pumpSpeed));
memset(pumpTime, 0, sizeof(pumpTime));
}
else if ( thisCommand == 'v' )
{
    Serial.print("Received voltage command ");
    Serial.print(voltComm);
    Serial.println(".");
    int mcpvolt = analogRead(A0);
    volt = mcpvolt * 4 + 3;
    Serial.print("Voltage signal from MCP4725 is currently ");
    Serial.print(mcpvolt);
    Serial.print(" bits, which scales to ");
    Serial.print(volt);
    Serial.print(" bits and is equivalent to ");
    Serial.print((volt / 4095.0) * 5.0);
    Serial.println(" V.");
    messages[msgIdx++] = "VSTART";
    messages[msgIdx++] = "int";
    if (voltComm[0] == 'n') // Ramp to minimum
    {
        Serial.println("Ramping voltage output down to 0.");
        vdown(volt, 0);
        messages[msgIdx++] = "VSTOP";
        messages[msgIdx++] = "0";
        //volt = 0;
    }
    else if (voltComm[0] == 'x') // Ramp to maximum
    {

```

```

    Serial.println("Ramping voltage output up to 4095.");
    vup(volt, 4095);
    messages[msgIdx++] = "VSTOP";
    messages[msgIdx++] = "4095";
    //volt = 4095;
}
else // Ramp up/down to value
{
    int vf = atoi(voltComm);
    if (volt < vf)
    {
        Serial.print("Ramping voltage output up to ");
        Serial.print(vf);
        Serial.println(".");
        vup(volt, vf);
        messages[msgIdx++] = "VSTOP";
        messages[msgIdx++] = "int";
        //volt = vf;
    }
    else if (volt > vf)
    {
        Serial.print("Ramping voltage output down to ");
        Serial.print(vf);
        Serial.println(".");
        vdown(volt, vf);
        messages[msgIdx++] = "VSTOP";
        messages[msgIdx++] = "int";
        //volt = vf;
    }
    else
    {
        Serial.println("Target voltage is not different, doing nothing.");
        messages[msgIdx++] = "VSTOP";
        messages[msgIdx++] = "int";
    }
}
float CHECKVOLT = analogRead(A0);
if (CHECKVOLT > 0)
{
    digitalWrite(VLED,HIGH);
}
else
{
    digitalWrite(VLED,LOW);
}
messages[msgIdx++] = "\n";
memset(voltComm, 0, sizeof(voltComm));
}
else if ( thisCommand == 't' )
{
    Serial.print("Received test message; ");
    messages[msgIdx++] = "TEST_STATUS"; // in the future, could add something to actually test
hardware
    messages[msgIdx++] = "boolean";
    messages[msgIdx++] = "\n";
}
else if ( thisCommand == 's' )
{
    Serial.print("Received request for device status; ");
    messages[msgIdx++] = "STATUS UPDATES PENDING DELAZIFICATION";
    messages[msgIdx++] = "\n";
}
else if ( thisCommand == 'x' )
{
    Serial.println("Turning everything off; ");
    analogWrite(PUMP, 0);
    digitalWrite(PLED, LOW);
    memset(pumpSpeed, 0, sizeof(pumpSpeed));
    memset(pumpTime, 0, sizeof(pumpTime));
}

```

```

}
else if (thisCommand == 'r')
{
  Serial.println("Resetting...\n\n");
}
else if (thisCommand == 'i')
{
  Serial.println("Disabling power supply inhibit.");
  digitalWrite(INHIBIT,LOW);
  digitalWrite(ILED,LOW);
  messages[msgIdx++] = "INHIBIT";
  messages[msgIdx++] = "bool";
  messages[msgIdx++] = "\n";
}
else if (thisCommand == 'I')
{
  Serial.println("Enabling power supply inhibit.");
  digitalWrite(INHIBIT,HIGH);
  digitalWrite(ILED, HIGH);
  messages[msgIdx++] = "INHIBIT";
  messages[msgIdx++] = "bool";
  messages[msgIdx++] = "\n";
}
else
{
  Serial.println("Unrecognized command; ");
  digitalWrite(ELED, HIGH);
  delay(500);
  digitalWrite(ELED, LOW);
  delay(500);
  digitalWrite(ELED, HIGH);
  delay(500);
  digitalWrite(ELED, LOW);
}

// send a reply, to the IP address and port that sent us the packet we received
int buffIdx = 0;
for (int m = 0; m <= 10; m++)
{
  if (messages[m] != "\n")
  {
    for (int n = 0; n <= 20; n++)
    {
      if (messages[m][n] == '\0')
      {
        replyBuffer[buffIdx++] = ',';
        break;
      }
      else
      {
        replyBuffer[buffIdx++] = messages[m][n];
      }
    }
  }
  else
  {
    break;
  }
}
replyBuffer[buffIdx - 1] = '\0';

