Sector Analysis of the p53 Tumor Suppressor Protein

by

Josh Phythian
Class of 2025

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors from the College of Integrative Sciences
and with Departmental Honors in Computer Science

Acknowledgements

During my first swim lesson, when I was around 5 or 6 years old, the instructor just threw me into the pool, in the hopes I could figure out how to swim out of necessity. I was unsuccessful and had to be saved. My parents did not approve of this teaching method. Last summer, I started my first research job as a CIS summer research fellow for the Thayer Lab. Joining the lab in my junior summer, I had a lot to catch up on. The last time I took a biology class was high school. I started my junior spring semester, thinking I would major in Computer Science and Math. I never could have imagined that I would declare as a Computer Science and College of Integrative Sciences double major, join a computational molecular biophysics lab, and write a thesis for both majors. However, with Professor Thayer's consistent support as my research advisor and mentor, and the persistent support of the other Thayer lab summer researchers, I was able to travel to UC Merced to attend and present at the MERCURY computational chemistry consortium one and a half months after I joined the lab. This felt similarly to being thrown in the pool again, but this time I could ask others how to stay afloat, and eventually learn how to swim on my own.

I would like to thank Professor Thayer, for her unwavering support ever since I have joined the lab, and throughout the process of this thesis. I would like to thank my parents for supporting me and our weekly calls. I would also like to thank the rest of my family for their support, while I have figured out what I would like to do for a living. I would like to thank my fellow Lobster Room members, and I am proud to call all of you my peers. I would like to thank my

roommates Peter and Vahn, for always being people I look forward to seeing upon returning home after a long day. I would also like to thank my friends for always being there for me, even during times when we have all been stressed.

Finally, thank you to my thesis readers Professor Kelly Thayer, Professor James Lipton and Professor Danny Krizanc for assisting in this project.

Contents

CHAPTER 1

Introduction

1.1 Overview of the Project

The ultimate goal of the Thayer lab is to return p53 to its native function after it mutates. The p53 protein is implicated in over 50% of human cancers (Lane). When it mutates, it ceases to be able to perform its native function: killing damaged cells, leading to uncontrolled cell growth—the hallmark of cancer. Therefore finding a way to return p53 from its mutant state to its native state is of the utmost importance.

Engineering a molecule to rescue proteins requires atomic level information which is not available from wet lab studies. Computer science plays a critical role in this problem that Molecular Biology cannot: utilizing new advancements in high performance computing to produce high throughput models that produce cheaper and faster results, with atomic level resolution to explain the underlying mechanics. Allostery can broadly be defined as: one molecule binds to another macromolecule which modulates the binding behavior at a site distinct from the location of the binding event. An allosteric network is a system of residues that can communicate a signal across a protein from one site to another distinct site. Our lab hypothesizes that a network of motionally covarying residues, known as the sector, could send an allosteric signal to reactivate mutant p53.

1.2 Context of the Project

The goal of this project is to create a pipeline for finding the points of allosteric control, and applying to various systems (Figure 1.2.1). These points are distinct from the active site and when bound to, modulate the behavior of the protein through allosteric networks (Wodak et al.; Liu and Nussinov). The allosteric points in p53 have still not been discovered. This gap in knowledge suggested investigating methodologies for capturing long range signals in proteins for allosteric drug design. Our lab's approach to tackle this problem is through molecular dynamics (MD) simulations. The sector hypothesis posits that allosteric networks could be captured as a cohesive network of amino acid residues capable of transmitting an allosteric signal through a protein.

MD sectors is a method that takes a Molecular Dynamics simulation as input and outputs a subset of residues from the molecule the simulation focuses on: the sector or allosteric control points. Identifying these points is a bottleneck throughout the drug design process. This thesis builds upon previous iterations of MD sectors with high performance computing techniques, a novel refinement algorithm, and enhanced code accessibility, in order to open up that bottleneck, making a pipeline that is optimized, comprehensible, and efficient.

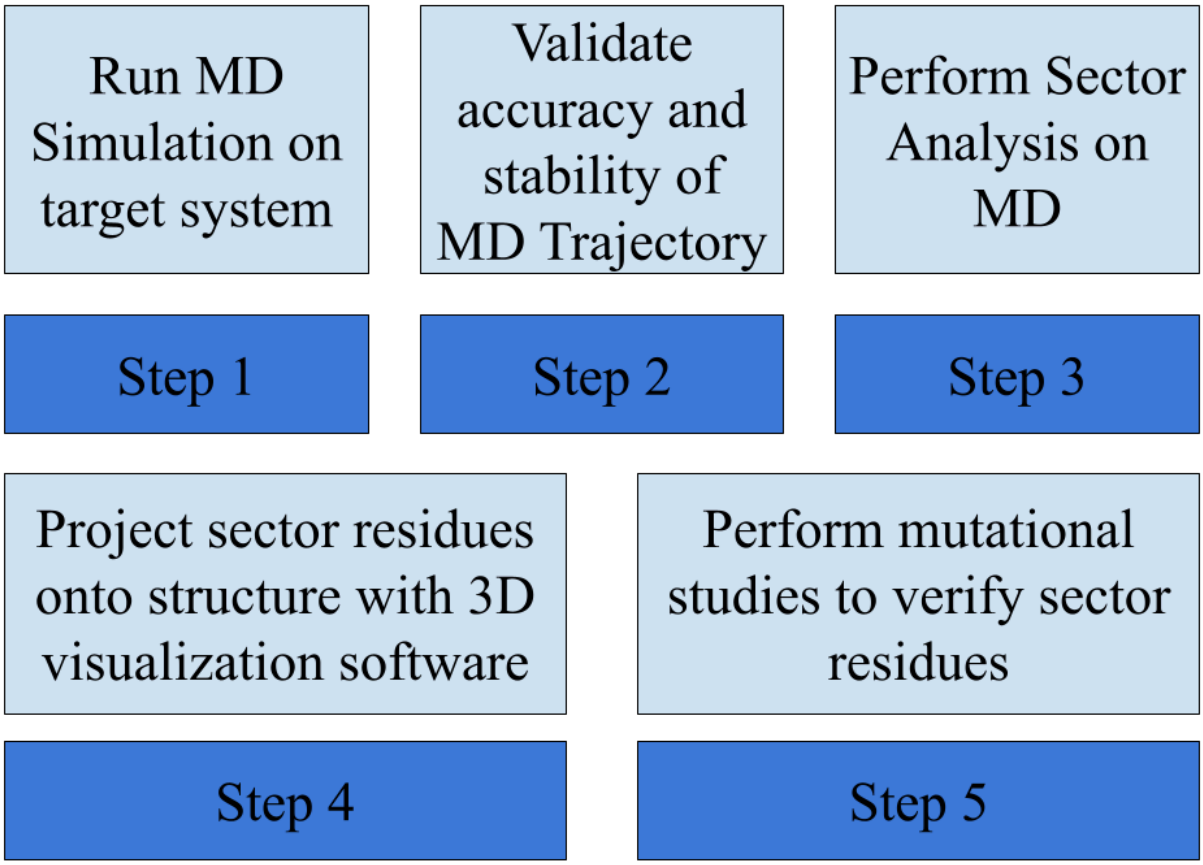


Figure 1.2.1 Thayer Lab's research process for identifying potential drug candidates.

CHAPTER 2

Background

2.1 High Performance Computing

Traditional drugs either fully inhibit or activate a protein. A new paradigm for drug design, allosteric therapeutics, involves manipulating the active site as opposed to blocking it, in order to modulate a protein's behavior. To carry this out, we need to implement action from a distance, in other words allostery. However the underlying molecular mechanics are poorly understood. Our lab is attempting to model allostery with networks. One way of identifying these networks is sector analysis, the topic of this thesis. One part of this thesis is to establish a pipeline that will take an MD trajectory and output a network of amino acid residues that have a correlated motion.

Although this sounds like a pure biological problem, it is not. If one were to try to identify these allosteric control points on a protein it would take a copious amount of time. If the system of interest has *N* residues, the sector would be a specific subset of residues with size *N*/5, rounding up. Let's say the system of interest is the Y220C DNA binding domain, which has 195 residues. If one were to try all of the possible subsets they would have to try $\binom{195}{39}$ different combinations to find the allosteric control points. This is not plausible as that is roughly 2,340,000,000,000,000,000,000,000,000,000,000,000,000,000 different possibilities. This does not even include the full length p53, which is in fact 393 amino acids long. This number will get smaller by utilizing biological knowledge,

but finding a molecule that binds to the target compound remains a substantial bottleneck in the drug discovery process. Finding the target points for a new drug to bind is also expensive. In 2010, identifying these points for a prospective drug was estimated to cost 94 million dollars and several years, with that estimation being noted as a great underestimate for unprecedented therapeutics (Mohs and Greig).

In addition to identifying the allosteric points of control, another obstacle with a purely biological approach is that there are no known methods for understanding allosteric signalling. This event occurs within a protein, so in order to capture information about long-range communications within a protein is not plausible with current technology.

This obstacle of cost and time suggests a different method for drug discovery: high performance computing and molecular dynamics simulations. In order to gain key insights upon the mechanisms underlying allosteric signalling, the Thayer lab uses molecular dynamics simulations to model systems of interest, with atomic level clarity. One example of use of MD is testing how molecules bind to prospective therapeutic compounds. This has yielded positive results thus far, as exemplified by the PK11000 compound shown to rescue Y220C, a frequent cancerous p53 mutant (Han and Thayer; Han et al.) Not only is this cheaper, it is much faster than trying to model these systems *in vitro*. The atomic level of detail provided by MD enables analysis also with an atomic level of clarity. However, this granularity comes at a cost: there is a copious amount of data. This suggests use of high performance computing, in order to process this data and in this case

gain insights of the allosteric control points for the system of p53, including its various isoforms.

At first, I was given an implementation of MD sectors, written in pure Python, that took around 23 minutes to fully execute when performed on the correlation matrix calculated from the MD of Y220C on a 2020 Macbook Air. This will be referred to as the first implementation. It would output errors, non-symmetric matrices, and artifacts of testing due to some environment or other technical issues. A correlation matrix is inherently symmetric, so this was wrong. It was difficult to understand the code, as there were untrackable single-use global variables, and function puzzleknots.

Although Python's concise readability almost mirrors natural language and enables type indeterminacy, there is a tradeoff: it is notably slower than other languages. Python is an interpreted language, meaning after code is executed it is run line by line at runtime. It also is far more disconnected from the hardware, with automatic dynamic memory allocation. This is exemplified by Python's heterogeneous lists, able to contain any amount of types. Let's say there is a list with an integer, a floating point number, a dictionary and even another list (Figure 2.1.1). Not only do each of these objects all have different amounts of memory associated with them, the data is scattered all over a computer's memory. The result of this flexibility is an iteration nightmare.

Figure 2.1.1 Simplified example of a Python heterogenous list. In this list there is an object of type penguin, integer, ladybug, and giraffe. The size of the rectangles for each object represent the differing amount of memory allocated for each of them.

In contrast, C is compiled, being turned into machine code or 1's and 0's, and optimized by the compiler before it runs. C does not have a list but a similar structure called an array. An array is homogeneous, meaning there can only be one type within a single array. Each object gets an equal amount of memory in sequence (Figure 2.1.2). Although C is more strict on how variables and types are defined, it enables more customization for how an object lies in memory. This is great for iteration because when iterating over an array, each object is right next to each other in sequence.

Figure 2.1.2 Simplified example of a C homogeneous array. In this array there are 8 objects of type integer. The size of the rectangles for each object are equal, representing the amount of memory allocated for each of them.

Pandas and NumPy, the two main data analysis modules used for this thesis, were written in C or C++. NumPy has a data structure very similar to C's array: the NumPy array. It supports vectorized operations, meaning if one were to apply a function to a whole NumPy array of length *N*, it would take only one operation. However, in Python, applying a function to a whole list of length *N* would take *N* operations. As *N* grows bigger, the total time it would take to iterate over the list would grow much larger.

By using custom NumPy data structures, built upon NumPy arrays, my implementation takes an average 0.02 seconds on the same Y220C matrix and Macbook. For clarity, there are no global variables, no confusing functions, and more abstraction to turn single use functions into multipurpose functions. For accessibility I made a Flask website, a Github repository, and a directory on the High Performance Computing Cluster (HPCC). Originally the website was hosted

using Wescreates, but in mid February 2025, the new update no longer supported Python apps. As a result, I have started hosting the website from Github.

In addition to utilizing NumPy, another way to optimize Python code is to add types. Normally, the interpreter will assign types automatically, but by doing this it makes the interpreter's job easier, thus enabling the code to execute faster. Another option for optimization is algorithmic optimizations. By making clever changes to an algorithm, one can get the same results, but faster. Finally, one can also make hardware optimizations. By processing data with the Graphical Processing Unit (GPU), it enables vectorization, creating an immense speedup by utilizing parallelization properties. An example of this is AMBER's switch from using central processing units (CPUs) to GPUs to parallelize running MD simulations, which led to roughly 10x speedup. (Walker).

My goals for the implementation of MD sectors for this thesis are twofold: first, use high performance computing and algorithm development techniques to improve accessibility efficiency, and conciseness, whilst not compromising clarity; second, provide molecular level insights pertaining to allosteric pathways within the p53 system and demonstrated applicability to other systems.

2.2 p53: The Tumor Suppressor Protein

"Guardian of the Genome" or p53 is a protein whose main function is to suppress tumors and is implicated in over 50% of human cancers (Lane)(Figure 2.2.1). When a cell has been afflicted with DNA damage, p53's job is to judge if the cell should be repaired or undergo apoptosis (meaning killing off the damaged cell, otherwise known as programmed cell death), in order to get rid of defective

or superfluous cells. However, when p53 mutates, it enables damaged cells to reproduce unhindered, which can lead to cancerous tumors. Even though it has a vital function, there are no known drugs that can restore p53 to its original function. Globally, cancer is one of the top leading causes of death, so curing cancer is of the utmost importance (Vogelstein et al.).

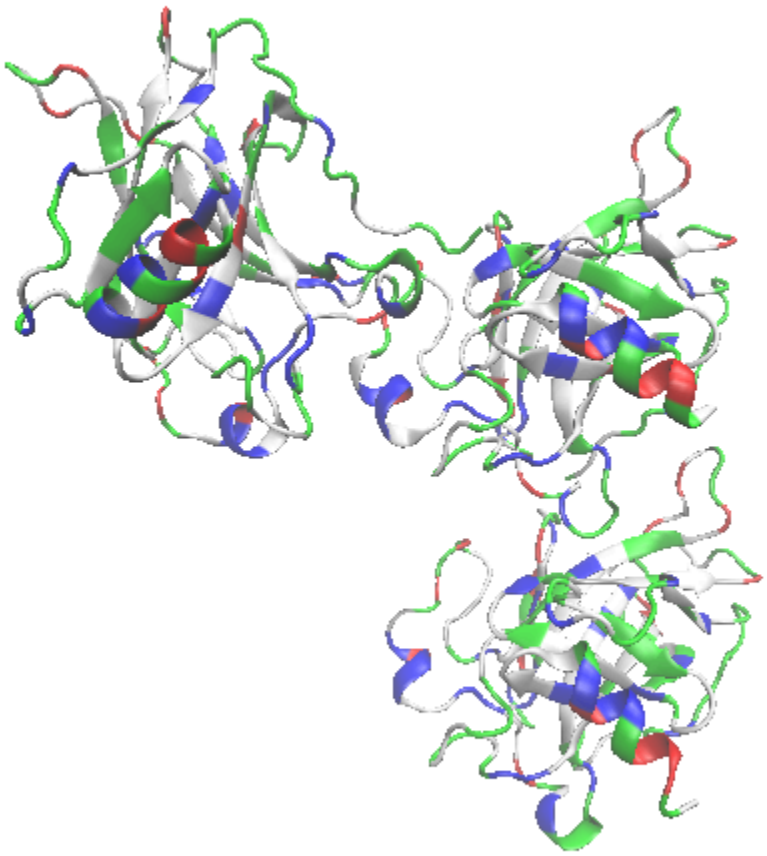


Figure 2.2.1 3-D visualization of p53. The gray residues are hydrophobic. The green residues are polar uncharged. The red residues are acidic. The blue residues are basic.

The core of p53 was first crystallized in 1994 by Cho et al., providing atomic-level insights into its structure and interaction with an engineered consensus DNA sequence (Cho et al.). Proteins are large molecules made up of a specific sequence of amino acid residues. The Protein Data Bank ID for this crystal structure is 1TUP (Berman et al.). The full-length p53 protein consists of 393 residues, which is made up of 3 regions: the DNA-binding domain (DBD), N-terminal and C-terminal. The DBD includes residues 102 to 292 and adopts a beta-barrel structure, along with several crucial elements involved in DNA binding: the L1 loop, L2 loop, L3 loop, and the strand-loop-helix (Borrero and El-Deiry)(Figure 2.2.2). The N and C terminal regions are intrinsically disordered and are responsible for regulating p53's activity through post-translational modifications.

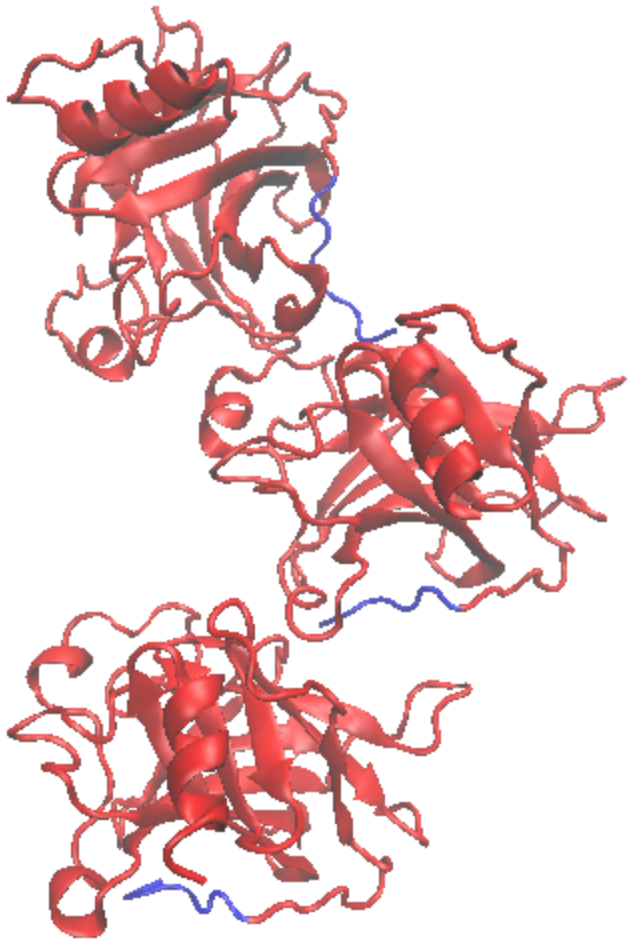


Figure 2.2.2 3-D visualization of p53 where the full length is blue and the DBD is red.

The 1TUP structure identifies eight residues that directly form hydrogen bonds with DNA: K120, R241, K248, K273, A276, A277, R280, and R283 (Cho et al.)(Figure 2.2.3). K120 refers to the 120th amino acid in p53, which is Lysine,

often abbreviated to K (Figure 2.2.4). There are ten mutations that are frequently in cancerous samples: R175H, R248Q, R273H, R248W, R175L, Y220C, R273C, R282W, R248L, R175P (Figure 2.2.5). The first capital letter is the name of the initial amino acid, the number refers to the position, and the last capital letter refers to the amino acid that the initial amino acid changes to. The positions where these mutations occur are known as hotspots. Notably all of these hotspots are within the DBD.

Figure 2.2.3 3-D visualization of p53 where the full length protein is blue and the hydrogen bonding residues are red beads.

Amino Acid	Three Letter Code	Single Letter Code
Alanine	Ala	A
Arginine	Arg	R
Aspargaine	Asn	N
Aspartic Acid	Asp	D
Cysteine	Cys	C
Glutamic Acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Three Letter Code	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Figure 2.2.4 Amino acid codes.

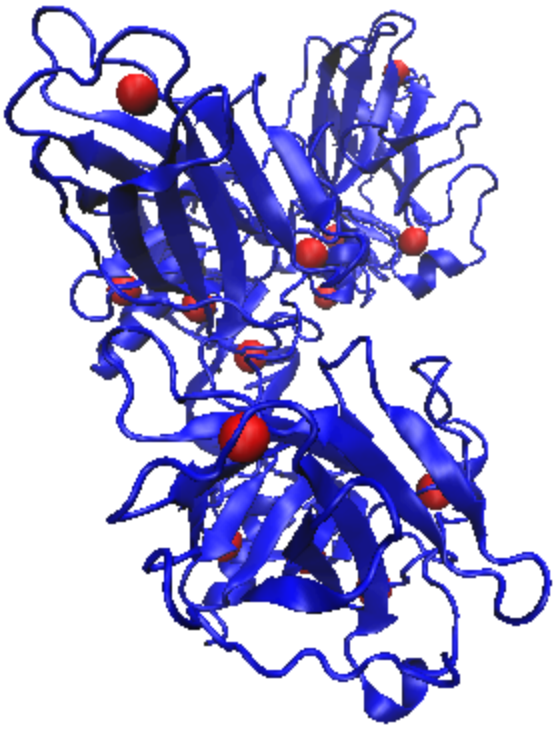


Figure 2.2.5 3-D visualization of p53 where the full length protein is blue and the hotspot residues are red beads.

Crystallography has shown promise for restoring p53 to its native function, but advancements have not succeeded yet. Crystallography is a branch of science that tries to find the structures and bond arrangements of molecules. However, due to Crystallography's static nature, it is difficult to capture the dynamic nature of p53 (Liu and Nussinov). H168R is capable of restoring p53 to normal binding for mutation R249S and T123A, which exemplifies that mutations

can be restored to their original function by perturbation within p53. However, other studies have focused on the Y220C mutation. Even though it is not one of the more frequent hotspots, it has been an area of focus because it plays a pertinent part in the restoration of p53. PK083, PK7088, PK11000 are three drugs that focus on reintroducing the aromatic ring that is lost in the Y220C mutation. PK083 and PK7088 did not show much promise, but PK11000 was able to covalently bind at 182, which is distinct from 220 and the DNA [thayer review]. In addition, the rescue of Y220C by PK11000 restored Hydrogen bonds at the active site [han article]. It was shown that this drug eradicates tumors in rodent studies, but due to it covalently binding to surrounding cysteines is not a viable drug candidate.

The p53 protein has 12 total isoforms. An isoform is a similar version of a protein, but with a slightly different amino acid sequence, which leads to different structures and behaviors (Gunning and Hardeman). Through mRNA splicing, a single gene can produce different protein variants before the mRNA is translated.