// Send message via UDP
Serial.print("Replying with message \");
Serial.print(replyBuffer);
Serial.println("\n.");
Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
Udp.write(replyBuffer);
Udp.endPacket();
memset(replyBuffer, 0, sizeof(replyBuffer));

```

```

    Serial.print("Message sent.");
    // If requested, reset the device after responding.
    if (thisCommand == 'r')
    {
        digitalWrite(RESET, LOW);
    }
    delay(10);
}
}

void vdown(int volt, int vfloor)
{
    while (volt > vfloor)
    {
        volt = volt - 1;
        if (volt > vfloor)
        {
            dac.setVoltage(volt, false);
        }
        else
        {
            dac.setVoltage(vfloor, false);
        }
        delay(5);
    }
}

void vup(int volt, int vceil)
{
    while (volt < vceil)
    {
        volt = volt + 1;
        if (volt < vceil)
        {
            dac.setVoltage(volt, false);
        }
        else
        {
            dac.setVoltage(vceil, false);
        }
        delay(5);
    }
}
}

```

1.3.2 arduino_control_center.py

A Python-based command line interface for sending commands to the Arduino over a network.

```

#!/usr/bin/env python

from __future__ import print_function
from __future__ import with_statement
from __future__ import division

__author__ = 'Kristopher I. White'
__email__ = 'kristopher.white@utsouthwestern.edu'
__license__ = 'Creative Commons Attribute By - http://creativecommons.org/licenses/by/3.0/us/'

import os, sys, socket
from time import sleep

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'

```

```

BLUE = '\033[94m'
GREEN = '\033[92m'
YELLOW = '\033[93m'
RED = '\033[91m'
GREY = '\033[90m'
BOLD = '\033[1m'
UNDERLINE = '\033[4m'
END = '\033[0m'

def comm_help():
    print(color.BOLD+color.RED+'\n\nThe following commands and arguments are
supported:' +color.END)
    print('\n'+color.UNDERLINE+'PUMP CONTROL'+color.END+'\n'+color.BOLD+'p'+color.END+', '+'s,
t\n\nThe argument p defines this as the pump control command, and arguments s and t set the pump
speed (0-100%) and pump pulse time (seconds), respectively.\n\nSetting s equal to 0 shuts off the
pump, and setting t equal to 0 or omitting it runs the pump continuously.')
    print('\n'+color.UNDERLINE+'POWER SUPPLY VOLTAGE'+color.END+'\n'+color.BOLD+'v'+color.END+',
'+'[min, max, #]', t\n\nThe first argument v defines this as the voltage control command, and the
second argument specifies where to move the power supply voltage. Specifying '\min\' or '\max\'
automatically ramps the supply to highest or lowest available voltage, while specifying a number
will move the supply to that value.')
    print('\n'+color.UNDERLINE+'INHIBIT'+color.END+'\n'+color.BOLD+'i'+color.END+', '+'[on,
off]\n\nToggles power supply inhibit. If '\off\'', inhibit will be disabled and power supply will be
enabled; if '\on\'', inhibit will be enabled and power supply will be disabled.')
    print('\n'+color.UNDERLINE+'STATUS REPORT'+color.END+'\n'+color.BOLD+'s'+color.END+'\n\nThe s
command returns the current state of all relevant pins of the Arduino.')
    print('\n'+color.UNDERLINE+'UPDATE'+color.END+'\n'+color.BOLD+'u'+color.END+'\n\nThe u command
is used to update the networking settings needed to connect to the Arduino.')
    print('\n'+color.UNDERLINE+'RESET'+color.END+'\n'+color.BOLD+'r'+color.END+'\n\nThe r command
reboots the Arduino.')
    print('\n'+color.UNDERLINE+'EXIT'+color.END+'\n'+color.BOLD+'x'+color.END+'\n\nThe x command
exits the CLI.')

def get_net_info():
    UDP_IP = ''
    while UDP_IP == '':
        UDP_IP = input('\n\nEnter the IP address of the Arduino: ')
    try:
        UDP_port = int(input('Enter the UDP port for communication (8887): '))
    except:
        UDP_port = 8887
    # try:
    #     UDP_buffer = int(input('Enter UDP buffer size (1024): '))
    # except:
    UDP_buffer = 1024
    return(UDP_IP, UDP_port, UDP_buffer)

def send_message(UDP_IP, UDP_port, UDP_buffer,message):
    print('\n\nSending message \''+color.BLUE+str(message)+color.END+'\' to Arduino...',end = ' ')
    conn = False
    dat = ''
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
        sock.bind(('', UDP_port))
        sock.sendto(str(message).encode(), (UDP_IP, UDP_port))
        while True:
            try:
                dat, addr = sock.recvfrom(UDP_buffer) # buffer size is 1024 bytes
            except:
                print('failed to receive UDP message.')
                break
            dat=dat.decode()
            if 'success' in dat:
                print('success!')
                conn = True
                break
            elif dat == '':
                print('failed.')
                conn = False
                break

```