Different isoforms exhibit different behaviors. For example, isoform 7 appears only in the bone marrow, colon, fetal brain, intestine, and testis, while isoform 9 is absent from the brain, breast, fetal liver, heart, intestine, lungs, and salivatory gland (Thayer et al.). Each isoform is determined by the permutation of including or excluding one to four of the isoform variable regions (IVRs)(Table 2.2.1). An isoform name in Δ notation will follow this format: Δ[0, 40, 133]p53[α, β, γ]. The number refers to which residue the isoform begins on. If the prefix is 0, includes IVR 1, if it is 40 then it includes IVR 2, but not IVR 1, if it is

133, then it does not include IVR 1 or IVR 2. An α suffix means that the isoform has IVR 3 and IVR 4. A β suffix has neither, a γ suffix means it has IVR 3, but not IVR 4. According to the crystal structure the DBD has residues 102 to 292. However, since not all isoforms include residues 1 to 132, this study will use the DBD' which begins at residue 133 instead of 94 (Armour-Garb et al.). This definition preserves the key functional elements of the DBD while encompassing its secondary structural features. Notably, this will exclude one of the residues that may form hydrogen bonds with the DNA, K120. Section 5.2 will investigate p53 isoforms 2 to 9. These specific isoforms were chosen because they all include the DBD' region, while isoforms 10 to 12 do not, since they start at residue 160.

Isoform	Δ Notation	1-39 IVR 1	40-132 IVR 2	133-159 DBD'	160-331 DBD'	IVR 3	IVR 4	Residues	Total Length
Isoform 1 (WT)	Δ0p53α	Y	Y	Y	Y	Y	Y	1-393	393
Isoform 2	Δ0p53β	Y	Y	Y	Y			1-341	341
Isoform 3	Δ0p53γ	Y	Y	Y	Y	Y		1-346	346
Isoform 4	Δ40p53α		Y	Y	Y	Y	Y	40-393	354
Isoform 5	Δ40p53β		Y	Y	Y			40-341	302
Isoform 6	Δ40p53γ		Y	Y	Y	Y		40-346	307
Isoform 7	Δ133p53α			Y	Y	Y	Y	133-393	261
Isoform 8	Δ133p53β			Y	Y			133-341	209
Isoform 9	Δ133p53γ			Y	Y	Y		133-346	214
Isoform 10	Δ160p53α				Y	Y	Y	160-393	234
Isoform 11	Δ160p53β				Y			160-341	182
Isoform 12	Δ160p53γ				Y	Y		160-346	187

Table 2.2.1 This table displays each isoform's name in Δ notation, its corresponding length, and the corresponding presence or absence of each isoform variable region.

2.3 MD Simulations

Molecular Dynamics Simulations (MD) are a close approximation of the behavior of proteins and other biomolecules in a given system. An MD takes the position of the atoms and the chemical bonds between those atoms in a system as input (Hollingsworth and Dror). This initial starting structure can be found from

many places, one of the more common ones being the Protein Data Bank (PDB). This affords high flexibility in engineering and designing a comprehensive set of experiments involving interactions with drug molecules and other ligands. The MD will output a trajectory, or a large text file sequencing the position or cartesian coordinates of each atom for each time step of the MD. This trajectory is essentially a movie showing a protein's behavior over the specified amount of time.

An MD is based upon Newtonian Mechanics, where the molecular behavior is calculated by repeatedly applying Newton's laws of motion to each time step. The position of an individual particle can be described as a function of time (Equation 2.3.3) by solving the differential equations of Newton's second law of motion (Equations 2.3.1 - 2.3.2).

Equation 2.3.1: $F = ma$

Equation 2.3.2: $\frac{d^2x_i}{dt^2} = \frac{F_{x_i}}{m_i}$

where i = atom index

m_i = mass of particle i

x_i = an individual coordinate of particle i

F_{x_i} = force on particle i in direction of x_i

Equation 2.3.3: $x_i(t) = x_i(t-1) + v_i(t-1)\,\Delta t$

where $x_i(t)$ = position of particle i at time t

v_i = velocity of particle i

In order to calculate how atoms and molecules interact with each other, an MD uses force fields, a mathematical model consisting of equations describing bond stretching, angle bending, torsional (dihedral) interactions and non-bonded interactions. An example of the functional form of a force field is shown in Figure 2.3.1. Bond stretching is the energy connected to the stretching or compressing of

bonds between atoms. This is calculated using Hooke's Law for harmonic motion. Angle bending is the energy associated with changes in bond angles. This is calculated using a harmonic potential. Torsional interactions are the energy related to rotation around a bond axis (torsion angle). Non bonded interactions are long range interactions that occur between atoms that are not directly bonded to each other. This usually follows Van der Waals interactions which are described using the Lennard–Jones potential, and electrostatic interactions are captured through a Coulombic term. A more rigorous force field could have additional terms, but will definitely include the previously mentioned 4 terms.

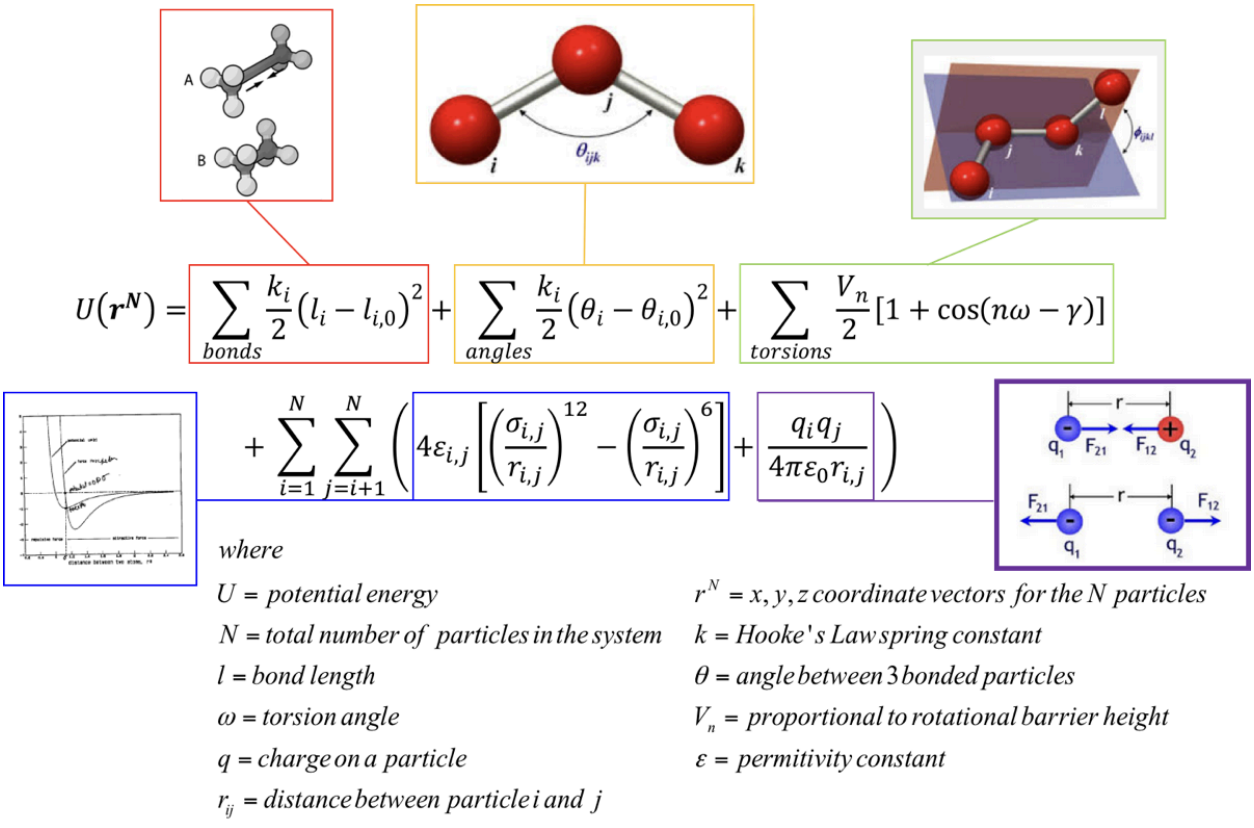


Figure 2.3.1 The functional form of a force field with four parameters: bonded

interactions, valence angles, torsions and non-bonded interactions. Adapted from: Fabry, Jonathon. (2020). *Network Analysis of Molecular Dynamics Sectors in the p53 Protein* (Undergraduate thesis). Department of Computer Science, Wesleyan University, p. 14.

Computing the positions and energies of the particles is computationally expensive, since each time step will have thousands of interactions in a biological system. When paired with wet lab studies, results can be used to validate data from simulations. Wet lab studies include physical manipulations of biological or chemical materials to produce empirical data.

MD enables one to analyze a biomolecular system with more precision than any experimental technique because of the ability to control the simulation conditions and precision of molecules involved in the simulation. This level of precision is useful for ascertaining an atomic level of clarity when modeling interactions for drugs we are trying to develop.

2.4 Allostery

Allostery, also known as “the second secret of life,” is a process in which perturbations on one site of a protein affect another distinct site on the same protein, through the propagation of signals (Hertig et al.). In 1904, Christian Bohr — before the word allostery was even invented — observed that a carbon dioxide molecule affected the binding affinity of oxygen to a protein (Liu and Nussinov). In other words, a conformational change occurred in the protein because a molecule bound to it, which modulated another molecule's binding affinity to a protein (Figure 2.4.1). Allostery plays a key part in many biological functions,

such as regulation of metabolic enzymes or metabolism. However, there are no current explanations for the principles underlying allostery, making it a pertinent area of study.

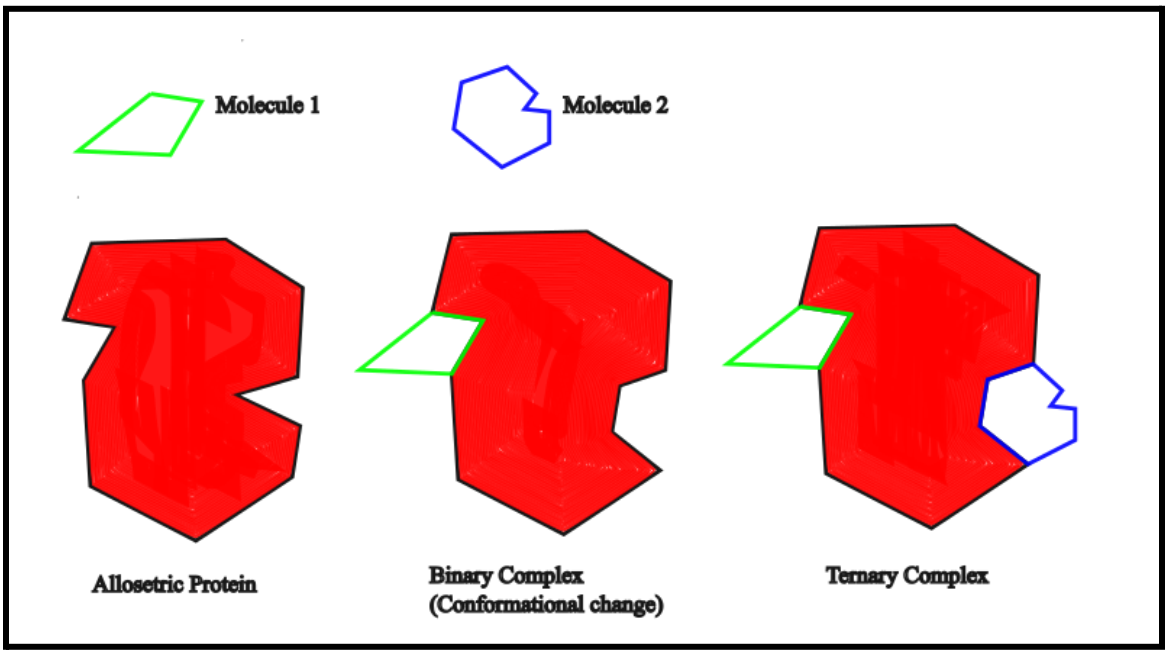


Figure 2.4.1 Simple illustration of allostery. When molecule 1 binds to the allosteric protein, it creates a conformational change to enable molecule 2 to bind to the protein.

2.5 Allosteric Drugs

The way allosteric drugs work is a great example of allosteric interactions within protein. Allosteric drugs can control an active site distally, enabling it to continue to carry out its function (DeDecker). This is an important trait if the activity of a protein only needs to be modulated and not stopped completely (Wodak et al.). Many currently undruggable diseases would benefit from this, for example cancer, where the restoration of the mutant p53 to wild-type would enable p53 to go back to its function of suppressing tumors.

That being said, the process of creating an allosteric drug has many difficulties: finding the allosteric points, being able to predict if the molecules will be able to bind the points energetically, and understanding allosteric signalling.

The first major challenge is locating these allosteric control points—specific residues within the protein that, when bound, propagate signals to the active site. Currently, there are no efficient experimental methods to pinpoint these residues, as doing so requires selecting a precise subset within the protein and ensuring each choice is correct. Not only is this process costly, but it also demands a significant amount of time.

The next challenge is predicting whether molecules will successfully bind to these allosteric sites—a problem known as energy docking. This involves determining whether the binding interactions are energetically favorable, which is crucial for drug design. While some experimental and computational data exist, they remain insufficient for training robust machine learning models. The lack of comprehensive datasets limits the accuracy of predictions, making it difficult to reliably identify viable allosteric drug candidates.

The final challenge is understanding long range allosteric signaling. The theoretical framework for allosteric signaling remains incomplete, making it difficult to predict how binding at one site influences distant regions of a protein. Advancing this field requires deeper computational analysis, as traditional experimental approaches alone have yet to provide a clear mechanistic understanding.

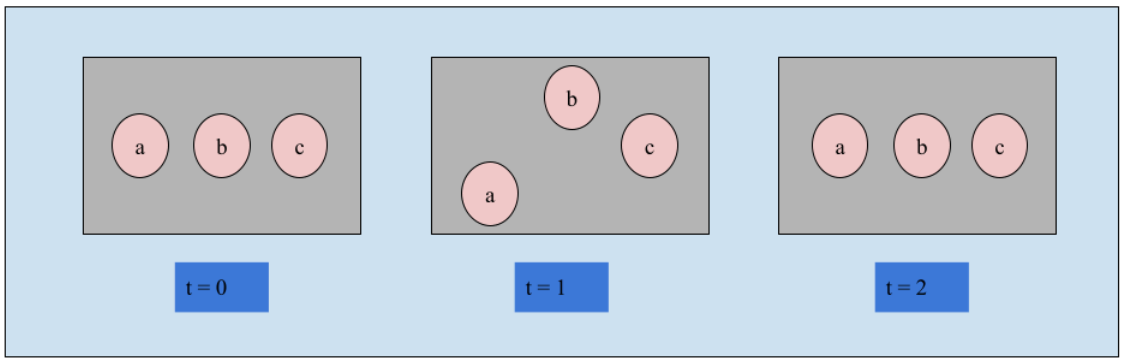
Even though these sound like purely biological problems, this thesis will aim to address finding the allosteric control points and understanding long range allosteric signaling, by leveraging high performance computational methods to improve our ability to model and predict allosteric interactions.

One method that has been used for finding the allosteric points is computer-aided drug design (CADD) with high-throughput methods, which can get around the economic and time costs of the experimental method, with a tradeoff of being computationally expensive. So far, CADD has had success in producing drugs that target active sites, but there are no known approaches for identifying allosteric points. This is where high performance computing plays a key part, as taking advantage of numerous modules to basically turn Python into C, and distributing code to the GPU can greatly cut down the computational costs.

This prompted the creation of MD sectors, a method based upon the sector hypothesis: allosteric networks could be captured as a cohesive network of amino acid residues capable of transmitting an allosteric signal through a protein. Our lab hypothesizes that if we were to get a motionally correlated subset of residues within the p53 protein, we could transmit a signal across an allosteric network, composed of the sector residues, in order to restore p53 to its native function.

Therefore, if our lab were able to identify the allosteric control points, which we hypothesize are the residues in the sector, we would be able to modulate the binding specificity of p53, therefore enabling p53 to go back to suppressing tumors. Jonathon Fabry, a previous Thayer lab member, demonstrated that sectors

are a cohesive network, by using Vpica values and Vpica Ratio Analysis (VRA). Each Vpica value represents a residue's overall correlation with the rest of the protein. VRA measures the covariance between residues inside and outside the sector, in order to create a more cohesive sector. In this case, cohesive means a subset of residues who move in a correlated motion to a high degree (Figure 2.5.1). The goal of MD sectors is to label residues such that they are in the sector or not in the sector, by weighing the correlation strength of the residues. The residues in the sector will have a higher total correlation value than the residues outside of the sector.



At $t = 0$ residues a, b, c initial positions are (0, 0), (1, 0), (2, 0) respectively.

At t = 1 they move to (0, -1), (1, 1), (2, 0) respectively.

At t = 2 they move back to (0, 0), (1, 0), (2, 0) respectively.

Figure 2.5.1 Illustration of simple distance correlation for 3 residues in R2. Residues a and b covary with each other, while residue c does not covary with the other residues.

The purpose of this thesis is to utilize high performance computing to enrich MD sectors into a high throughput model, leading to a systematic, efficient way to identify allosteric control points, and further refine them, which is

currently the missing piece in CADD. The long term goal of our lab is to develop allosteric therapeutics, which we hope will lead to approval through clinical trials, enabling human use, that would be able to cure currently undruggable diseases such as cancer and HIV, in safe dosages.

2.6 MD Sectors

Molecular Dynamics (MD) sectors, a method developed by the Thayer lab, is grounded in biophysical principles and statistical coupling analysis (SCA) sectors (Süel et al.; Rivoire et al.). The difference between SCA and MD sectors is that SCA sectors analyze evolutionary covariance, while MD sectors capture residue's motional covariance within an MD trajectory. Elastic and Gaussian Network models are among other methods used to identify potential allosteric residue networks. Although these methods provide residue networks, they do not provide verification of checking these networks. Other method's sectors measure cohesiveness in relation to the whole system, while my iteration will maximize each sector residue's cohesiveness in relation to other sector residues with the new refinement algorithm. This falls more in line with the hypothesis that allosteric networks are a cohesive network of amino acid residues capable of transmitting an allosteric signal through a protein.

The method for performing MD sectors is what follows: acquire a trajectory from an MD simulation, calculate a pairwise correlation matrix, and apply the sector algorithm.

First one must run an MD simulation, to acquire a trajectory to perform analysis on. Within the Thayer lab this will be created using the standard Thayer lab procedure as explained in section 3.2.

Second, calculate a pairwise motion correlation matrix from the previous matrix (Figure 2.6.2). This matrix will have a diagonal, functioning as a line of symmetry, since the row and column labels are identical. Also, the values in this matrix will range from -1.0 to 1.0, -1.0 being not correlated at all, and 1.0 being correlated. If two residues are the same, their correlation value will be 1.0 because they are the same residue, so they will move in the same motion.

Res #	236	238	239	237	195	235	274	194	137	179	196	175	178	242	176	138	248
236	0	0.909	0.838	0.956	0.87	0.917	0.669	0.791	0.896	0.847	0.866	0.737	0.82	0.658	0.724	0.907	0.565
238	0.909	0	0.926	0.958	0.862	0.764	0.663	0.883	0.792	0.925	0.79	0.876	0.925	0.818	0.885	0.773	0.705
239	0.838	0.926	0	0.843	0.697	0.683	0.829	0.726	0.805	0.796	0.623	0.735	0.813	0.857	0.777	0.721	0.797
237	0.956	0.958	0.843	0	0.903	0.862	0.585	0.867	0.837	0.918	0.879	0.831	0.895	0.701	0.815	0.86	0.569
195	0.87	0.862	0.697	0.903	0	0.792	0.419	0.904	0.688	0.851	0.919	0.828	0.825	0.599	0.794	0.748	0.472
235	0.917	0.764	0.683	0.862	0.792	0	0.556	0.621	0.881	0.718	0.89	0.558	0.675	0.451	0.523	0.929	0.359
274	0.669	0.663	0.829	0.585	0.419	0.556	0	0.39	0.764	0.483	0.375	0.396	0.507	0.654	0.453	0.612	0.716
194	0.791	0.883	0.726	0.867	0.904	0.621	0.39	0	0.562	0.902	0.763	0.939	0.885	0.73	0.921	0.594	0.57
137	0.896	0.792	0.805	0.837	0.688	0.881	0.764	0.562	0	0.701	0.742	0.528	0.682	0.569	0.53	0.942	0.533
179	0.847	0.925	0.796	0.918	0.851	0.718	0.483	0.902	0.701	0	0.789	0.912	0.979	0.753	0.911	0.718	0.591
196	0.866	0.79	0.623	0.879	0.919	0.89	0.375	0.763	0.742	0.789	0	0.698	0.75	0.468	0.649	0.828	0.335
175	0.737	0.876	0.735	0.831	0.828	0.558	0.396	0.939	0.528	0.912	0.698	0	0.913	0.802	0.98	0.54	0.604
178	0.82	0.925	0.813	0.895	0.825	0.675	0.507	0.885	0.682	0.979	0.75	0.913	0	0.791	0.928	0.686	0.631
242	0.658	0.818	0.857	0.701	0.599	0.451	0.654	0.73	0.569	0.753	0.468	0.802	0.791	0	0.854	0.479	0.79
176	0.724	0.885	0.777	0.815	0.794	0.523	0.453	0.921	0.53	0.911	0.649	0.98	0.928	0.854	0	0.521	0.668
138	0.907	0.773	0.721	0.86	0.748	0.929	0.612	0.594	0.942	0.718	0.828	0.54	0.686	0.479	0.521	0	0.414
248	0.565	0.705	0.797	0.569	0.472	0.359	0.716	0.57	0.533	0.591	0.335	0.604	0.631	0.79	0.668	0.414	0

Figure 2.6.2 Snippet of Motion covariance matrix for Y220C. The purple cells are residue numbers, low values are mapped to blue and high values are mapped to green.

Third, apply the sector algorithm to the pairwise motion covariance matrix, which will produce a subset of residues with a total length of 20%. Although the 20% was initially based upon *ad hoc* observations, Fabry did a comprehensive study on the various cutoffs and illustrated that 20% was a reasonable number (Rivoire et al.; Fabry and Thayer). These sector residues hypothetically identify the residues within the allosteric network.

One of the first implementations of MD sectors by Bharat Lakani output the following sector residues (Figure 2.6.3):

L130, N131, V143, R158, A159, M160, C176, D184, S185, D186, G187, L188, A189, P190, P191, Q192, H193, L194, D207, D208, R209, N210, S215, V216, V217, V218, M237, I254, I255, E258, G262, N263, L264, N268.

Jonathon Fabry did another implementation of sectors, and he obtained these residues (Figure 2.6.4):

P128, A129, L130, N131, V143, R158, A159, M160, P177, S183, D184, S185, D186, G187, L188, A189, P190, P191, Q192, H193, R196, D207, D208, R209, N210, S215, V216, V217, V218, M237, I254, E258, G262, N263, L264, N268, E286, E287.

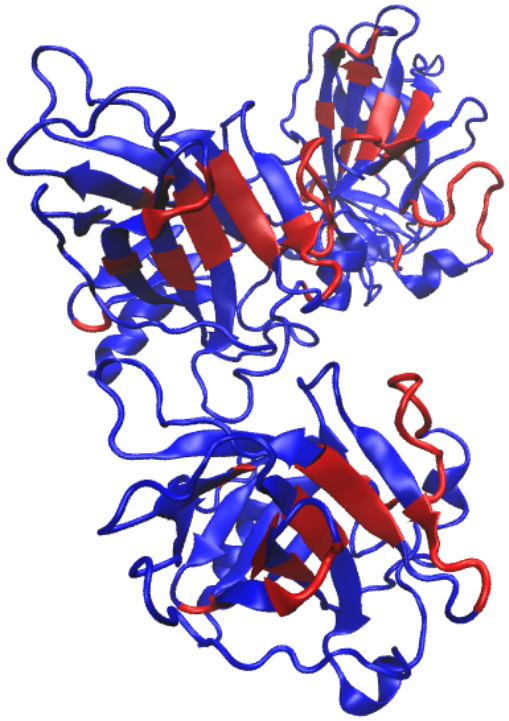


Figure 2.6.3 Bharat Lakani's sector residues (red) projected onto full length p53 (blue).

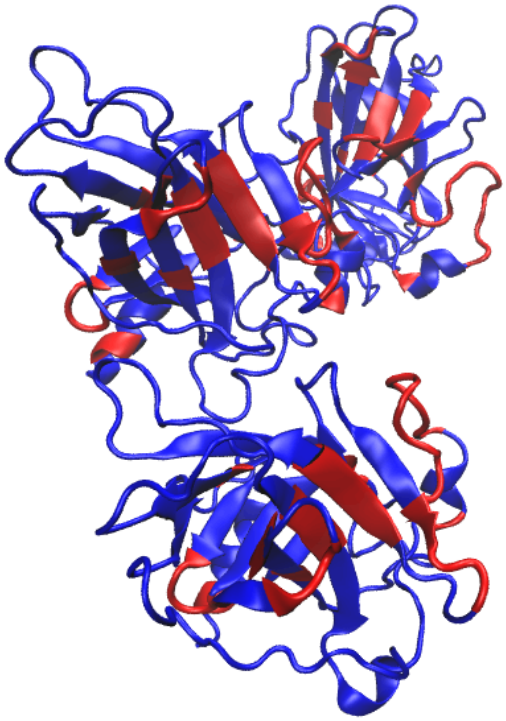


Figure 2.6.4 Jonathon Fabry's sector residues (red) projected onto full length p53 (blue).

CHAPTER 3

Methods

3.1 Python Modules

3.1.1 NumPy

NumPy or Numerical Python is an open source Python library that is mostly written in C and C++ for optimization purposes, enabling fast computation with Python's flexibility. It contains a data structure, similar to a native Python list, a NumPy array. A NumPy array is in sequence in the computer's memory, similar to a C array, while a Python list is not in sequence in memory. While a Python list can be non-homogeneous, and has dynamic memory allocation, NumPy arrays are homogeneous and statically allocated. This enables faster iteration of arrays than lists, which becomes increasingly useful as the datasets grow bigger. In addition NumPy supports vectorized operations. In Python, adding each element of two lists of length N together would take N operations. However, in NumPy, it would only take one operation to add two lists of length N.

3.1.2 Pandas

Pandas is an open source Python library including data structures and functions for data analysis. This implementation of MD sectors was initially written using Pandas to handle all of the matrix manipulations, but switched to NumPy because it is faster. A benefit of using Pandas was the ease of indexing, since each column in a dataframe could be indexed by the column name, whereas

NumPy required index by position in the array. After implementing NumPy, Pandas was only used to load .csv files and plot heatmaps with Seaborn, due to their high compatibility.

3.1.3 Matplotlib

Matplotlib is a Python library enabling visualization of data, supporting static or dynamic 2D and 3D plots (Zotero). It is written mostly in Python with other parts being written with NumPy—for performance, C, and Javascript. Bar plot graphs in this thesis are produced using Matplotlib

3.1.4 Seaborn

Seaborn is a Python library built upon Matplotlib, but with more customizability. This is used throughout this thesis to produce heatmaps, a way to display a matrix graphically. Heatmaps are produced by loading the correlation matrix data as a data frame then using the function *seaborn.heatmap()* with some other parameters.

3.1.5 Typing

Typing is a native Python module, which is utilized so one function can return two NumPy arrays. Without importing this module, returning two NumPy arrays is not possible. This trait helped in reducing execution time because without Typing matrix sorting functions would have to be run twice, one time to calculate the sorted matrix, another for calculating the labels of the matrix. With Pandas this was not an issue, since the data frame is labeled, but NumPy arrays are not labeled.

3.1.6 Time

The Time module is used in order to assess the runtime of my functions. This module was used for the *time_func* function, a wrapper function that would print the amount of seconds a function takes to fully execute. To use this, add the decorator *@time_func* on the line before the definition of the function of interest. This was useful for integrating different versions of the Sector code and seeing where the bottlenecks were.

3.1.7 Os

The Os module enabled quick importing and exporting different datasets. It is used to export spreadsheets containing function times—generated by *time_func*, visualizations, and matrices. *os.path.join()* is frequently used to set up import and export directories.

3.1.8 Imagemagick

Imagemagick is an open source library used throughout this thesis to convert Tachyon files to PNG files. This function saved a copious amount of time, as individually exporting each Tachyon file to be PNG format would take significantly longer than typing a Bash command.

3.1.9 VMD

VMD is a molecular visualization tool that is used for visualizing and analyzing structures such as proteins, nucleic acids, and other small molecules. VMD used openGL to create 3D visualizations of molecules according to the rules of chemistry and molecular geometry, displaying where each atom in a

molecule is a coordinate, respecting their van der Waal radii. VMD enables one to toggle the different drawing methods for visualization. Each visualization used throughout this thesis roughly follows this procedure.

First, load the PDB file corresponding to the molecule of interest in VMD. Then, to ensure consistent visualization across different constructs, apply a predefined orientation by entering the following commands in the VMD Tk:

```
mol modstyle 0 top NewCartoon
mol modcolor 0 top ColorID 0
color Display Background white
axes location Off
```

These commands adjust the drawing method to be New Cartoon, turn the color of the protein to blue, change the background color to white, and get rid of the axes.

To visualize Y220C, WT, or PK11000, which differ in scale from the full-length p53 protein, adjusting the view in VMD was required for consistency across figures. Specifically, applying a saved orientation matrix tailored to these constructs by entering the following command:

```
set f [open "view_matrix.txt" r]; gets $f view; close $f; molinfo top set
{center_matrix rotate_matrix scale_matrix global_matrix} $view
```

These matrices were created by visualizing one of the systems and manually adjusting the view to be correct, then outputting the orientation matrix with VMD commands.

If the molecule being visualized is full-length p53, a different orientation matrix is applied to account for its larger scale:

```
set f2 [open "view_matrix2.txt" r]; gets $f2 view2; close $f2; molinfo top
set {center_matrix rotate_matrix scale_matrix global_matrix} $view2
```

If the p53 isoforms are being visualized, a different orientation matrix is applied:

```
set f3 [open "view_matrix3.txt" r]; gets $f3 view3; close $f3; molinfo top
set {center_matrix rotate_matrix scale_matrix global_matrix}; $view3
```

Second, obtain the corresponding sector residues from *run_sector_analysis*. The residues will be output with two formats, amino acid residue names with the corresponding numbers and just the residue numbers. Residue identifiers with chain letters (e.g., “THR253”), are incompatible with VMD. In the selected atoms tab, type "resid" and copy just the sector residue numbers. Turn the style to view, adjust the thickness of the sector residues to be .3, and turn the color to red. The visualization is now ready to be exported, by going to the render tab and saving the image as Tachyon. This image type is

unsupported by text editors so imagemagick's convert function in the bash terminal is necessary for timely conversion of the Tachyon files to PNG files. This is the command:

```
for f in *.tga; do convert "$f" "${f%.tga}.png"; done
```

The images are now ready for the thesis.

3.1.10 Flask

Flask is a lightweight web server gateway interface that enables servers to interact with Python code. This is perfect for displaying the results of this thesis, since all that is needed is a way to upload files, run Python code on those files, and output the results.

3.1.11 Other High Performance Computing Modules

There are other high performance Python modules not used in this thesis, such as CuPy, Cython, multiprocessing, and Numba. Although these modules are great tools for parallelization and GPU acceleration, the overhead generated by using these modules made execution longer than using only NumPy or Pandas.

3.2 MD Simulations

Molecular dynamics simulations output a trajectory, which the sector analysis is calculated upon. MD simulations focusing on p53 for this thesis used standard Thayer Lab protocol (Figure 3.2.1). The AMBER 16 molecular simulation package was utilized for parameterization and modeling. Solvent water

was represented using the TIP3P potential, while atomic interactions were modeled with the ff99SB force field (Lindorff-Larsen et al.). Monovalent Cl^- and K^+ ions were described using the ions08 model. To ensure accurate electrostatic interactions, particle mesh Ewald summation was applied with a 10 Ångstrom cutoff (Thayer et al.). The SHAKE algorithm was used to enforce the necessary constraints on hydrogen bond motions. The system underwent energy minimization before being gradually heated to 300K, with temperature regulation maintained via Berendsen coupling to a heat bath. Following an equilibration phase, the simulation was run for 200 nanoseconds using the pmemd code within a parallel computing architecture.

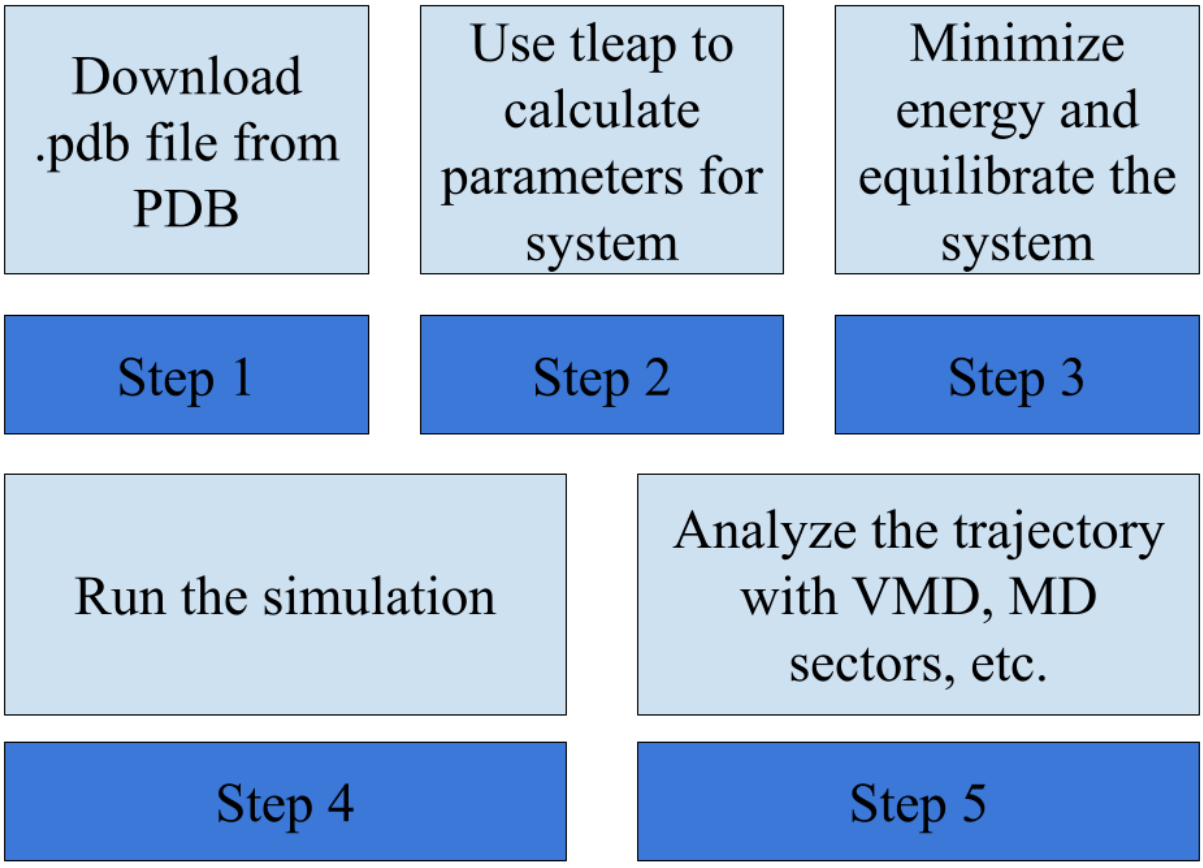


Figure 3.2.1 Basic MD simulation procedure

3.2.1 CPPTRAJ

CPPTRAJ is a program for analyzing and processing MD trajectories and associated datasets. CPPTRAJ is written mostly in C and C++ and is already highly optimized. Within this thesis, CPPTRAJ was used to gather correlation matrices from the MD trajectories.

CHAPTER 4

Results I: Creation of MD Sector Analysis Algorithm

4.1 MD Sector Analysis Algorithm

When starting to build the sector analysis algorithm, I had the mathematical formulas and some code that a previous lab member had written to identify sectors. The algorithm I was given was implemented in pure Python and took around 23 minutes to finish executing on the Y220C correlation matrix (193 x 193), on a 2020 Apple MacBook Air equipped with an Apple M1 chip (8-core CPU), with 8 GB of unified memory. For my first implementation, I looked for optimization opportunities and used pandas as the main data analysis module and it took ~1 minute to finish executing on a 2020 Apple MacBook Air. For the next implementation I used numpy as the main data analysis module, but still used pandas for its efficient csv loading capabilities. The NumPy implementation took ~0.02 seconds to fully execute on a Y220C correlation matrix on a 2020 Apple MacBook Air (Figure 4.1.1).

My main goals when revising the first implementation were clarity and efficiency. My first step was to create a function to see how long it takes for another function to finish executing (*time_func*). To use the time function, simply add it as a decorator for whatever function needs to be timed. After parsing through the first implementation, I had several attributes I wanted to change: use

of Python lists, inefficient indexing of matrices, single use functions — I wanted this code to be more abstract, so each function could be used many times, getting rid of single use global variables, scalability, capacity for throughput. For the first two qualms, there was a simple solution: Pandas. A Pandas dataframe is easier and faster to index than a 2-Dimensional Python list. It is built upon the NumPy architecture, which was coded in C, and there are functions like iLoc, which provide much more clarity for accessing elements in the matrix. I fixed the use of single use global variables by adding more cohesion between functions. For capacity of throughput, I used dynamic directory usage. The code just requires a directory for the .dat files and another directory for .pdb files, and then it could be executed.

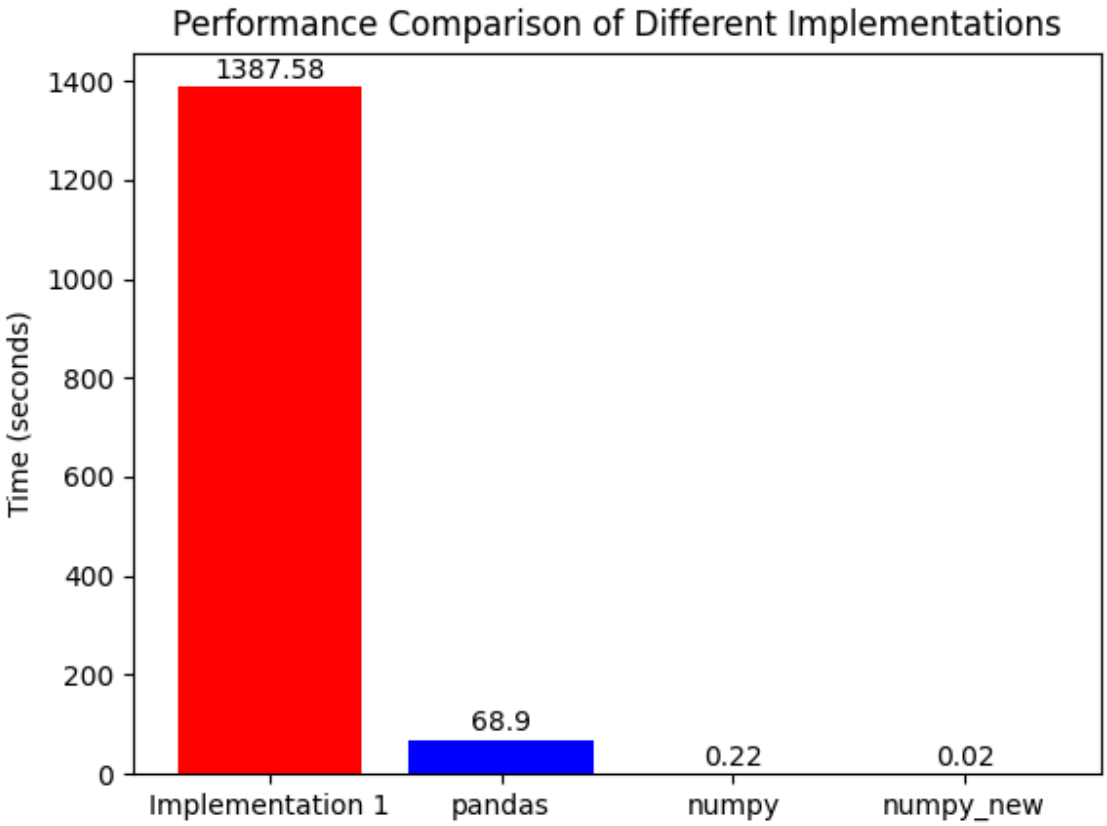


Figure 4.1.1 Bar graph displaying the execution time for each implementation of MD sectors. The numpy_new implementation is the same as the numpy implementation, but incorporates the change described in 4.3.6.

4.2 Pandas Implementation

Upon beginning the new implementation of Sectors, the first step was to brainstorm the major functions that were needed. First, a function was to turn all of the 1.0s that were along the diagonal to 0.0. This is named *zero_bonded_residues*. Next, a function responsible for sorting a symmetric matrix where the top left of the matrix had the highest values and the bottom right

had the lowest values. This is named the *rearrange_matrix* function. Finally a function would define a sector or the top 20% of a matrix, then swap in residues that were not in the top 20%, see if the total sector value was greater, keep the swap if the total value was greater, and if not keep iterating. I will refer to this as the *switch_values* function. Within *switch_values*, several other functions were necessary: one to cut out the sector, and another to find the total value of the sector.

Using the os module, one could dump a directory of different correlation matrices and see the sectors for each matrix. Generally, the structure of this followed: get all the names of files in a directory, then for each name save the analyzed matrix under the corresponding name, using f strings.

4.2.1 get_column_sums_and_sort

The purpose of this function is to return a series which has the residues with the highest total column value to the left and lowest to the right. The parameter of this function is the dataset that needs to be sorted. First, this function sums each column where each column represents a residue. The total column value for a residue is also known as a Vpica value (Fabry and Thayer). Second, it will sort them using the pandas *sort_values* function.

4.2.2 zero_bonded_residues

The purpose of this function is, given a dataset from pandas *read_csv*, find the 1.0's on the diagonal and convert them into 0.0. We do this because the residues will have a 100% correlation with themselves, so it will raise the total sector values. Figure 4.2.2.1 is an example of what the original matrix would like

as a heatmap. Figure 4.2.2.2 is an example of what a Zero Bonded Residue matrix would look like as a heatmap.

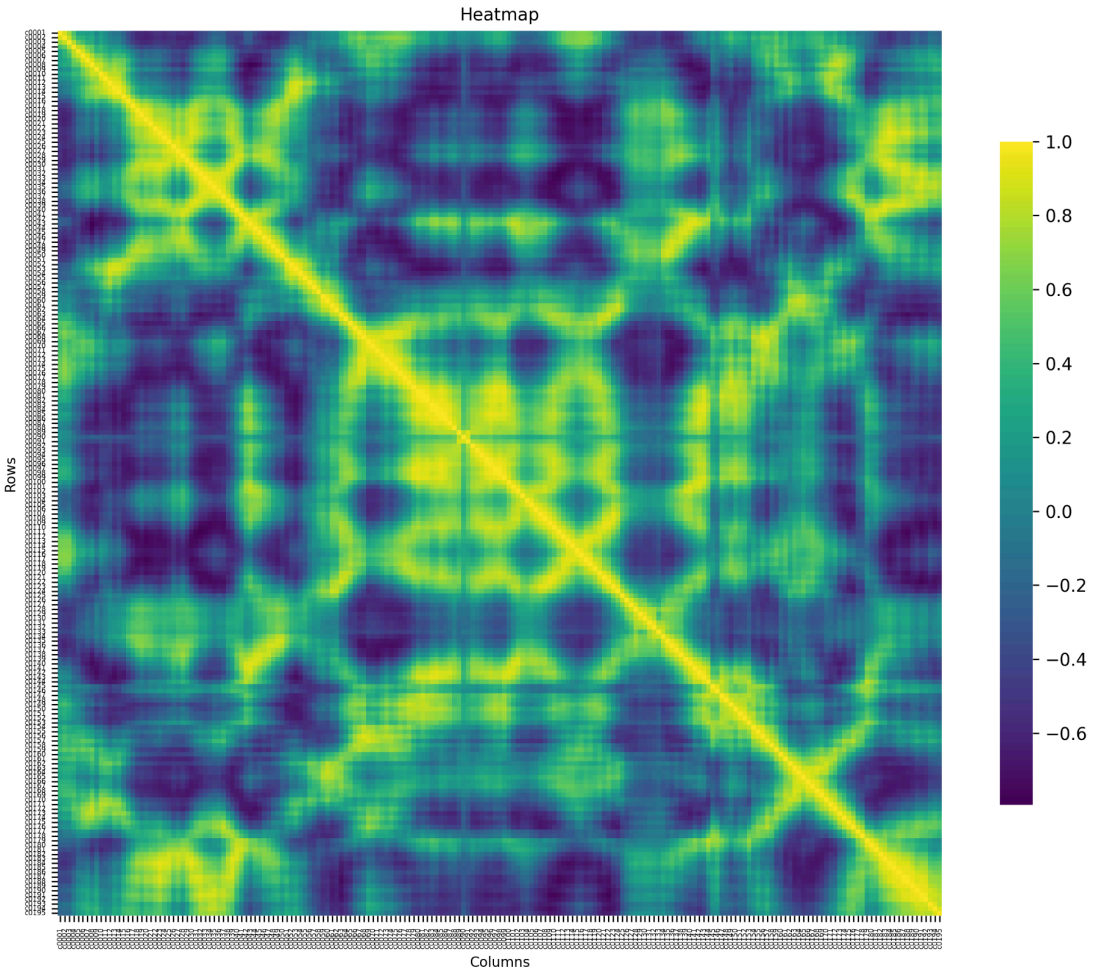


Figure 4.2.2.1 Heatmap of initial Y220C correlation matrix.