```

    if conn == True:
        print(color.GREY+"Data received: "+str(dat)+color.END)
        return(conn,dat)

# Main
if __name__ == "__main__":
    print('\n'+color.BOLD+color.RED+'Arduino UDP CLI'+color.END)
    UDP_IP, UDP_port, UDP_buffer = get_net_info()
    connected,data = send_message(UDP_IP, UDP_port, UDP_buffer,'t')
    alive = True # This flag keeps the interface alive
    vconfig = False
    bits = 4095
    while alive == True:
        comm =
input('\n['+color.PURPLE+'p'+color.END+', '+color.PURPLE+'v'+color.END+', '+color.GREEN+'i'+color.E
ND+',s,u,r,x,?]: ').lower().split(',')
        comm = [item.strip() for item in comm]
        if connected == True and (comm[0] == 'p' or comm[0] == 'pump'):
            print('\nUpdating pump status ->', end=' ')
            if 'mini' in comm:
                comm.remove('mini')
                comm[0] = 'p'
            else:
                comm[0] = 'p'
            print(comm)
            if len(comm) == 1:
                comm = ['P', '100', '0']
            elif len(comm) == 3:
                pass # Need to add some stuff here
            elif len(comm) == 2:
                comm.append('0')
            else:
                print('Incorrect number of arguments for pump command received.')
                continue
            send_message(UDP_IP,UDP_port,UDP_buffer,str(comm[0]+' '+comm[1]+' '+comm[2].strip()))

        elif connected == True and (comm[0] == 'v' or comm[0] == 'voltage'):
            if len(comm) < 2:
                print('More information needed to set power supply voltage.')
            else:
                if comm[1] == 'max':
                    print('\nRamping up to maximum voltage.')
                    conn,dat=send_message(UDP_IP,UDP_port,UDP_buffer,'v,x')
                elif comm[1] == 'min':
                    print('\nRamping down to zero voltage.')
                    conn,dat=send_message(UDP_IP,UDP_port,UDP_buffer,'v,n')
                elif comm[1].replace('.', '', 1).isdigit():
                    if vconfig == False:
                        try:
                            vmin = float(input('Please specify the minimum output voltage of the
power supply in kV (e.g., 0): '))
                        except:
                            vmin = 0.0
                        try:
                            vmax = float(input('Please specify the maximum output voltage of the
power supply in kV (e.g., 10): '))
                        except:
                            vmax = 10.0
                        vconfig = True
                    else:
                        if float(comm[1]) > vmax:
                            comm[1] = float(vmax)
                        elif float(comm[1]) < vmin:
                            comm[1] = float(vmin)
                print('\nDirectly setting power supply voltage to '+comm[1].strip()+ ' kV.')
                try:
                    dacValue = int((float(comm[1].strip())/(vmax-vmin))*bits)
                except ValueError:
                    print('Value error; please enter a valid voltage')

```

```

        continue

conn,dat=send_message(UDP_IP,UDP_port,UDP_buffer,'v,c,'+ '{0:04d}'.format(dacValue))
    else:
        print('Invalid voltage command.')

    elif connected == True and (comm[0] == 'i' or comm[0] == 'inhibit'):
        if comm[1] == 'on':
            thisComm = 'I'
        elif comm[1] == 'off':
            thisComm = 'i'
        else:
            thisComm = ''
        if thisComm != '':
            if thisComm == 'i':
                print('\nDisabling power supply inhibit.')
            elif thisComm == 'I':
                print('\nEnabling power supply inhibit.')
            conn,dat=send_message(UDP_IP,UDP_port,UDP_buffer,thisComm)
            if conn == True:
                parsed = dat.lower().split(',')
        else:
            print('Unrecognized inhibit command, try again.')

    elif connected == True and (comm[0] == 's' or comm[0] == 'status'):
        print('\nGetting status...')
        conn,dat=send_message(UDP_IP,UDP_port,UDP_buffer,'s')
        if conn == True:
            parsed = dat.lower().split(',')
            print(parsed)

    elif connected == True and (comm[0] == 'r' or comm[0] == 'reset'):
        print('\nResetting Arduino...')
        send_message(UDP_IP,UDP_port,UDP_buffer,'r')

    elif comm[0] == 'u' or comm[0] == 'update':
        print('Updating Arduino connection information...')
        UDP_IP, UDP_port, UDP_buffer = get_net_info()
        send_message(UDP_IP, UDP_port, UDP_buffer,'t')

    elif comm[0] == '?' or comm[0] == 'help':
        comm_help()

    elif comm[0] == 't' or comm[0] == 'test':
        send_message(UDP_IP,UDP_port,UDP_buffer,'t')

    elif comm[0] == 'x' or comm[0] == 'exit':
        print('\nGoodbye.\n')
        alive = False

    else:
        print(comm)
        print('\nCommand not recognized. Try again, perhaps?')

```