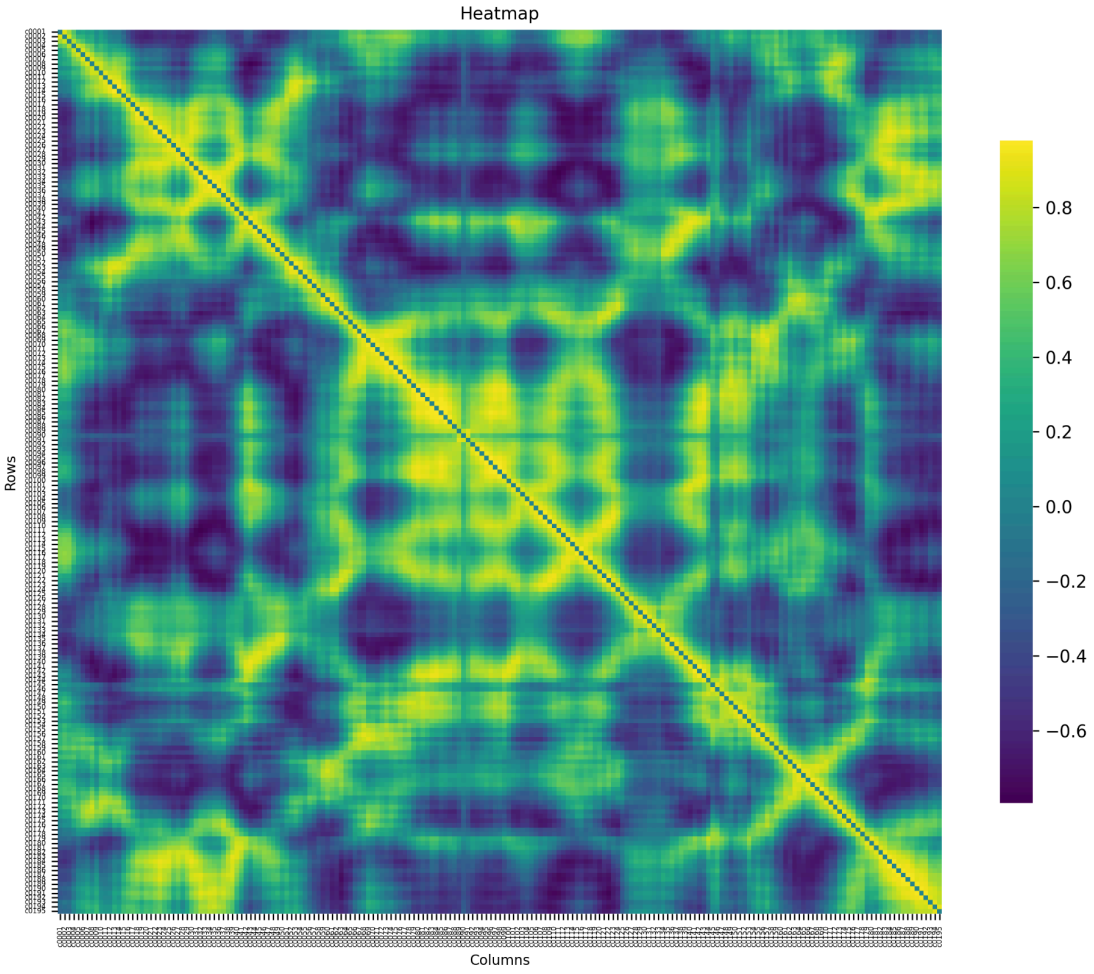


Figure 4.2.2.2 Heatmap of Y220C correlation matrix after *zero_bonded_residues* function.

4.2.3 rearrange_matrix

The purpose of this function is to create a sorted symmetric matrix, according to the values in the original dataset (*N* x *N*) and an index of columns (*N*) (Figure 4.2.3.1). The column index in this case are the corresponding labels for the sorted matrix. The sorted column index was used to define an outline of a dataset. This is found using the *get_column_sums_and_sort* function. The original

dataset was essentially used as a dictionary to look up the values that correspond to the dataset outline. Although this was faster than the first implementation, there was room for improvement. Each time this function would create another N x N matrix. First the *rearrange_matrix* function would be called for the initial rearrangement. Then it is called for each swap made in the *switch_values* function, N - (N / 5) times, for each residue outside of the sector, and the sector is 20% of the residues. This implementation used the *at* function to fill in the outlined dataset, which is useful for accessing a single cell, but repeated use is not recommended.

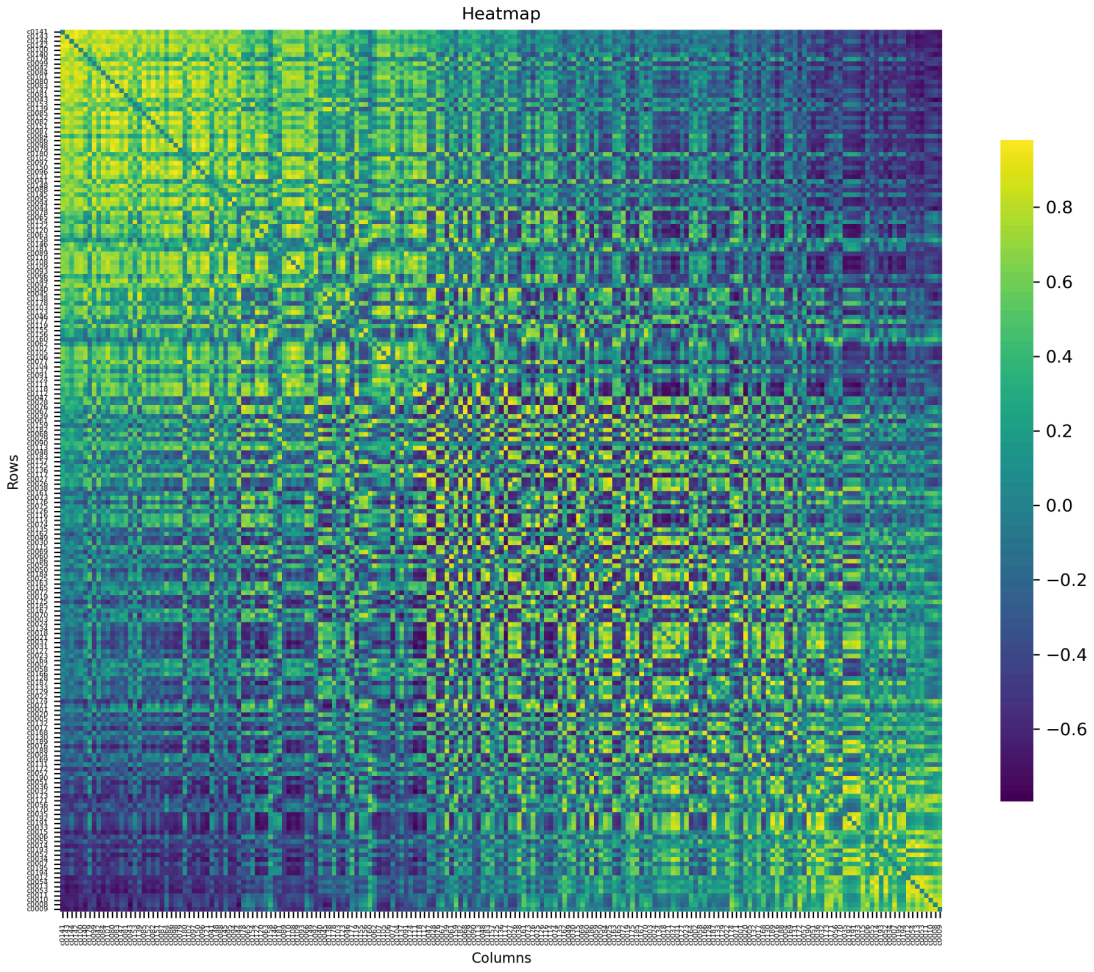


Figure 4.2.3.1 Heatmap of Y220C correlation matrix after *rearrange_matrix* function.

4.2.4 get_sector

The purpose of this function is, given a dataset and sector size, return the sector, using iLoc. The sector size is the amount of the distinct residues in the system divided by 5 rounding up.

4.2.5 get_sector_values

The purpose of this function is, given a sector, return the total value of the sector using the pandas sum function.

4.2.5 switch_values

The purpose of this function is to exhaustively find any residues outside of the sector, given by the *rearrange_mat* function, that create a greater total sector value when swapped with the last residue in the sector (Figure 4.2.5.1). We do this because some residues may be more correlated to the sector itself, and not the whole correlation matrix. The *rearrange_mat* function sorts based upon a residues correlation to the whole matrix, whereas the *switch_values* function refines the sector, maximizing total sector value by swapping in residues with more correlation to the sector itself. This is a feature that other implementations of MD sectors do not have. This function takes three parameters: a dataset that has had *rearrange_mat* applied to it, a Pandas series of the sorted residues not in the sector, and the lookup dataset—used as a lookup dictionary for each value. First, the sector is calculated using the *get_sector* function. Second, the total sector value is acquired through the *get_sector_values* function. The initial total sector value will be referred to as *curr*. Next, each residue outside of the sector is switched into the sector, in order to see if there is a higher total sector value.

This is done with a reutilization of the *rearrange_matrix* function, where it is called for each swap made N-(N/5) times, N being the number of unique residues in the matrix. Each time the *rearrange_matrix* is called the index of columns is different. This new index is the sorted index, except that the last residue in the sector was swapped with the ith residue outside of the sector. A new

dataset is constructed using the *rearrange_matrix* function. Then if the new total value of the sector was greater, the swapped in residue would become a part of the sector. Otherwise, it will keep iterating until there are no more residues to try.

After implementing these functions, it successfully calculated the sectors, but it was not as fast as I had hoped. Each execution of the switch value function led to $0.8N$ new data frames being created with *rearrange_mat.* If *rearrange_mat* were to be faster, it would speed up the total runtime significantly.

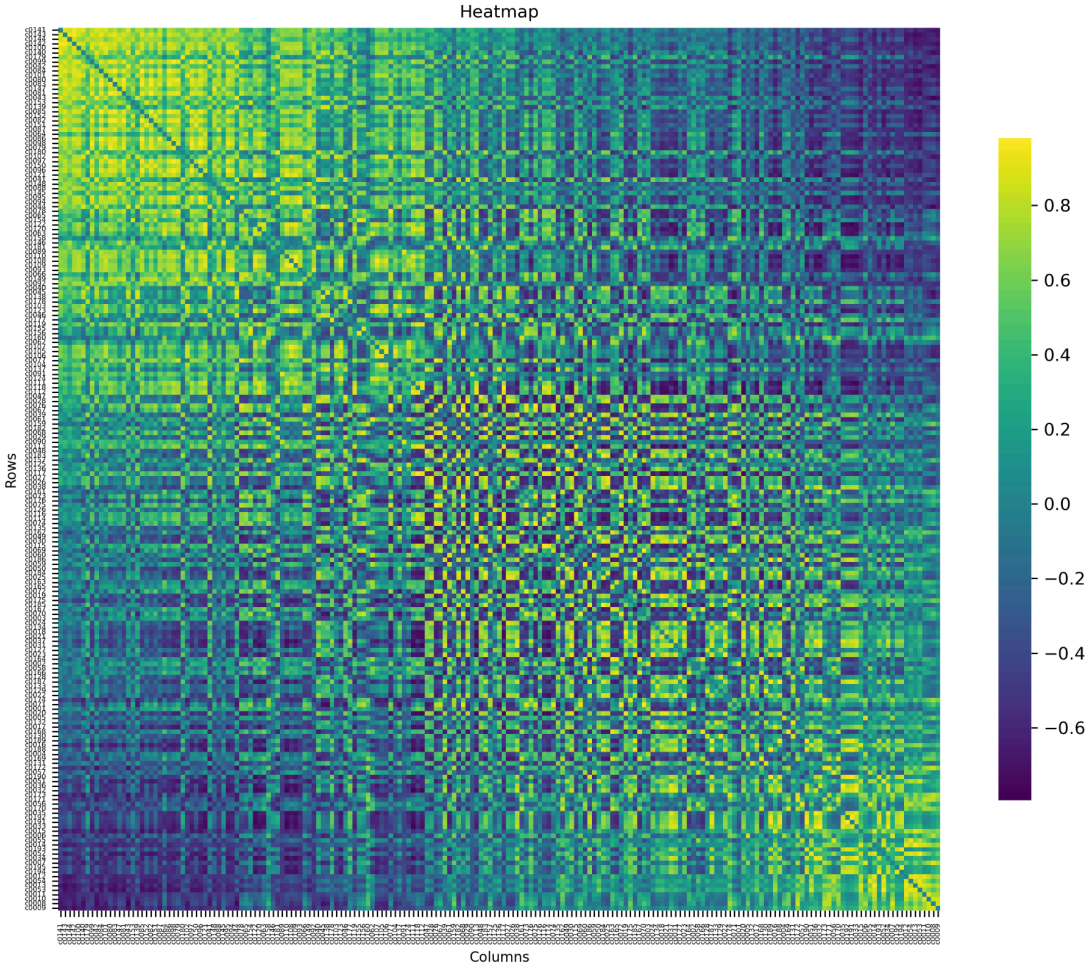


Figure 4.2.5.1 Heatmap of Y220C correlation matrix after *switch_values* function.

4.3 NumPy Implementation

The Pandas implementation achieved some of the goals that I wanted: readability, scalability, efficiency, and optimized throughput efficiency. However, there was room for improvement in efficiency. Specifically, the indexing techniques used throughout the code, and the *rearrange_mat* function could both be faster. If everything were array based and indexed by location, rather than by name of the residue, there would be a reduction in execution time. This involved ceasing the use of the pandas *at* function. Additionally, in-place operations appeared to be an interesting lead but per operation overhead proved to be greater than creating a new matrix and filling it in with the NumPy *ix_* function.

Consequently this motivated a NumPy implementation. There was now a clear logic established from the Pandas implementation. However, there was an issue: NumPy has no dataframe, which meant a higher difficulty implementing the indexing of the dataset. The solution was a new data structure, where instead of a data frame, a 2 dimensional NumPy array with the corresponding matrix values and a 1 dimensional NumPy array that kept track of the residues was used in place. Although Pandas is built upon NumPy, array based indexing in NumPy is much faster, especially as the matrices get bigger.

In addition to faster indexing, switching to NumPy had another benefit: the *ix_* function. This function creates an N dimensional array constructed from N 1 dimensional arrays, and then fills in the outline according to the values. The *ix_* function was essentially the *rearrange_mat* function but faster.

4.3.1 load_dat_file

The purpose of this function is, given a string representing a filepath, return a Tuple of data and column_labels. Data is the 2-Dimensional array reflecting the values in the correlation matrix and column_labels are a 1-Dimensional reflecting the corresponding labels. This function first uses the pandas *read_csv* function with a \s delimiter since it is loading a *.dat* file and not a *.csv* file.

4.3.2 get_column_sums_and_sort_np

The purpose of this function is identical to the pandas implementation but is executed slightly differently. The parameters of this function are a 2-Dimensional array (the dataset) and a 1-Dimensional array. First, using the NumPy *sum* function, it gets each column sum. Next, the indices that would sort the input 1-Dimensional array are retrieved using the NumPy *argsort* function with a descending order. Then the column labels are sorted according to that order and returned.

4.3.3 zero_bonded_residues_np

The purpose of this function is identical to the pandas implementation, but is more concise due to NumPy's *fill_diagnol* function. With *fill_diagnol* one can give the function two parameters an array of any dimensionality and a scalar, and it will return the same array but with the scalar being filled for the diagonal.

4.3.4 rearrange_matrix_np

The purpose of this function is identical to the *rearrange_mat* function. By using the NumPy *ix_* function, this version of rearranging the matrix is much faster.

4.3.5 get_sector_np

The purpose of this function is identical to the *get_sector* function from chapter 4.2.5.

4.3.6 switch_values_np

The purpose of this function is identical to the *switch_values* function from 4.3.6. The logic is also very similar, but instead of using *rearrange_matrix*, this function calls the NumPy *ix_*. When calculating the new sector value after switching another residue in, only the sector is needed to find this value. Before, *ix_* would be called to fill in the whole new matrix with the swapped residues, but that is redundant. It was then switched to fill in a matrix based only on the sector residues, and then gathering the total sector value. Instead of creating a new N x N dataset for each call, a (N/5) x (N/5) matrix was created instead. Upon testing with the Y220C correlation matrix, a 193 x 193 matrix, calculating the total sector value took 0.002 seconds instead of 0.02. This change led to around a 10 fold speedup in the *switch_values_np* function for most matrices.

CHAPTER 5

Results II: Applying the Algorithm for p53 Rescue

Having implemented the sectors method, the next step is to apply them to one of the lab's systems of interest: p53. When p53 mutates, it ceases to function, which enables uncontrolled cell growth, cancer. The Thayer Lab's goal is to create a drug that will reinstate p53 to its native function of suppressing tumors using allosteric therapeutics. The results from these analyses are a potential allosteric network that can communicate a signal across a protein from one site to another distinct site. Our lab hypothesizes these motionally covarying residues, known as the sector, could send an allosteric signal to reactivate mutant p53. After identifying the sector residues, the next step is to create 3 dimensional visualizations, projecting them onto each corresponding structure, in order to gain structural insights and look for important residues that reside in the sector. WT stands for wild type, which is a non-mutated version of a protein. In this case, the protein is p53. Y220C is a mutated version of the wild type, where Y mutated to C at residue 220. PK11000 is Y220C with the PK11000 drug bound to it.

5.1 Sector Analysis of p53

The following sectors for each study are listed in order from highest correlation value to lowest.

Here are the sectors that were ascertained from Y220C (Figure 5.1.1):

TYR236, CY2238, ASN239, MET237, ILE195, ASN235, VAL274, LEU194, LEU137, HE1179, ARG196, ARG175, HIE178, CY2242, CY2176, ALA138, ARG248, TYR234, GLU180, ASN247, PRO177, MET246, CYS182, ALA159, ARG181, HIE193, ARG174, CYS275, VAL197, GLN192, GLY245, PRO191, VAL216, GLN136, MET243, SER183, SER240, PRO190, ALA189

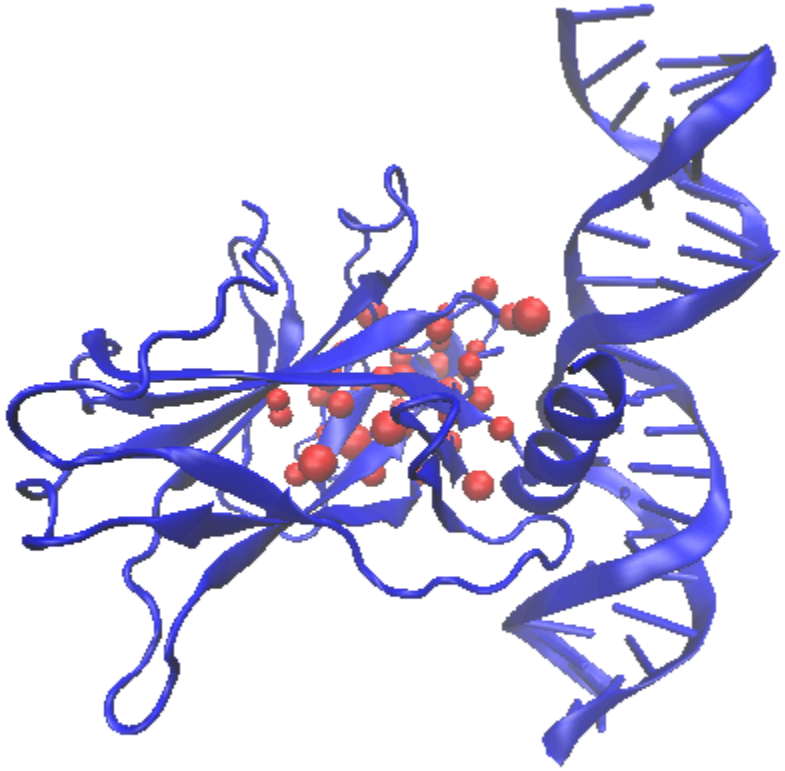


Figure 5.1.1 Sector residues (red) projected onto Y220C (blue).

Here are the sectors that were ascertained from PK11000 (Figure 5.1.2):

ILE195, LEU194, TYR236, THR253, CY2238, VAL272,

MET237, ALA161, ARG175, ASN239, HIE193, ILE251,

CY2176, ARG174, VAL173, LEU252, SER240, ALA138,

VAL274, MET246, ARG273, GLN192, CY2242, LEU137,

GLU180, ILE162, MET160, PRO177, HE1179, GLU271,

PRO191, ARG196, GLY245, HIE178, ASN247, ASN235,

SER241, ARG249, GLY244

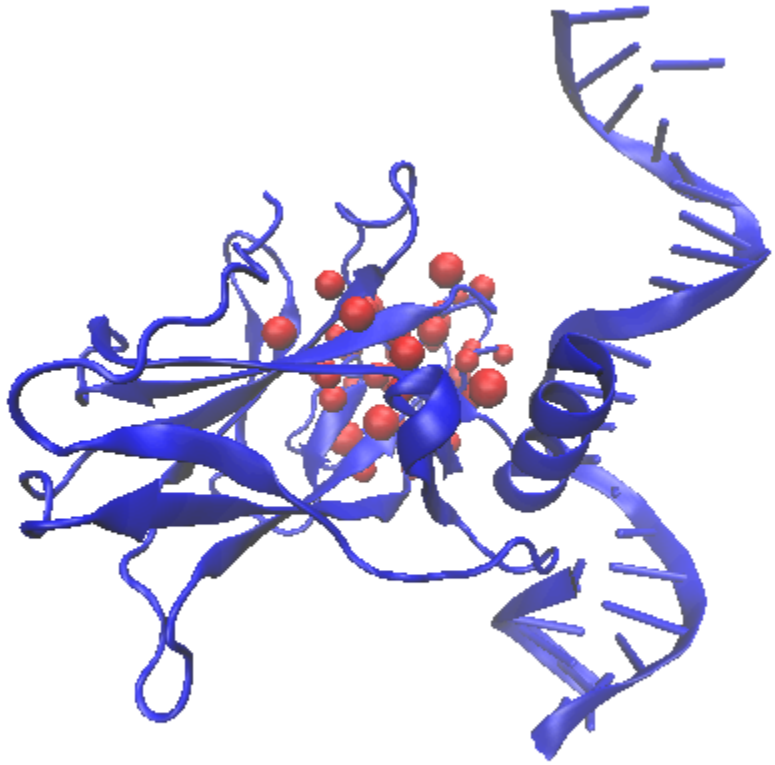


Figure 5.1.2 Sector residues (red) projected onto PK11000 (blue).

Here are the sectors that were ascertained from WT (Figure 5.1.3):

THR253, LEU252, ALA161, GLU271, ILE162, PHE270, VAL272, ILE254, MET160, TYR163, SER269, LYS132, LYS164, ILE251, MET133, ALA159, MET169, GLN165, SER166, THR170, ARG273, HIE168, GLN167, ILE255, VAL173, ASN131, GLU171, VAL172, ARG248, ARG249, ASN268, PHE134, PRO250, LEU130, TYR126, SER127, ILE195, SER215, SER240

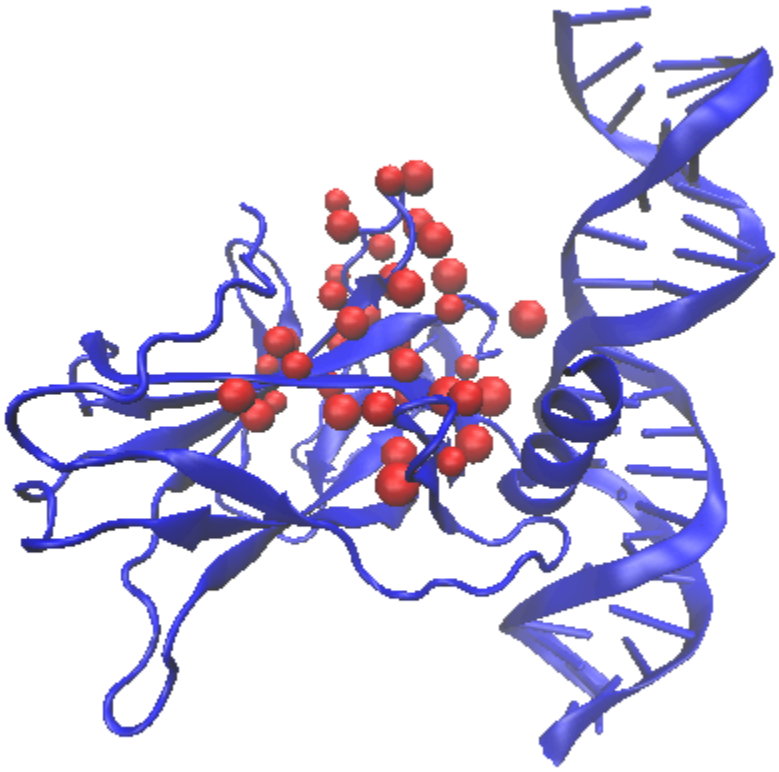


Figure 5.1.3 Sector residues (red) projected onto WT (blue).

There are 2 intersecting residues between Y220C, WT, and PK11000: ILE195 SER240. There are 4 intersecting residues between Y220C and WT: ILE195 ALA159 SER240 ARG248. There are 13 intersecting residues between PK11000 and WT: GLU271 SER240 ALA161 ILE195 ILE162 LEU252 VAL272 ARG273 VAL173 ARG249 ILE251 THR253 MET160. There are 26 intersecting residues between Y220C and PK11000: CY2242 SER240 CY2238 VAL274 PRO177 HIE178 CY2176 MET246 ARG196 TYR236 ILE195 ASN239 HE1179 GLU180 ASN247 GLN192 LEU137 ARG174 HIE193 ASN235 MET237 GLY245 ALA138 PRO191 LEU194 ARG175 (Figure 5.1.4).

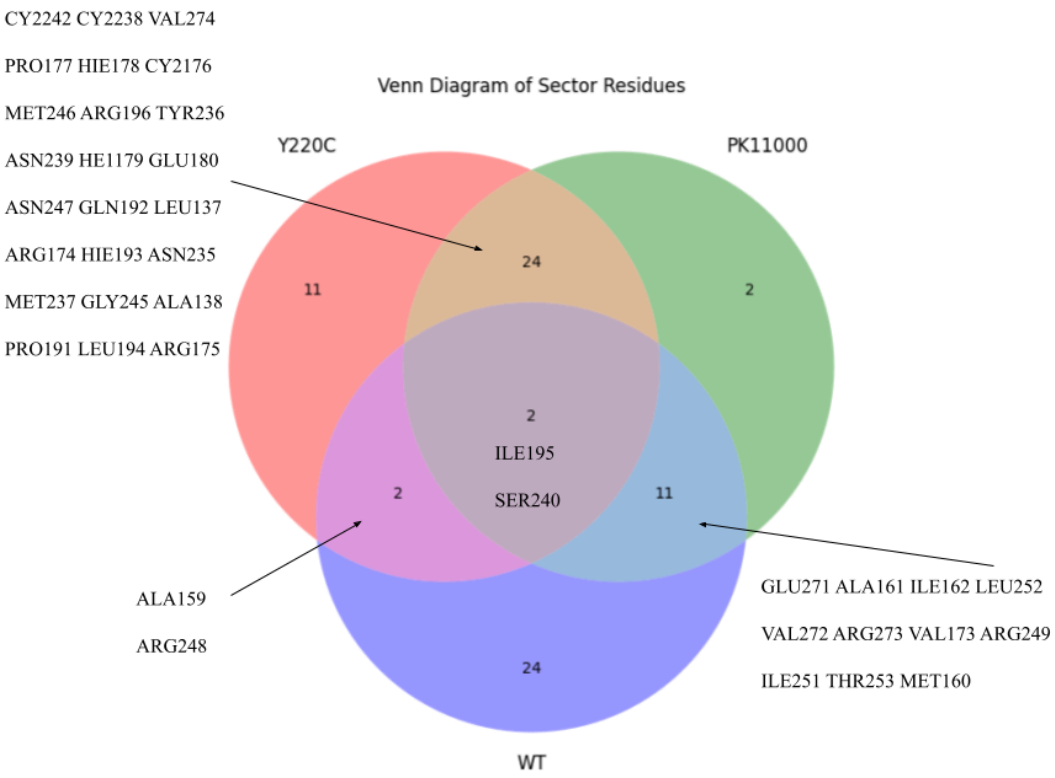


Figure 5.1.4 Venn diagram of intersecting sector residues.

5.1.1 Sector Analysis of p53: Key Findings

After identifying the sector residues, further analysis can be done by filtering non solvent accessible points or residues that are not visible on the surface of p53 using VMD. These points are the allosterically accessible points that a therapeutic should bind to.

There are 35 allosteric control points found for Y220C (Figure 5.1.1.1):

GLN136, LEU137, ALA138, ALA159, ARG174, ARG175,
CY2176, PRO177, HIE178, HE1179, GLU180, ARG181,
CYX182, PRO190, PRO191, HIE193, LEU194, ILE195,
ARG196, VAL197, VAL216, TYR234, ASN235, TYR236,
MET237, CY2238, ASN239, SER240, CY2242, MET243,
GLY245, ASN247, ARG248, VAL274, CYS275

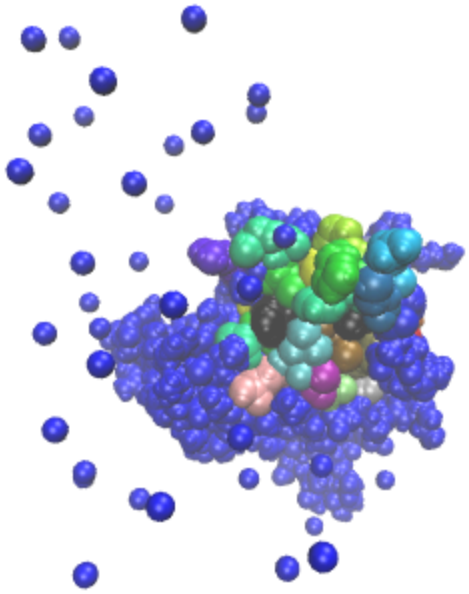


Figure 5.1.1.1 Allosteric points of control (not blue) projected onto Y220C (blue).

There are also 35 allosteric control points for WT (Figure 5.1.1.2):

TYR126, SER127, LEU130, ASN131, LYS132, PHE134,

ALA159, MET160, ALA161, ILE162, TYR163, LYS164,

GLN165, SER166, GLN167, HIE168, MET169, THR170,

GLU171, VAL172, VAL173, ILE195, SER240, ARG248,

ARG249, PRO250, ILE251, LEU252, THR253, ASN268,

SER269, PHE270, GLU271, VAL272, ARG273

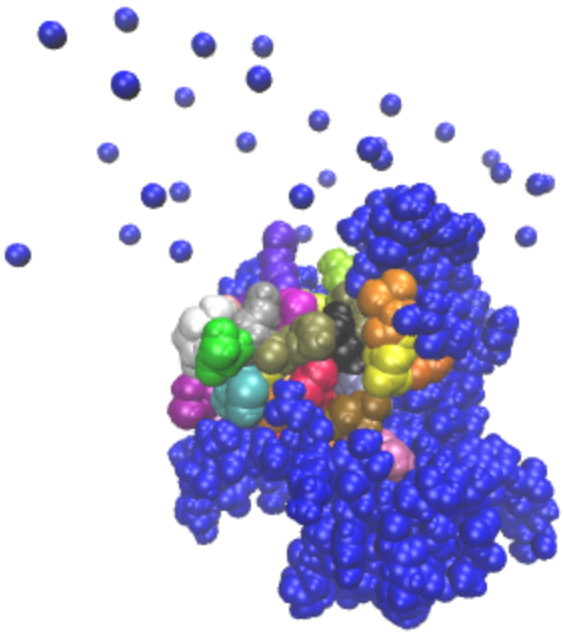


Figure 5.1.1.2 Allosteric points of control (not blue) projected onto WT (blue).

There 28 allosteric control points for PK11000 (Figure 5.1.1.3):

LEU137, ALA138, MET160, ILE162, ARG174, ARG175, CY2176, PRO177, HIE178, GLN192, LEU194, ILE195, ARG196, ASN235, MET237, CY2238, ASN239, SER240, SER241, CY2242, GLY244, ARG249, ILE251, LEU252, GLU271, VAL272, ARG273, VAL274

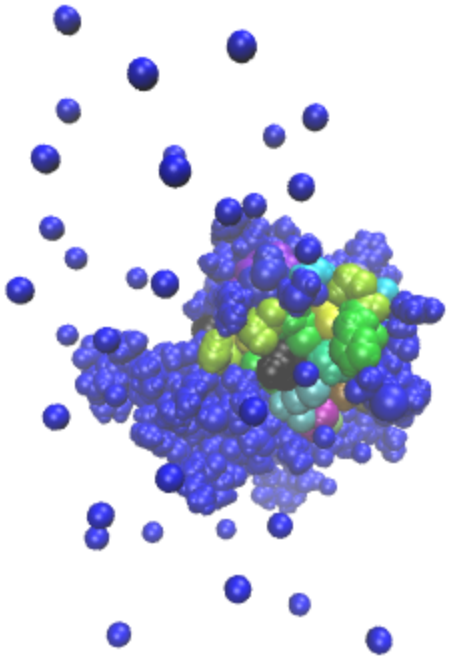


Figure 5.1.1.3 Allosteric points of control (not blue) projected onto PK11000 (blue).

5.2 Sector Analysis of the DBD' for p53 isoforms

By analyzing the DBD' for isoforms 2 to 9, consisting of residues 134 to 312, it enables analysis over a standard length of residues between each isoform. Each isoform differs in length, so a comparison between each of them, when they all have different sizes of sectors does not make sense. The goal of running sector analysis on isoforms 2 to 9 is to see how the behavior of the N and C terminal regions, or regulatory regions affect the behavior of the DBD' for each isoform. It

is difficult to capture the nature of these intrinsically disordered regions due to their dynamic tendencies. These analyses can provide insights on how the allosteric networks in each isoform are affected by the behavior of the N and C terminal regions and how different isoforms of p53 may exhibit variable responses to treatment.

5.2.1 Results

The following sectors for each study are listed in order from highest correlation value to lowest.

The following residues were identified from isoform 2 (Figure 5.2.1.1):

TYR234, ASN235, ALA138, CYS135, GLN136, PHE134, LEU137, LYS139, HIE233, THR140, TYR236, ARG273, LEU194, VAL197, ILE195, CYS275, ALA276, VAL274, MET237, CY2238, ILE232, CYS141, CYS277, VAL272, ARG196, HIE193, GLU271, ASN239, PRO278, LEU252, GLY279, HIE178, MET160, GLU198, SER240

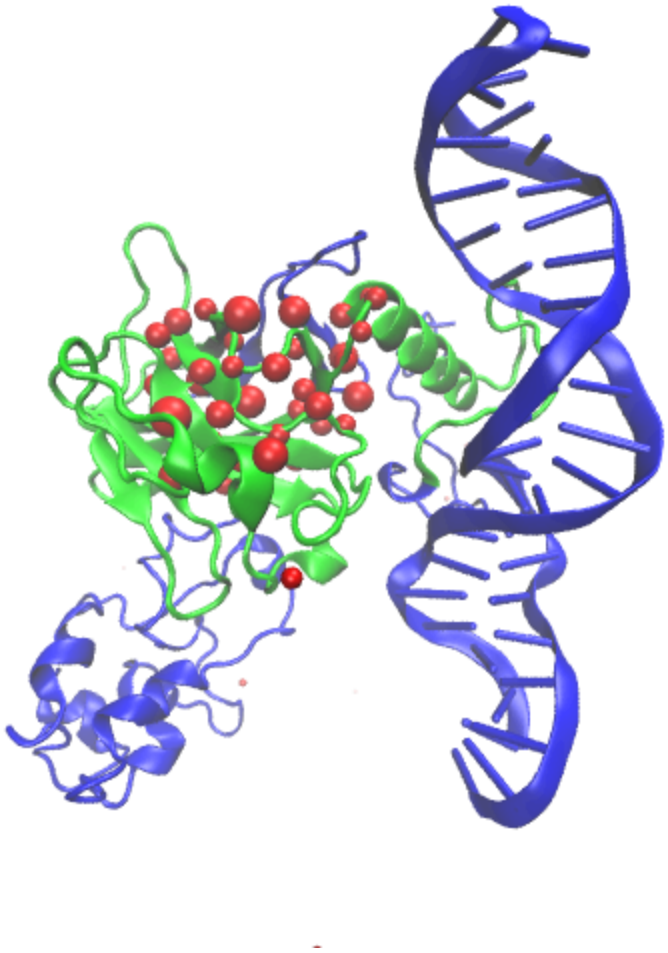


Figure 5.2.1.1 Sector residues (red) projected onto the DBD' (green) of isoform 2 (blue).

The following residues were identified from isoform 3 (Figure 5.2.1.2):

TYR234, HIE233, ALA138, LYS139, ARG196, ILE195, PHE134,

CYS135, ASN235, VAL197, THR140, ILE232, GLN136,

LEU137, GLU271, ARG273, LEU194, CYS277, THR231,

ALA276, ARG202, CYS141, VAL272, CYS275, ASN200,

LEU201, PRO278, TYR236, SER185, LEU188, VAL274,

CY2238, GLU198, THR230, SER183

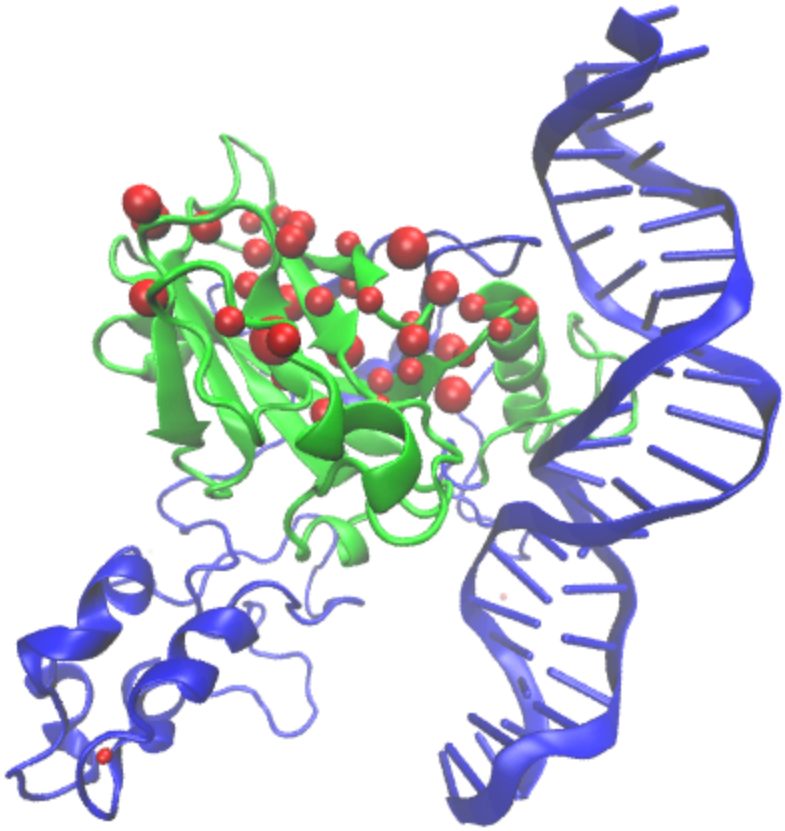


Figure 5.2.1.2 Sector residues (red) projected onto the DBD' (green) of isoform 3 (blue).

The following residues were identified from isoform 4 (Figure 5.2.1.3):

ASN235, TYR236, LYS139, THR140, TYR234, ALA138,

GLU198, MET237, VAL197, ARG196, CYS141, HIE233,

ILE195, LEU137, GLN136, GLY199, ASN200, CYS135,

PRO142, LEU194, CY2238, VAL272, ILE232, LEU201, VAL274,

VAL143, THR253, SER183, PRO191, HIE193, ALA189, HE1179,

VAL203, THR231, ASP184

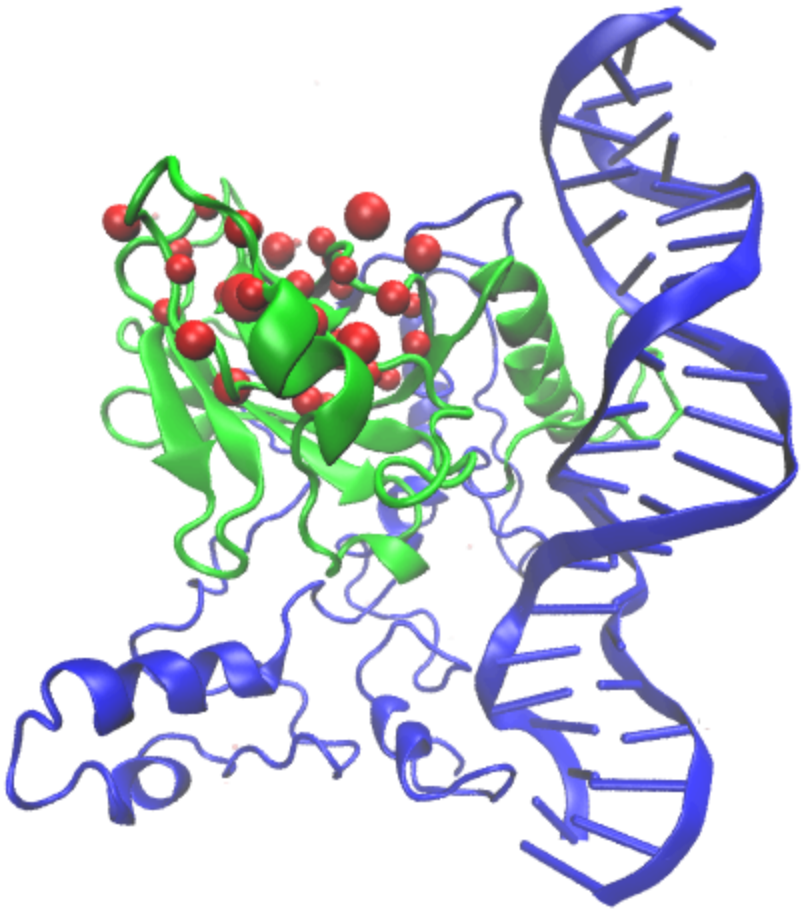


Figure 5.2.1.3 Sector residues (red) projected onto the DBD' (green) of isoform 4 (blue).

The following residues were identified from isoform 5 (Figure 5.2.1.4):

TYR236, ASN235, ALA138, LEU137, LYS139, GLN136,

MET237, TYR234, ILE195, THR140, ARG196, CYS135,

CY2238, VAL197, GLU198, LEU194, CYS141, VAL274,

HIE233, ASN239, CYS275, HE1179, ALA276, HIE193, THR253,

ARG175, CYS277, PRO191, PRO142, PHE134, ALA189,

ILE232, ALA161, GLU180, GLN192

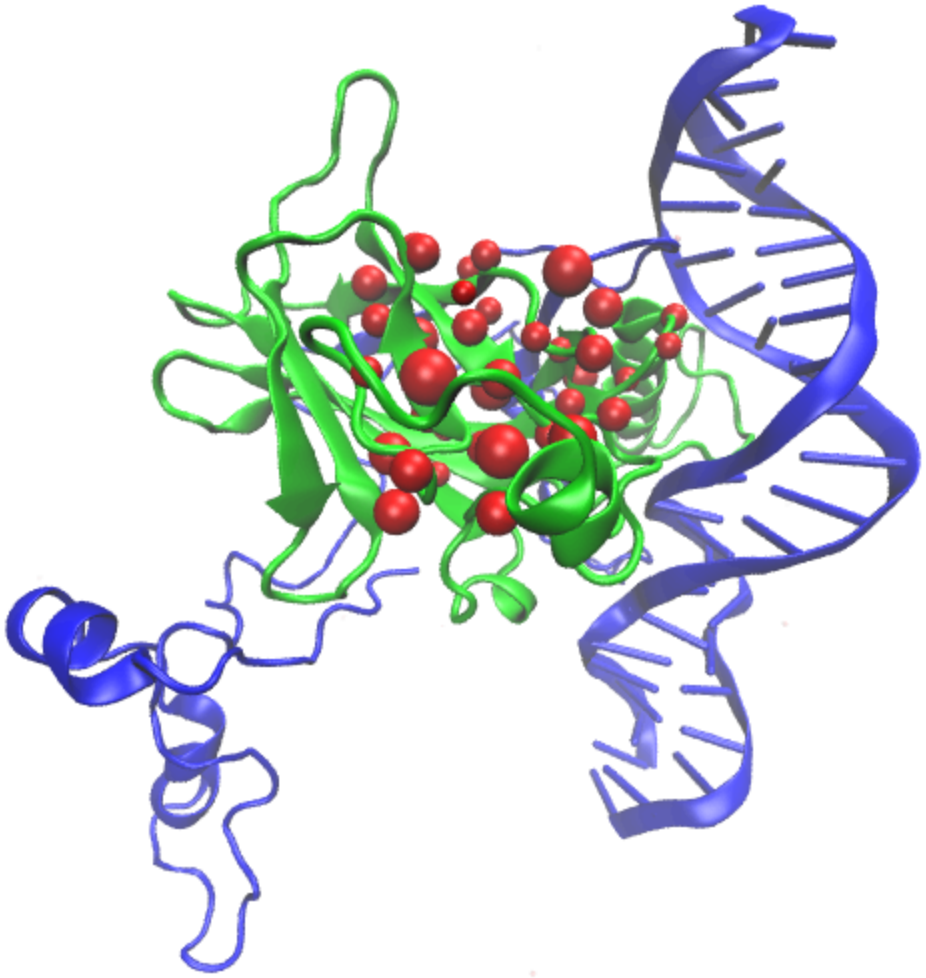


Figure 5.2.1.4 Sector residues (red) projected onto the DBD' (green) of isoform 5 (blue).

The following residues were identified from isoform 6 (Figure 5.2.1.5):

TYR236, ALA138, MET237, ARG196, ASN235, LYS139,

ILE195, LEU194, LEU137, THR140, TYR234, CY2238,

GLN136, HIE193, PRO191, VAL197, HE1179, CYS135, ASP186,

GLU198, ALA189, CYS182, ARG175, GLN192, ARG181,

PRO190, CYS141, ASN239, VAL274, HIE233, CY2176, HIE178,

SER183, GLU180, PRO177

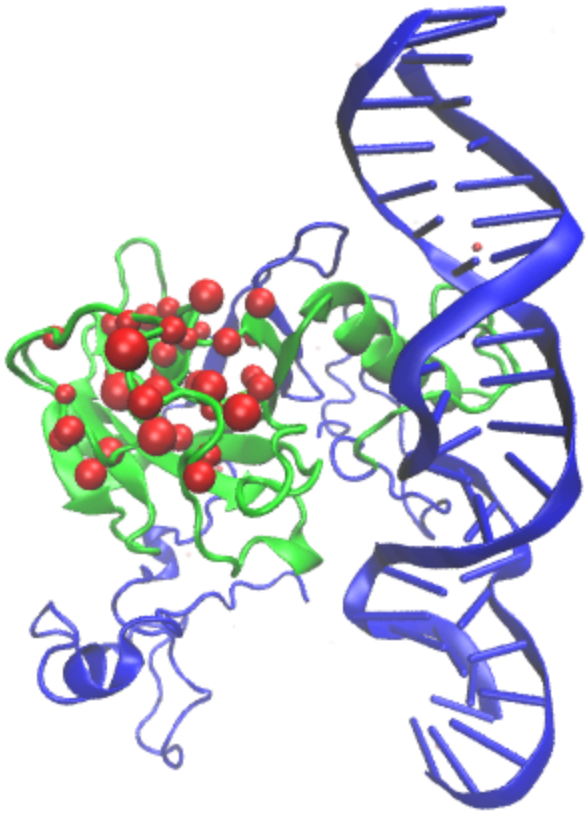


Figure 5.2.1.5 Sector residues (red) projected onto the DBD' (green) of isoform 6 (blue).

The following residues were identified from isoform 7 (Figure 5.2.1.6):

ASN235, GLU198, TYR234, ARG196, VAL197, LEU201,

GLY199, HIE233, GLU204, ASN200, VAL203, PRO219,

VAL217, ARG202, TYR236, PRO222, CYS220, TYR205,

VAL218, VAL216, LEU206, MET160, THR140, CYS141,

GLU221, ASP207, ILE195, ILE232, SER215, ALA159, ALA161,

ILE254, LEU194, ARG158, PRO190

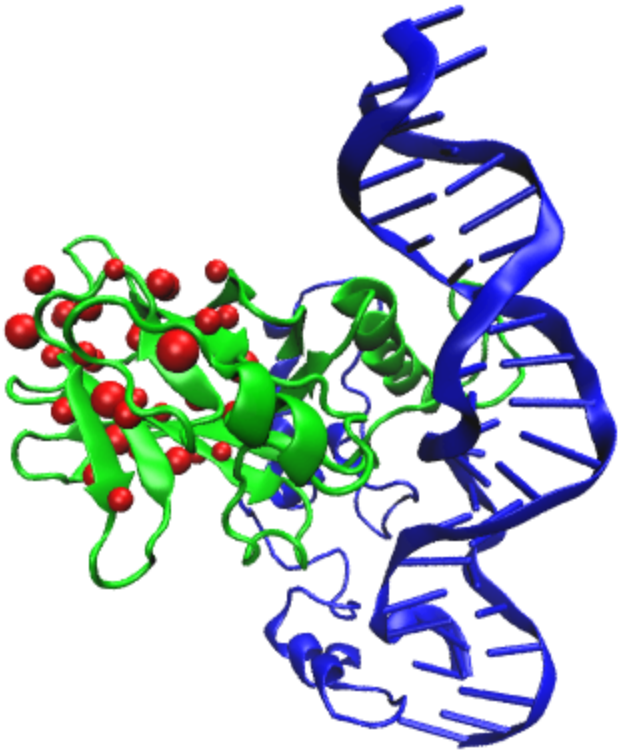


Figure 5.2.1.6 Sector residues (red) projected onto the DBD' (green) of isoform 7 (blue).

The following residues were identified from isoform 8 (Figure 5.2.1.7):

ARG196, VAL217, VAL197, TYR205, GLU204, LEU206,

VAL218, PRO219, ASN235, GLU198, MET237, TYR236,

ASP207, VAL216, VAL203, MET160, ILE195, LEU194,

ALA159, CYS220, PRO190, ARG202, ARG158, CY2238,

PRO191, ASP208, ALA189, VAL157, TYR234, SER215,

GLY199, ALA161, HIE193, GLN192, LEU188

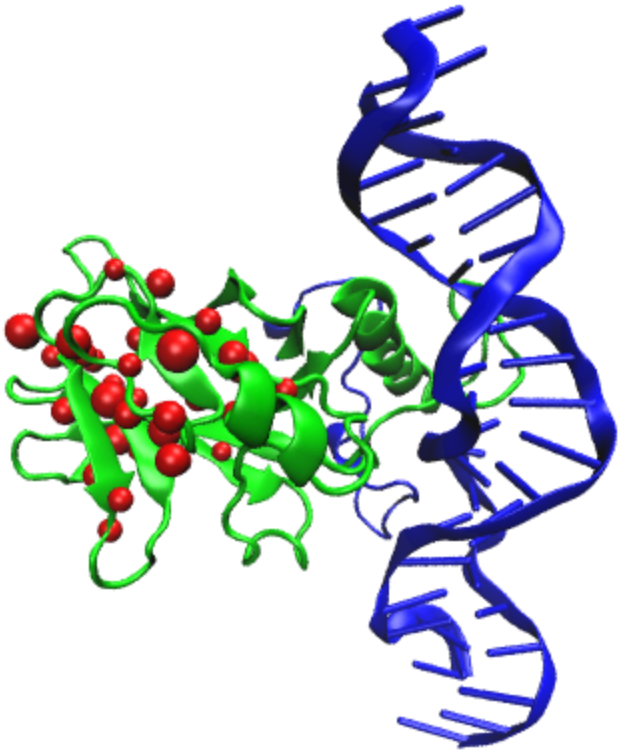


Figure 5.2.1.7 Sector residues (red) projected onto the DBD' (green) of isoform 8 (blue).

The following residues were identified from isoform 9 (Figure 5.2.1.8):

ARG196, PRO190, ALA189, ILE195, VAL197, PRO191,

LEU206, MET237, LEU194, TYR205, GLN192, LEU188,

CY2238, GLU204, VAL217, TYR236, HIE193, GLY187,

VAL203, ASP186, SER185, ASP207, GLU198, ARG202,

ASP184, VAL218, VAL216, ASN235, PRO219, ARG181,

SER215, ASN239, LEU201, CY2176, CYS182

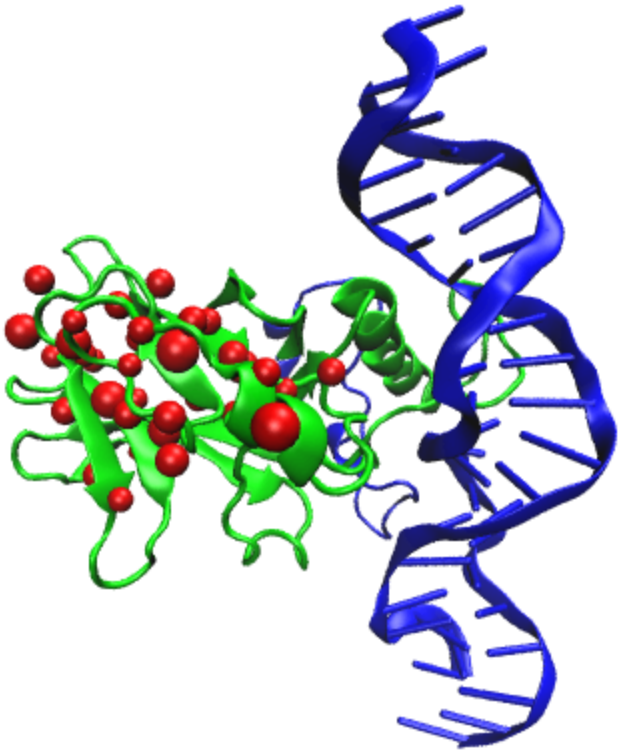


Figure 5.2.1.8 Sector residues (red) projected onto the DBD' (green) of isoform 9 (blue).

5.2.2 Key Findings

The following results were identified when comparing each of the isoform sector residues to the hydrogen bonding sites included in the DBD' (residues 241, 248, 273, 276, 277, 280, 283). Isoforms 2 and 3 had 3 intersecting residues with the hydrogen bonding residues: 277, 273, 276. These isoforms both begin at IVR 2, and are closely related in length, with Isoform 3 just being 5 residues longer. Isoform 5 and the hydrogen bonding residues had 277 and 276 in common.

Isoforms 4, 6, 7, 8, 9 had no intersecting residues with the hydrogen bonding residues.

The following results were identified when comparing each of the isoform sector residues to the hot spot residues (residues 175, 220, 248, 273, 282). Isoforms 4 and 9 had no residues in common with the hot spots. Isoforms 2 and 3 had residue 273 in common with the hotspots. These two isoforms include IVR 1, but do not have the same length as isoform 3 includes IVR 3, while isoform 2 has neither IVR 3 or IVR 4. Isoforms 5 and 6 had residue 175 in common with the hotspots. These two isoforms begin at IVR 2, while isoform 6 has IVR 3, but isoform 5 has neither IVR 3 or IVR 4. Isoforms 7 and 8 had residue 220 in common with the hotspots. These isoforms do not include IVR 1 or IVR 2, and start at the DBD'. Isoform includes IVR 3 and IVR 4, while isoform 8 includes neither. An interesting trend is that, with the exception of isoforms 4 and 9, isoforms that begin at the same IVR region each have another isoform with the same intersecting hot spots within their corresponding sectors. Isoform 4 is the first isoform in the set of isoforms beginning at IVR 2 and isoform 9 is the last isoform in the set of isoforms beginning at the DBD'.

5.3 Sector Analysis of full length p53 isoforms

The goal of running sector analysis on p53 isoforms 2 to 12 is to see which residues on each isoform have the most correlated motion. Since isoforms 10 to 12 begin at residue 160, they do not include the DBD'. Currently, the concept of a DBD" is being investigated within the lab, where the DBD" begins at residue 160 and ends at IVR 3. Each isoform has varied lengths, which means

each isoform's sector residues will vary in length as well. Consequently, each isoform's sectors will be compared with the hot spot and hydrogen bonding sites on p53, rather than the other isoform's sectors, due to the varied lengths.

5.3.1 Results

The following sectors for each study are listed in order from highest correlation value to lowest.

The following residues were identified from isoform 2:

MET160, ALA159, ARG196, VAL216, ILE195, TYR234,
VAL197, SER215, ALA161, ASN235, GLU198, HIE193,
VAL203, LEU201, TYR205, THR253, PRO190, TYR236,
ARG202, LEU194, ILE232, ALA189, PRO191, HIE233, ASP184,
THR140, VAL217, MET237, GLU204, HIE214, VAL218,
CYS141, GLU224, ARG158, ASN200, ILE254, GLN192,
LYS139, ALA138, PRO142, GLY199, VAL157, ILE255, LEU206,
CYS182, LEU137, THR231, CY2238, ARG181, PRO219,
VAL143, GLU180, SER183, HE1179, SER185, ARG175,
GLN136, THR123, LEU188, VAL225, CYS135, THR230,
ARG174, ASN239, CYS124, SER227, ARG213, GLY187

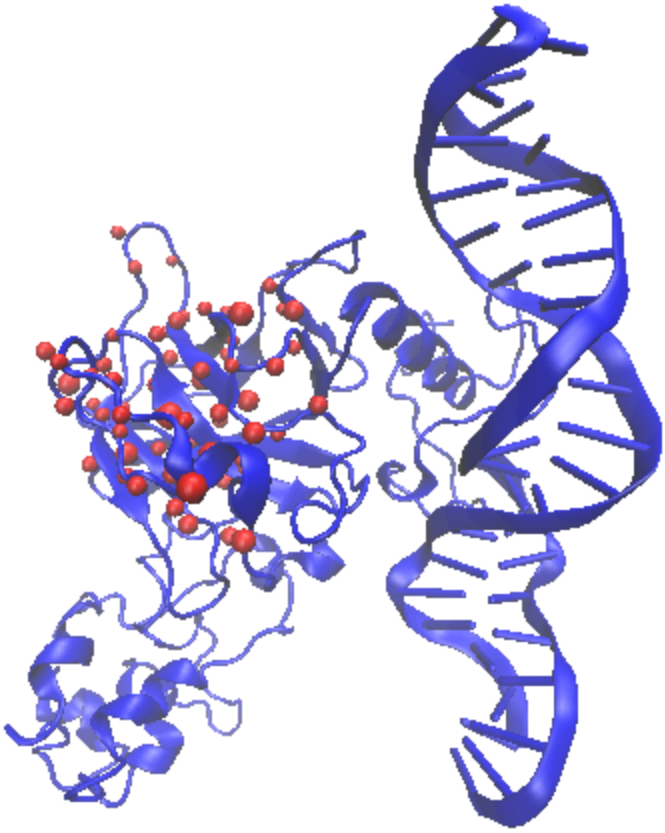


Figure 5.3.1.1 Sector residues (red) projected onto isoform 2 (blue).

The following residues were identified from isoform 3:

VAL272, THR253, GLU271, ASN235, TYR234, ARG196,

CYS135, MET133, TYR236, VAL197, PHE134, THR123,

GLN136, CYS124, ILE195, CYS141, THR140, VAL274, LYS139,

LEU252, ARG273, VAL122, THR125, GLU198, LEU137,

HIE233, PRO278, PHE270, ILE232, ALA138, SER121, LYS132,

CYS277, TYR126, VAL203, CYS275, PRO142, MET237,

VAL216, ALA276, VAL143, SER116, ASN239, ALA159,

GLY279, LYS120, THR231, ARG202, SER240, ARG282,

ALA189, ALA119, THR118, SER127, GLY117, ILE251,

LEU201, LEU188, SER215, VAL218, ALA161, ASN200,

ASP186, VAL217, MET160, HIE115, ASP281, SER241, ARG280

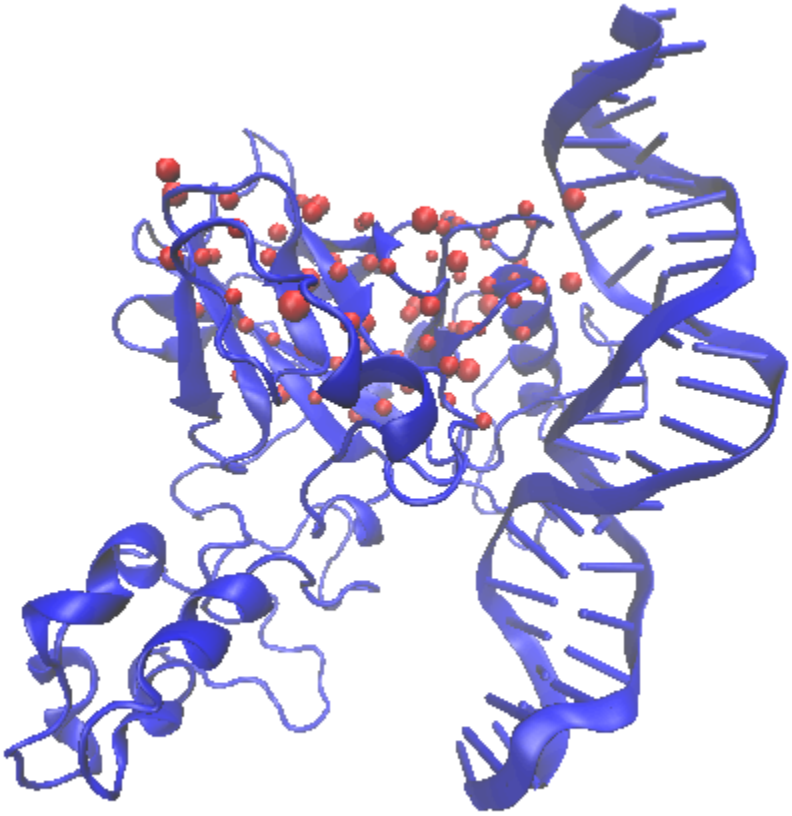


Figure 5.3.1.2 Sector residues (red) projected onto isoform 3 (blue).

The following residues were identified from isoform 4:

VAL272, GLU271, MET133, CYS135, CYS124, LYS132,

ASN235, TYR236, PHE270, LYS139, PHE134, THR123,

THR125, GLN136, TYR126, CYS141, THR140, TYR234,

ALA138, VAL274, LEU137, SER127, ARG273, THR253,

HIE233, ASN131, PRO142, VAL122, PRO278, VAL143,

MET237, GLU198, PRO128, VAL197, CYS275, CYS277,

LEU252, GLY199, ALA276, GLY279, GLN144, ARG196,

ILE195, THR231, ILE232, GLY112, CY2238, ARG282, ILE251,

SER121, ASN200, PHE113, ASP281, LEU130, ASN239,

ALA119, ARG280, SER269, LYS120, ALA129, LEU111,

THR118, GLU336, SER116, GLY117, LEU194, THR230,

ALA161, ARG283, HIE115

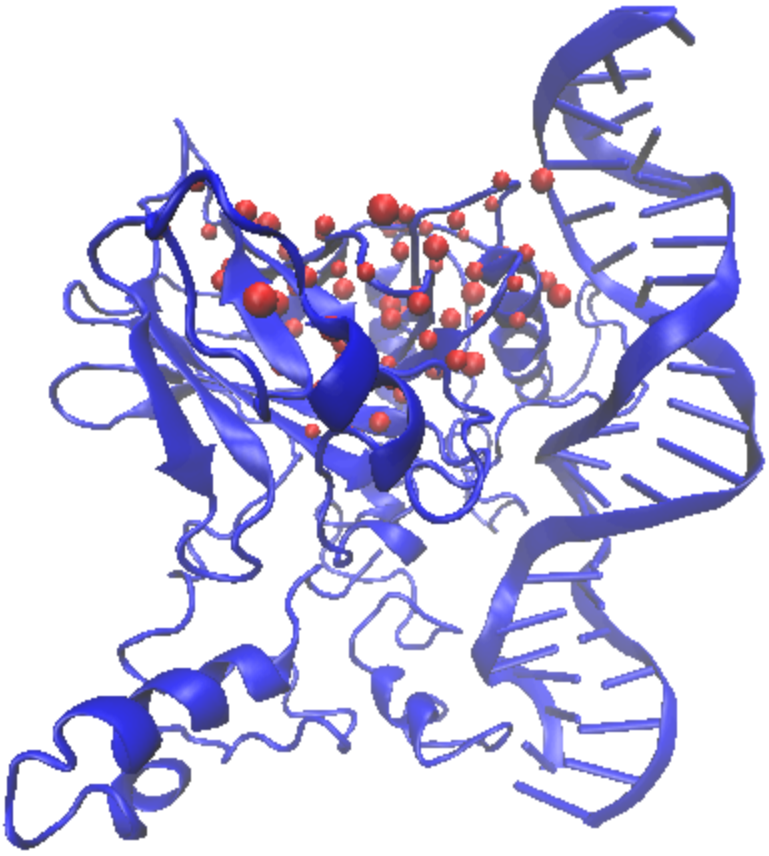


Figure 5.3.1.3 Sector residues (red) projected onto isoform 4 (blue).

The following residues were identified from isoform 5:

CYS135, GLN136, THR253, TYR236, CYS124, THR123, LEU137, VAL274, ASN235, LYS139, VAL272, CYS141, TYR234, THR140, ALA138, PHE134, VAL122, CYS275, MET237, ILE195, MET133, HIE233, CY2238, ARG273, SER121, THR125, PRO278, ARG196, ASN239, ALA276, CYS277, VAL197, GLU198, PRO142, LEU194, ALA161,

LYS120, VAL143, ILE232, GLY279, SER116, MET160, ILE251,

ILE254, LEU252, TYR126, ALA159, GLU271, THR118,

ARG280, SER240, ARG175, ILE255, ALA119, HIE193, GLY117,

HE1179, CY2242, ASP281, SER241

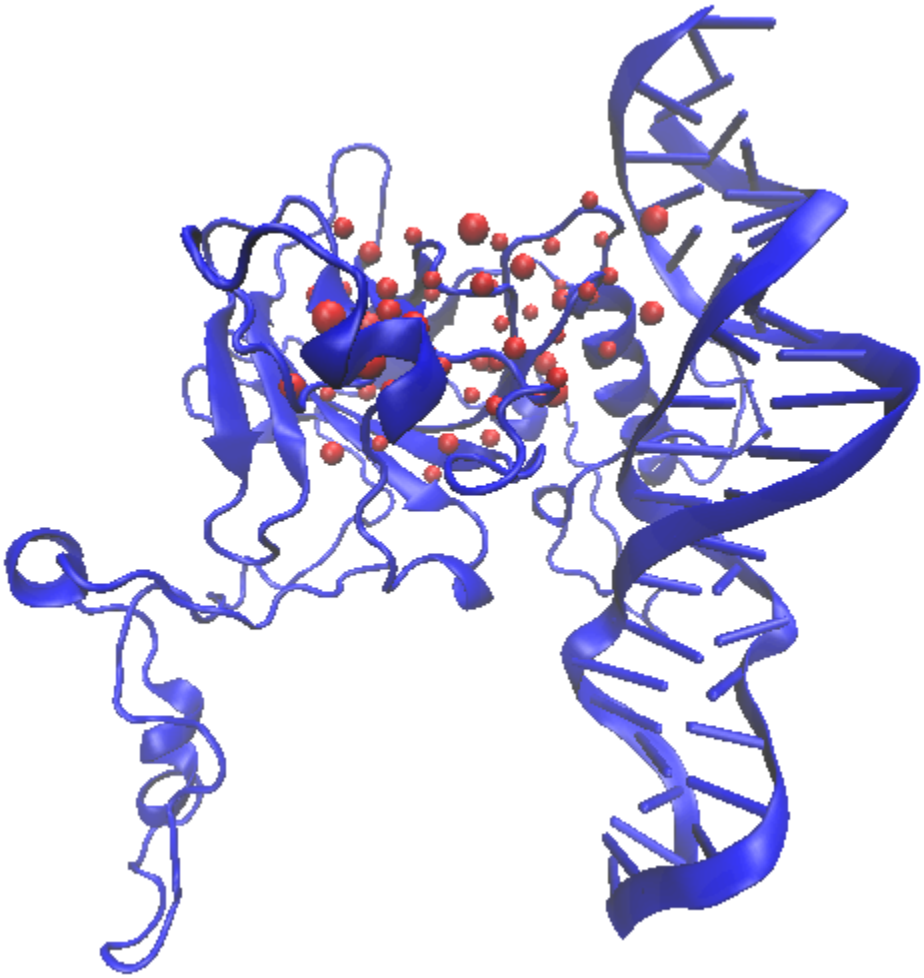


Figure 5.3.1.4 Sector residues (red) projected onto isoform 5 (blue).

The following residues were identified from isoform 6:

ASN235, TYR236, TYR234, ARG196, ALA138, ILE195,

LYS139, THR253, THR140, MET237, CYS135, VAL197,

GLN136, LEU194, LEU137, CYS141, THR123, GLU198,

HIE233, CYS124, CY2238, HIE193, VAL274, ALA161, PRO191,

ASP186, PRO142, ALA189, VAL272, CYS182, ARG175,

ASN239, HE1179, GLN192, PRO190, PHE134, SER183,

GLY199, ARG181, CY2176, LEU188, ALA276, CYS275,

MET133, LEU252, HIE178, VAL122, GLY187, ILE162, HIE214,

ASP184, ARG174, MET160, ILE251, LEU201, SER185, ILE232,

ARG273, THR125, PRO278, PRO177

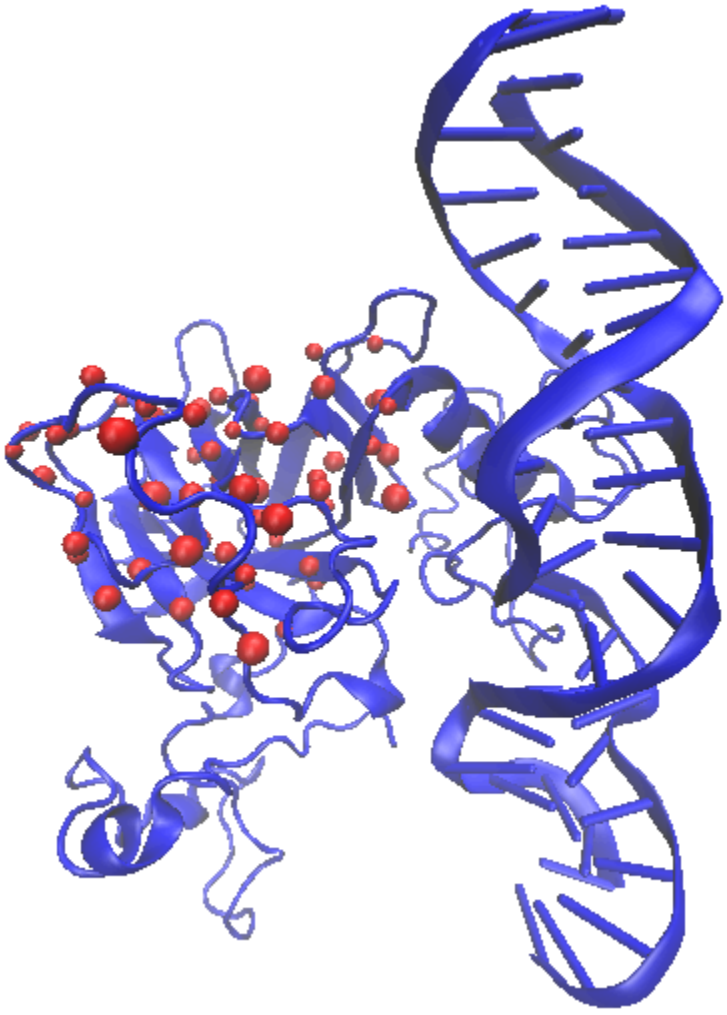


Figure 5.3.1.5 Sector residues (red) projected onto isoform 6 (blue).

The following residues were identified from isoform 7:

ILE251, LEU252, VAL272, THR253, ILE162, TYR163, ALA161, GLU271, SER269, PHE270, ILE254, LYS164, MET160, VAL173, ILE195, ILE255, MET169, MET133, ALA159, TYR234, PRO250, ASN268, VAL172, SER215, THR170, THR256, HIE168, HIE214, ARG158, GLU171, ARG267, GLN165, VAL216, GLY266, ILE232, ARG213, VAL157, LEU257, LEU265, ASP207, ARG196, VAL217, LEU264, LEU145, HIE233, VAL218, ASP259, GLU258, ASP208, GLU221, LEU206, ARG209

Figure 5.3.1.6 Sector residues (red) projected onto isoform 7 (blue).

The following residues were identified from isoform 8:

TYR234, ILE195, TYR236, ASN235, VAL216, ARG196, ALA159, VAL217, VAL203, GLU204, VAL197, VAL218, TYR205, ARG158, SER215, HIE233, ARG202, VAL157, LEU194, MET160, LEU206, PRO219, ILE232, MET237, ALA161, HIE193, THR253, THR140, LEU201, ALA189,

GLU198, ARG156, CYS141, ILE255, ALA138, ASP207,

LYS139, CYS220, HIE214, LEU188, PRO190

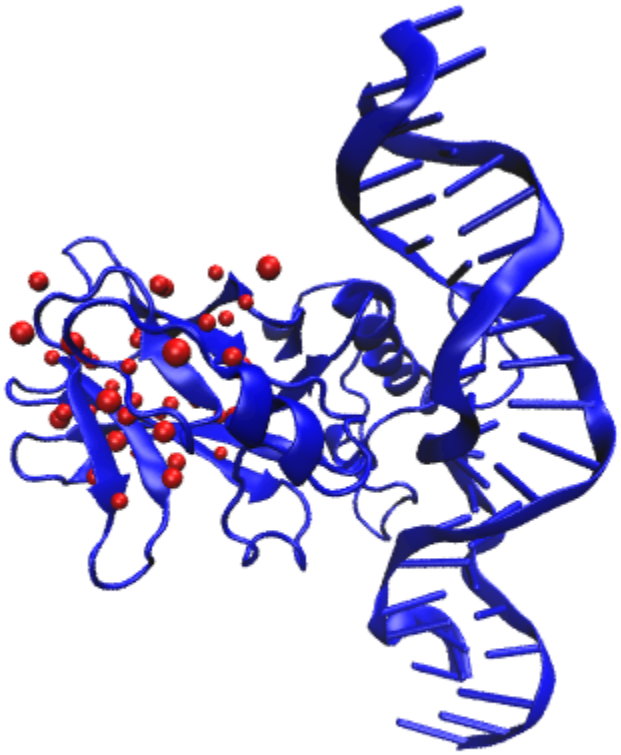


Figure 5.3.1.7 Sector residues (red) projected onto isoform 8 (blue).

The following residues were identified from isoform 9:

ILE195, LEU194, TYR236, HIE193, MET237, ALA189,

PRO190, PRO191, ARG196, LEU188, GLN192, TYR205,

CY2238, VAL216, ASN235, GLY187, ARG175, GLU204,

SER183, GLU180, ASP184, SER215, HIE214, LEU206, ASP186,

SER185, VAL203, ARG174, ARG181, VAL217, VAL197,

TYR234, HE1179, LEU201, ARG202, ASP207, CYS182,

VAL173, ALA138, CY2176, ALA161, PRO177

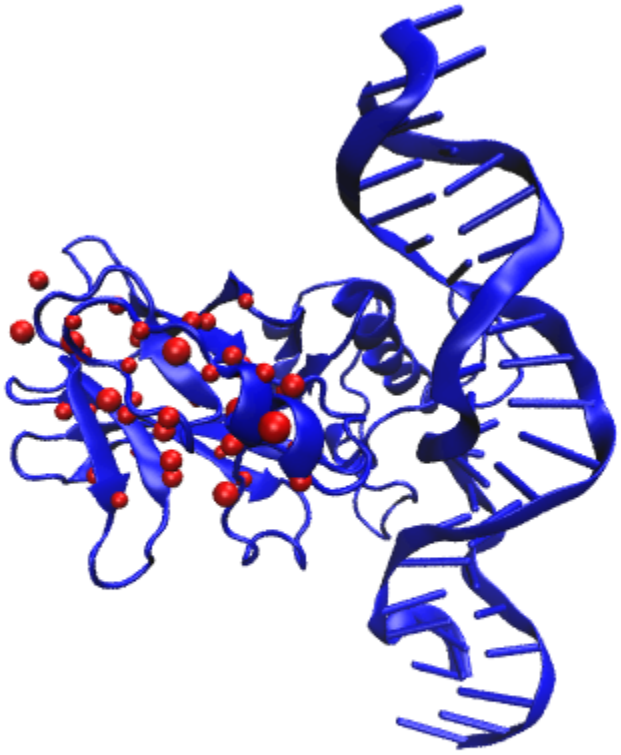


Figure 5.3.1.8 Sector residues (red) projected onto isoform 9 (blue).

The following residues were identified from isoform 10:

CY2238, ASN239, CY2242, TYR236, MET237, MET243,

LEU194, ASN235, GLY245, ARG175, SER241, MET246,

CY2176, SER240, ASN247, HE1179, HIE233, ILE195, TYR234,

HIE178, ARG248, SER183, ARG196, GLY244, VAL197, GLU198, VAL272, GLN192, HIE193, ARG174, THR231, PRO191, PRO177, GLU180, VAL274, CYS182, ALA189, GLY199, ASN200, ILE232, PRO190, VAL173, ARG273, ARG181, ASP184, LEU188

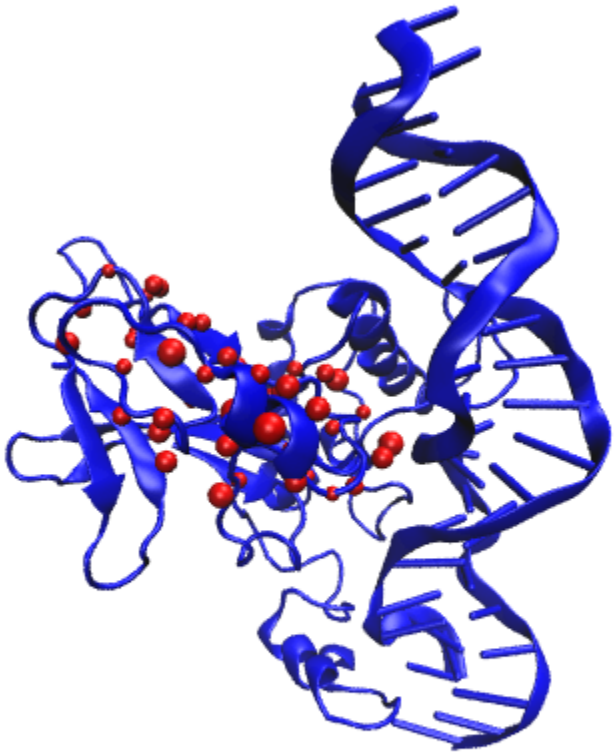


Figure 5.3.1.9 Sector residues (red) projected onto isoform 10 (blue).

The following residues were identified from isoform 11:

LEU194, HIE193, ILE195, GLN192, PRO191, PRO190, ARG175, ARG174, ALA189, HIE214, ASP207, ARG196, VAL173, SER215, MET237, ALA161, TYR205, LEU206, TYR236, GLY187, ASP208, VAL216, ILE251, GLU180, SER185, ARG213, ARG209, CY2238, PHE212, LEU188, VAL172, SER183, ILE162, ARG181, CYS182, ASN210

Figure 5.3.1.10 Sector residues (red) projected onto isoform 11 (blue).

The following residues were identified from isoform 12:

ILE251, TYR236, LEU194, VAL173, ILE195, ALA161, ARG174, VAL272, ARG175, MET237, MET246, HIE193, ILE162, CY2238, LEU252, ARG196, CY2176, PRO250, THR253, ASN235, TYR163, GLN192, VAL172, GLY245, ARG249, ARG273, HIE214, MET160, CY2242, TYR234, GLU180, PRO177, ASN239, GLY244, ASN247, GLU171, PRO191

Figure 5.3.1.11 Sector residues (red) projected onto isoform 12 (blue).

5.3.2 Key Findings

The following results were identified when comparing each of the isoform sector residues to the hydrogen bonding sites (residues 120 241, 248, 273, 276, 277, 280, 283) and the hot spots (residues 175, 248, 273, 175, 220, 282).

Isoform 2's sector included hot spot 175. It did not include any other hydrogen bonding residues. Isoform 3's sector included hot spots 273 and 282. The sector had 6 hydrogen bonding residues: 120, 241, 273, 276, 277, 280. Isoform 4's sector included hot spots 273 and 282. It also had 6 hydrogen bonding residues: 120, 273, 276, 277, 280, 283. Isoform 5's sector included hot spots 175 and 273. It also had 6 hydrogen bonding residues: 120, 241, 273, 276, 277, 280. Isoform 6's sector included hot spots 175 and 273. It had 2 hydrogen bonding residues 273 and 276. Isoform 7's sector had no hot spot or hydrogen bonding residues. Each of the following isoforms do not share any residues with the hydrogen bonding sites. Isoform 8's sector had one hot spot residue at 220 and no hydrogen bonding residues. Isoforms 9, 10, 11, and 12 sectors included one hot spot residue at 175.

CHAPTER 6

Results III: Applications to Other Biological Systems

6.1 Preliminary Findings on the Nucleosome

The nucleosome is responsible for compacting DNA into chromatin while also playing an important role in regulating accessibility to genetic materials. If one were to unfold all of the DNA inside the nucleosome, it would reach a length of two meters. All of this DNA, 145 to 147 base pairs, is wrapped around the core of the nucleosome which is made up of eight histone proteins. Adjacent nucleosomes are connected with linker histone proteins. There are 6 total analyses in this section (Figure 6.1.1). They will be denoted with n[A,B][+/-][lk, LS]. This refers to nucleosome A and/or B, + being with the linker histone and - being without the linker histone. If the dataset includes nucleosomes A and B, the third term will be lk, as opposed to LS. The different datasets are nAB-lk, nAB+lk, nA+Ls, nA-LS, nB+LS, and nB-LS.

The results of these findings aim to investigate two ideas: how MD sectors perform for larger systems and how structural dynamics vary in response to the presence and absence of a linker histone.

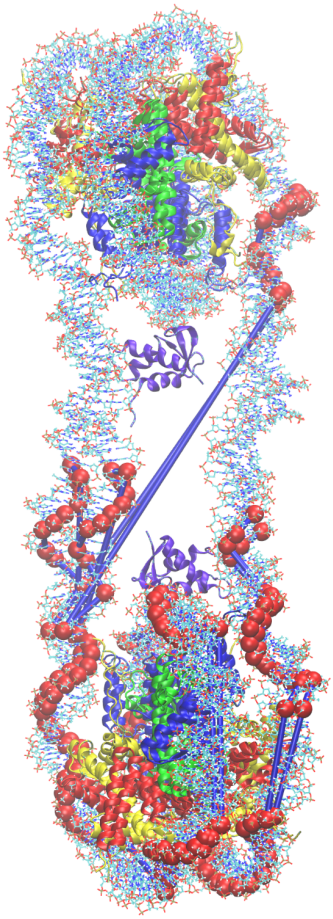


Figure 6.1.1 3-D visualization of nucleosome, including nAB-lk, nAB+lk, nA+Ls, nA-LS, nB+LS, and nB-LS

6.1.1 Results

Each of the following sectors are sorted with the highest correlated residues at the beginning and the least correlated residues at the end.

nAB+lk sector residues include:

325 324 266 321 322 328 326 323 265 327 318 329 555 320 221

224 225 319 357 557 267 228 253 358 317 250 222 330 550 220

556 249 268 297 218 553 223 217 196 554 226 252 227 331 229

315 296 354 191 558 195 219 316 294 332 314 199 293 300 298

295 292 192 214 193 353 251 197 246 200 232 359 194 190 299

301 203 366 263 216 350 260 254 215 362 198 189 333 231 356

269 230 363 184 551 213 310 547 546 271 248 549 313 244 245

183 370 312 355 247 311 302 182

nAB-lk sector residues include:

253 266 254 260 252 228 232 251 250 249 265 231 259 229 233

255 235 230 234 321 227 236 261 244 248 225 256 318 224 237

226 268 263 257 267 238 317 258 264 239 262 305 241 301 304

245 243 308 246 242 314 325 240 358 324 557 310 322 309 223

247 320 300 362 359 302 306 357 307 303 297 315 271 222 298

311 221 319 555 299 363 328 361 316 269 220 275 366 272 274

323 556 365 169 360 313 558 270 554 296 326 354 356 276 273

312 364 400 553 168 219 329 190

nA+LS sector residues include:

444 441 445 554 447 446 500 496 497 442 549 251 268 443 555

252 253 358 440 359 254 229 141 556 449 165 232 233 140 144

137 267 550 495 499 357 356 501 439 228 139 364 502 138 355

248 250 136 448 225 498 142 363 326 143 255 249 503 360 266

552 269 493 231 230 546 329 166 135 133 322 494 366 236 367

365 224 354 234 132 168 226 169 330 37 227 235 247 450 134

548 145 438 276 275 245 361 492 167 325 164 408 256 362 353

63 222 221 237 547 246 551 368 557 34 436 319 464 504 453 437

259 38 465 171 62 33 327 258 147 323 36 125 130 129 66 126 510

409 173 352 369 223 131 545 463 333 553 87 70 86 170 59 127 84

524 457 41 146 260 370 462 73 35 67 302 244

nA-LS sector residues include:

452 70 453 71 74 469 86 457 84 456 85 67 79 468 83 454 82 69

462 81 449 455 470 458 80 73 461 68 72 66 75 466 460 78 471 87

450 465 464 451 123 463 122 121 459 77 467 126 119 125 448 76

114 120 65 124 504 6 472 7 64 505 63 117 118 115 506 473 111

502 127 5 446 503 445 508 130 129 113 110 447 62 509 10 116 8

88 497 128 112 133 4 516 14 512 11 500 513 107 9 501 507 390

108 474 13 498 389 12 494 132 134 3 109 131 61 60 15 496 520

444 517 493 391 393 137 515 510 17 499 18 89 106 16 59 2 388

397 495 136 519 394 511 514 392 135 24 105 490 442 90 387

nB+LS sector residues include:

650 651 649 652 173 443 653 61 444 172 633 632 174 629 631 62

171 630 628 442 58 447 697 654 615 57 627 634 440 60 445 441

170 468 611 448 614 471 267 467 619 635 116 618 64 117 511 65

694 66 169 166 811 624 513 63 815 612 648 510 446 616 512 118

644 617 67 68 167 493 508 470 626 647 509 59 465 469 69 466

657 439 520 693 704 70 516 517 489 620 613 18 175 623 115 698

496 472 71 488 701 828 636 622 621 492 610 514 696 168 119

497 56 490 825 494 507 114 625 19 515 695 645 464 113 495 519

17 656 474 74 518 75 637 552 521 646 491 549 14 120 506 55 554

473 449 643 80 818 700 826 79 72 833 78 20 112 15 16 608 451

524 438 268 266 692 11

nB-LS sector residues include:

170 171 446 167 447 443 69 169 450 442 445 172 449 650 552

444 448 550 649 468 168 439 73 451 551 70 173 471 500 166 72

549 452 472 440 441 74 648 469 261 467 501 68 66 454 453 497

516 513 71 262 455 76 499 498 438 502 395 75 465 259 470 359

258 517 637 633 77 65 520 174 456 67 260 509 496 457 512 79

358 475 463 257 89 493 466 86 636 494 256 82 458 495 514 399

548 515 464 83 90 460 398 357 81 274 503 524 461 254 519 510

634 78 436 394 437 165 80 255 284 253 87 459 462 85 504 518

651 473 492 508 360 285 396 92 93 523 511 88 488 435 521 273

630 265 507 91 119 120 406 361 50

6.1.2 Key Findings

The nucleosome has ~1000 residues. The analyses for p53 have been within the 100 to 300 residue range. It took an average of ~0.78 seconds for each

analysis of the nucleosome. The sector analysis algorithm fared well in a larger system. Each of the datasets have differing sector residues, so the presence and absence of the linker histone does affect the most correlated residues.

CHAPTER 7

Results IV: MD Sectors Website

7.1 Improved Accessibility with Flask

To make the results of this thesis more accessible and interactive, a Flask website application was developed, a lightweight Python web framework. It consists of two HTML pages, one for input and another that displays the processed inputs. Instead of having the systems of interest. There is the ability to click the example button, which will display a scrollable example matrix.

The app.py code contains only two functions. One for rendering the input page, called *upload_form*. The other function, *example_file*, stores the file path for the example matrix and returns it. The last function, *process_files*, is the most complex one, it gathers the uploaded files, checks if there are both a .dat and .pdb file, saves the files to the web server, applies sectors to the files—with exception handling, and finally returns the result page, with the processed data.

Wescreates used to support Python apps, but on March 12th, 2025, they updated, and stopped supporting Python apps. This prompted hosting the website with pythonanywhere.com, a free online web hosting service. The website can be found at https://jphythian.pythonanywhere.com/.

Figure 7.1.1 Input page for Flask website.

Results

TYR141, CY2143, ASN144, MET142, ILE100, ASN140, VAL179, LEU99, LEU42, HE184, ARG101, ARG80, HIE83, CY2147, CY281, ALA43, ARG153, TYR139, GLU85, ASN152, PRO82, MET151, CYS87, ALA64, ARG86, HIE98, ARG79, CYS180, VAL102, GLN97, GLY150, PRO96, VAL121, GLN41, MET148, SER88, SER145, PRO95, ALA94, LYS44, VAL78, MET65, ARG154, VAL122, SER120, ARG63, THR158, SER146, ALA181, ASP89, TYR110, VAL108, GLU109, LEU93, ALA66, GLY149, GLY92, CYS40, THR45, HIE138, ARG178, GLU103, VAL123, CYS46, VAL177, HIE119, PRO155, ILE156, ILE160, VAL62, ARG107, ASN105, LEU106, VAL77, GLY104, ILE137, ASP91, PRO124, LEU111, ARG118, ASP112, PRO47, THR28, GLU76, ILE67, PHE39, ARG61, ILE159, CYS182, TYR68, CYS29, SER90, ASP113, VAL48, PRO183, LEU157, CYS125, THR136, PHE117, VAL27, SER26, MET38, THR161, HIE73, GLU176, THR75, GLU126, THR116, ARG114, MET74, THR135, LEU162, GLN49, THR30, ASN115, LYS69, THR60, ASP186, GLY59, LEU50, GLY184, LYS25, GLU163, SER165, GLN72, LEU19, PHE175, ARG185, GLY167, GLN70, PRO3, ALA24, CYS134, PHE18, SER21, LYS37, TYR31, PRO127, THR23, ASP164, VAL2, PRO58, SER166, PRO128, ARG187, ASP133, GLU129, GLY22, SER174, SER71, SER1, HIE20, GLN5, SER132, GLY17, ASN168, VAL130, THR189, LEU16, ARG188, SER4, LEU169, GLY131, ARG172, PRO57, GLU190, TRP51, ASN36, SER32, ASN173, GLY171, PRO56, LEU170, LEU35, GLU192, GLU191, PRO33, ARG15, LYS6, THR55, PHE14, ASN193, VAL52, ALA34, THR7, ARG195, LEU194, TYR12, SER54, GLY13, ASP53, SER11, GLY10, TYR8, GLN9

Figure 7.1.2 Example of output page for Flask website.

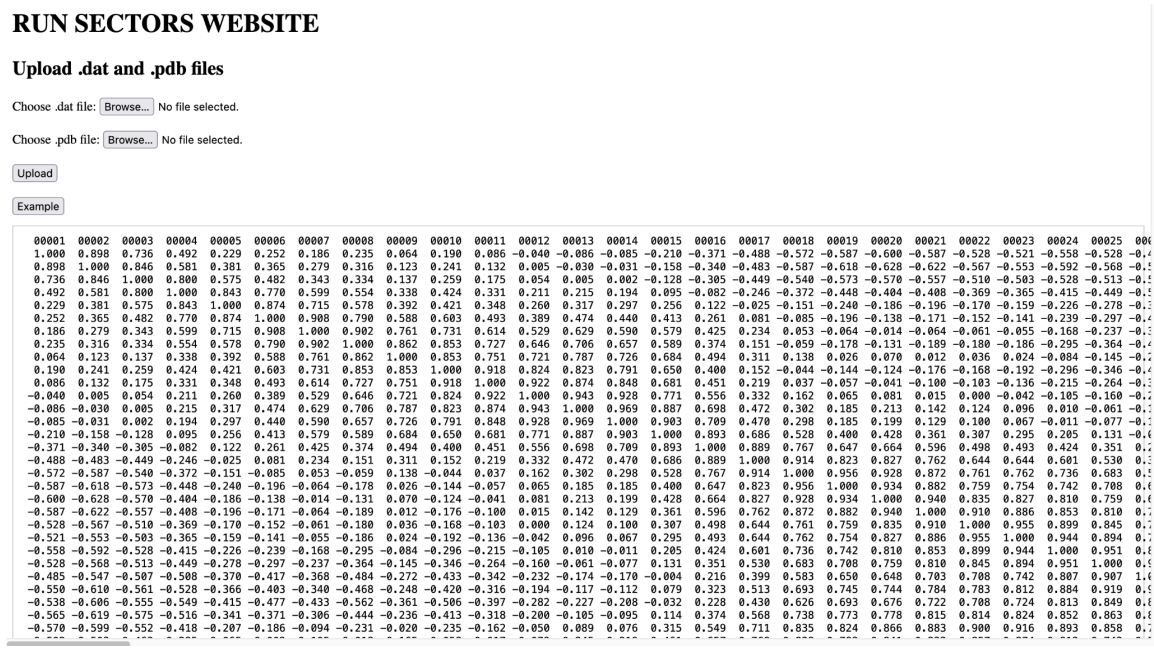


Figure 7.1.3 Example of page after example button has been clicked.

One assumption for this website is that the .dat and .pdb start at the same index. For example, when analyzing the isoforms, some of the .pdb files start at residue 40, but are indexed starting at 1. The work around for this is to add or subtract the starting residue number and then add 1 to the input .dat file.

CHAPTER 8

Conclusions and Future Directions

8.1 Key Findings

8.1.1 HPC for algorithm development

This iteration of MD sectors advances the ability to find sectors of a system of interest by using high performance computing techniques and introducing a novel refinement algorithm. These changes have provided a more efficient, comprehensive, and accessible way to identify more cohesive sector residues for a system of interest. MD sectors can also be applied to other model systems, beyond p53, as is demonstrated for the nucleosome. Although MD sectors were created with the intention to be applied to a correlation matrix, it can also be applied to any symmetric matrix, such as pairwise distance covariance matrix for other biological insights. This thesis is a study in how high performance computing plays a critical role in advancing biological sciences, enabling high throughput models suitable for the drug design pipeline.

8.1.2 Molecular level insights

Now that the points of allosteric control have been identified (section 5.1.1), the next step is testing these points and integrating them into small molecule binders provided by other Lab members. As a starting point, molecular dynamics (MD) simulations could be performed to identify plausible candidates,

which could then be validated through collaboration with an experimental lab conducting wet-lab studies

As a starting point, one could mutate residues with the expectation that the signals would be broken. The lab has previously developed an MD level screening for efficacy of pathways, which could be applied to test these prospective allosteric points (Lakhani et al.).

The results of this thesis align with the Thayer Lab's ultimate goals: to develop allosteric therapeutics that can hopefully gain approval through clinical trials, enabling human use, and safely treat currently undruggable diseases such as cancer and HIV, in safe dosages — by opening a major bottleneck in the Thayer Lab's allosteric drug design pipeline with this implementation of MD sectors.

Bibliography

Armour-Garb, Isabel, et al. “Variable Regions of P53 Isoforms Allosterically Hard Code DNA Interaction.” *The Journal of Physical Chemistry. B*, vol. 126, no. 42, Oct. 2022, pp. 8495–507. *PubMed*, https://doi.org/10.1021/acs.jpcb.2c06229.

Berman, Helen M., et al. “The Protein Data Bank.” *Nucleic Acids Research*, vol. 28, no. 1, Jan. 2000, pp. 235–42. *Silverchair*, https://doi.org/10.1093/nar/28.1.235.

Borrero, Liz J. Hernández, and Wafik S. El-Deiry. “Tumor Suppressor P53: Biology, Signaling Pathways, and Therapeutic Targeting.” *Biochimica et Biophysica Acta. Reviews on Cancer*, vol. 1876, no. 1, Aug. 2021, p. 188556. *PubMed Central*, https://doi.org/10.1016/j.bbcan.2021.188556.

Cho, Y., et al. “Crystal Structure of a P53 Tumor Suppressor-DNA Complex: Understanding Tumorigenic Mutations.” *Science (New York, N.Y.)*, vol. 265, no. 5170, July 1994, pp. 346–55. *PubMed*, https://doi.org/10.1126/science.8023157.

DeDecker, Brian S. “Allosteric Drugs: Thinking Outside the Active-Site Box.” *Chemistry & Biology*, vol. 7, no. 5, May 2000, pp. R103–07. *ScienceDirect*, https://doi.org/10.1016/S1074-5521(00)00115-0.

Fabry, Jonathan D., and Kelly M. Thayer. “Network Analysis of Molecular Dynamics Sectors in the P53 Protein.” *ACS Omega*, vol. 8, no. 1, Jan. 2023, pp. 571–87. *ACS Publications*, https://doi.org/10.1021/acsomega.2c05635.

Gunning, Peter W., and Edna C. Hardeman. “Fundamental Differences.” *eLife*, vol. 7, Feb. 2018, p. e34477. *eLife*, https://doi.org/10.7554/eLife.34477.

Han, In Sub M., et al. “Insights into Rational Design of a New Class of Allosteric Effectors with Molecular Dynamics Markov State Models and Network Theory.” *ACS Omega*, vol. 7, no. 3, Jan. 2022, pp. 2831–41. *ACS Publications*, https://doi.org/10.1021/acsomega.1c05624.

Han, In Sub M., and Kelly M. Thayer. “Reconnaissance of Allostery via the Restoration of Native P53 DNA-Binding Domain Dynamics in Y220C Mutant P53 Tumor Suppressor Protein.” *ACS Omega*, vol. 9, no. 18, Apr. 2024, pp. 19837–47. *PubMed Central*, https://doi.org/10.1021/acsomega.3c08509.

Hertig, Samuel, et al. “Revealing Atomic-Level Mechanisms of Protein Allostery with Molecular

Dynamics Simulations." *PLoS Computational Biology*, vol. 12, no. 6, June 2016, p. e1004746. *PubMed*, https://doi.org/10.1371/journal.pcbi.1004746.

Hollingsworth, Scott A., and Ron O. Dror. "Molecular Dynamics Simulation for All." *Neuron*, vol. 99, no. 6, Sept. 2018, pp. 1129–43. *ScienceDirect*, https://doi.org/10.1016/j.neuron.2018.08.011.

Lakhani, Bharat, et al. "Evolutionary Covariance Combined with Molecular Dynamics Predicts a Framework for Allostery in the MutS DNA Mismatch Repair Protein." *The Journal of Physical Chemistry B*, vol. 121, no. 9, Mar. 2017, pp. 2049–61. *ACS Publications*, https://doi.org/10.1021/acs.jpcb.6b11976.

Lane, D. P. "P53, Guardian of the Genome." *Nature*, vol. 358, no. 6381, July 1992, pp. 15–16. *www.nature.com*, https://doi.org/10.1038/358015a0.

Lindorff-Larsen, Kresten, et al. "Improved Side-Chain Torsion Potentials for the Amber ff99SB Protein Force Field." *Proteins: Structure, Function, and Bioinformatics*, vol. 78, no. 8, 2010, pp. 1950–58. *Wiley Online Library*, https://doi.org/10.1002/prot.22711.

Liu, Jin, and Ruth Nussinov. "Allostery: An Overview of Its History, Concepts, Methods, and Applications." *PLoS Computational Biology*, vol. 12, no. 6, June 2016, p. e1004966. *PubMed Central*, https://doi.org/10.1371/journal.pcbi.1004966.

Mohs, Richard C., and Nigel H. Greig. "Drug Discovery and Development: Role of Basic Biological Research." *Alzheimer's & Dementia : Translational Research & Clinical Interventions*, vol. 3, no. 4, Nov. 2017, pp. 651–57. *PubMed Central*, https://doi.org/10.1016/j.trci.2017.10.005.

Rivoire, Olivier, et al. "Evolution-Based Functional Decomposition of Proteins." *PLoS Computational Biology*, vol. 12, no. 6, June 2016, p. e1004817. *PubMed*, https://doi.org/10.1371/journal.pcbi.1004817.

Süel, Gürol M., et al. "Evolutionarily Conserved Networks of Residues Mediate Allosteric Communication in Proteins." *Nature Structural Biology*, vol. 10, no. 1, Jan. 2003, pp. 59–69. *PubMed*, https://doi.org/10.1038/nsb881.

Thayer, Kelly M., et al. "Navigating the Complexity of P53-DNA Binding: Implications for Cancer Therapy." *Biophysical Reviews*, vol. 16, no. 4, Aug. 2024, pp. 479–96. *PubMed*, https://doi.org/10.1007/s12551-024-01207-4.

Vogelstein, Bert, et al. "Surfing the P53 Network." *Nature*, vol. 408, no. 6810, Nov. 2000, pp.

307–10. *www.nature.com*, https://doi.org/10.1038/35042675.

Walker. *Amber (PMEMD) GPU Support - Recommended Hardware*. https://ambermd.org/gpus/recommended_hardware.htm. Accessed 17 Apr. 2025.

Wodak, Shoshana J., et al. “Allostery in Its Many Disguises: From Theory to Applications.” *Structure (London, England : 1993)*, vol. 27, no. 4, Apr. 2019, pp. 566–78. *PubMed Central*, https://doi.org/10.1016/j.str.2019.01.003.

Appendix A: run_sectors_numpy.py

```
"""
Available at:
https://github.com/joshphythian/Sector_Indentification_and_Analysis

Input:
a correlation matrix (or any symmetric matrix)

Output:
a heatmap of the original matrix
a heatmap of the rearranged matrix
a heatmap of the swapped matrix
sectors of the matrix
csv files of the matricies (optional)
"""

import seaborn as sns
import matplotlib.pyplot as plt
import time
import csv
import os
import numpy as np
import pandas as pd
from typing import Tuple

def time_func(func):
      def wrap_func(*args, **kwargs):
      t1 = time.time()
      result = func(*args, **kwargs)
      t2 = time.time()
      delta = t2 - t1
      print(f'Function {func.__name__!r} executed in
{(delta):.4f}s')

      output_dir = "/Users/joshp/Desktop/thesis/data"
      if not os.path.exists(output_dir):
            os.makedirs(output_dir)

      with open(os.path.join(output_dir,
'function_times_for_thesis_numpy.csv'), 'a', newline='') as
csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow([func.__name__, delta])

      return result
      return wrap_func
```

```
def output_csv(dataset: np.ndarray, labels: np.ndarray, directory:
str, filename: str):
      """
      Parameters:
      dataset : np.ndarray
      labels : np.ndarray
      directory : String
      filename : String

      Output:
      Saves the dataset to a CSV file with labels.
      """
      file_path = os.path.join(directory, filename)

      # Prepare the header
      header = '\t' + '\t'.join(labels)
      # Prepare the data with row labels
      data_with_labels = np.column_stack((labels[:, np.newaxis],
dataset))

      # Save to file
      np.savetxt(file_path, data_with_labels, fmt='%s',
delimiter='\t', header=header, comments='')
      print(f'{filename} saved to {file_path}')

def load_dat_file(filename: str) -> Tuple[np.ndarray, np.ndarray]:
      """
      Parameters:
      filename : str

      Output:
      Returns the NumPy array from the DAT file along with labels.
      """
      # Read the data using pandas and then convert to NumPy arrays
      # works for .dat files
      df = pd.read_csv(filename, delimiter='\s+')

      # works for csv files
      # df = pd.read_csv(filename, index_col=0)

      data_array = df.values               # 2D array for data
      column_labels = df.columns.to_numpy() # 1D array for column
labels
      # row_labels = column_labels         # 1D array for row
labels (assuming square matrix)

      return data_array, column_labels

def zero_bonded_residues_np(dataset: np.ndarray) -> np.ndarray:
```

```
        """
        Parameters:
        dataset : np.ndarray

        Output:
        Returns a NumPy array where the diagnol has been zeroed out.
        """
        data_modified = dataset.copy()

        # Set the main diagonal to 0
        np.fill_diagonal(data_modified, 0)

        return data_modified

def get_column_sums_and_sort_np(dataset: np.ndarray, column_labels:
np.ndarray) -> np.ndarray:
        """
        Parameters:
        dataset : np.ndarray
        column_labels : np.ndarray

        Output:
        sorted_column_sums : np.ndarray
        sorted_labels : np.ndarray

        This function takes a dataset and returns the sum of each
column, and then sorts the columns in descending order.
        """
        # Sum columns while keeping rows
        column_sums = dataset.sum(axis=0)

        # Get sort order (indices) that would sort the sums in
descending order
        sorted_indices = np.argsort(-column_sums)  # Negative for
descending order

        # Sort the sums and labels accordingly
        # sorted_column_sums = column_sums[sorted_indices]
        sorted_labels = column_labels[sorted_indices]

        return sorted_labels

# @time_func
def rearrange_matrix_np(dataset: np.ndarray, sorted_indices:
np.ndarray) -> np.ndarray:
        """
        Parameters:
        dataset : np.ndarray
        sorted_indices : np.ndarray
```

```
      Output:
      rearranged_dataset : np.ndarray

      Rearranges the dataset based on the sorted_indices.
      """
      rearranged_dataset = dataset[np.ix_(sorted_indices,
sorted_indices)]
      return rearranged_dataset

def get_sector_np(dataset: np.ndarray, sector_size: int) ->
np.ndarray:
      """
      Parameters:
      dataset : np.ndarray
      sector_size : int

      Output:
      sector : np.ndarray

      Returns the top-left sector of the dataset.
      """
      return dataset[:sector_size, :sector_size]

def get_sector_outside_np(dataset: np.ndarray, sector_size: int) ->
np.ndarray:
      """
      Parameters:
      dataset : np.ndarray
      sector_size : int

      Output:
      sector : np.ndarray

      Returns everything but the top-left nxn sector of the
dataset.
      """
      return dataset[sector_size:, sector_size:]

def get_sector_values_np(dataset: np.ndarray) -> float:
      """
      Parameters:
      dataset : np.ndarray

      Output:
      total_sector_value : float

      Returns the sum of all values in the dataset.
      """
```

```
      total_sector_value = dataset.sum()
      return total_sector_value

# @time_func
def switch_values_np(dataset: np.ndarray,
                     outside_indices: np.ndarray,
                     labels: np.ndarray,
                     dataset_zeroed_lookup: np.ndarray,
                     output_dir: str) -> Tuple[np.ndarray,
np.ndarray]:
      """
      Parameters:
      dataset : np.ndarray (rearranged dataset)
      outside_indices : np.ndarray (indices of columns outside the
sector in rearranged dataset)
      labels : np.ndarray (rearranged labels)
      dataset_zeroed_lookup : np.ndarray (zeroed original dataset)
      output_dir : str

      Output:
      modified_dataset : np.ndarray
      modified_labels : np.ndarray

      The function attempts to swap columns from outside the sector
into the sector to increase the total sector value.
      """
      output_dir = output_dir + '/swaps'
      i = 1  # iterator for names of csv files

      copy_of_dataset = dataset.copy()
      copy_of_labels = labels.copy()

      n = dataset.shape[1]
      sector_size = int(n / 5)
      curr = get_sector_values_np(get_sector_np(dataset,
sector_size))
      print("here is the intial sector value", curr)
      # print("Here is intial get_sector_np\n",
get_sector_np(dataset, sector_size))

      # Indices in the rearranged dataset
      for outside_idx in outside_indices:
      swapped_col_idx = sector_size - 1  # Index of last column in
sector

      # Swap columns and rows in the dataset
      new_indices = np.arange(n)
      new_indices[swapped_col_idx], new_indices[outside_idx] =
new_indices[outside_idx], new_indices[swapped_col_idx]
```

```
      sector_indicies = new_indices[:sector_size]
      sector_sum = sector_indicies.sum()
      # Rearrange the dataset and labels accordingly
      rearranged_labels = labels[new_indices]

      # create sector with ix_ function
      new_way_of_sector = dataset[np.ix_(sector_indicies,
sector_indicies)]
      temp = get_sector_values_np(new_way_of_sector)

      ##### I leave these here because these can be useful for
testing and seeing
      ##### the changes for made for each swap
      # print(f"Swap {i}: Swapping '{labels[outside_idx]}' into
sector.")
      # print(f"New sector value: {temp:.3f}")

      # output CSV for each swap (optional)
      # output_csv(rearranged_dataset, rearranged_labels,
output_dir,
f'swap{i}_{copy_of_labels[swapped_col_idx]}_for_{labels[outside_idx
]}.csv')
      i += 1

      if temp > curr:
            print(f"New sector value {temp} is greater than current
{curr}. Applying swap.")
            curr = temp
            copy_of_dataset = dataset[np.ix_(new_indices,
new_indices)]
            copy_of_labels = rearranged_labels
      # print()

      return copy_of_dataset, copy_of_labels

def plot_heatmap_np(matrix: np.ndarray, labels: np.ndarray,
                  output_dir: str, filename: str, title: str =
"Heatmap") -> None:
      """
      Plots and saves a heatmap of the given matrix with the
provided row and column labels.
      """
      # Convert the matrix to a pandas DataFrame for easier
labeling
      df = pd.DataFrame(matrix, index=labels, columns=labels)

      plt.figure(figsize=(10, 8))
      # Create the heatmap
      ax = sns.heatmap(df, cmap='viridis',
```

```
            xticklabels=True, yticklabels=True,
            square=True, cbar_kws={"shrink": .75})

      ax.set_xticklabels(ax.get_xticklabels(), fontsize=4,
rotation=90)  # rotated and smaller x-axis labels
      ax.set_yticklabels(ax.get_yticklabels(), fontsize=4)  #
smaller y-axis labels

      plt.title(title, fontsize=10)
      plt.xlabel("Columns", fontsize=8)
      plt.ylabel("Rows", fontsize=8)
      plt.tight_layout()

      # save figure
      output_path = os.path.join(output_dir, filename)
      plt.savefig(output_path, dpi=300)
      plt.close()

      print(f"Heatmap saved to: {output_path}")

@time_func
def run_everything_np(base_dir_dataset: str, output_directory: str)
-> np.ndarray:
      # load data
      dataset, column_labels = load_dat_file(base_dir_dataset)
      # Ensure output directory exists
      if not os.path.exists(output_directory):
      os.makedirs(output_directory)


      # Get rewritten matrix that replaces 1's with 0's
      dataset_zeroed_lookup = zero_bonded_residues_np(dataset)

      # # gen heatmap of orign data

      # # Re-zero the dataset so all the numbers touching diagonal
are zero as well
      # # The re-zeroing does not happen anymore, as it screws up
the sector values.
      # datasetzeroed =
zero_bonded_residues_np(dataset_zeroed_lookup)

      # Now get ranking of columns greatest to least based on sum
of columns
      sorted_labels =
get_column_sums_and_sort_np(dataset_zeroed_lookup, column_labels)
      # Get the sorted indices
      sorted_indices =
np.argsort(-dataset_zeroed_lookup.sum(axis=0))
```

```
      # Rearrange the dataset based on sorted_columns, making sure
it stays symmetric
      rearranged_dataset =
rearrange_matrix_np(dataset_zeroed_lookup, sorted_indices)
      rearranged_labels = column_labels[sorted_indices]


      # Get sector size
      n = rearranged_dataset.shape[0]
      sector_size = int(n / 5)

      # Get indices of columns in the sector and outside
      sector_indices = np.arange(sector_size)
      outside_indices = np.arange(sector_size, n)

      # Do the switch
      swapped_matrix, swapped_labels =
switch_values_np(rearranged_dataset, outside_indices,
rearranged_labels, dataset_zeroed_lookup, output_directory)

      # uncomment these if you want to output
      # the csvs and heatmaps
      """
      # output the csvs
      output_csv(dataset, column_labels, output_directory,
'original_matrix.csv')
      output_csv(dataset_zeroed_lookup, column_labels,
output_directory, 'zeroed_Y220C_matrix_.csv')
      output_csv(rearranged_dataset, rearranged_labels,
output_directory, 'rearranged_Y220C_matrix_.csv')
      output_csv(swapped_matrix, swapped_labels, output_directory,
'swapped_Y220C_matrix_.csv')


      # plot heatmaps
      plot_heatmap_np(dataset_zeroed_lookup, column_labels,
output_directory, 'orig_matrix_heatmap.png')
      plot_heatmap_np(rearranged_dataset, rearranged_labels,
output_directory, 'rearranged_matrix_heatmap.png')
      plot_heatmap_np(swapped_matrix, swapped_labels,
output_directory, 'swapped_matrix_heatmap.png')
      """
      return swapped_labels[:sector_size]

"""
The next segment of code corresponds to correcting the names of the
residues within the sector.
"""
```

```
def GetResidueList(fileName: str) -> list[str]:
      """Extracts the amino acid chain from a PDB file

      Args:
      fileName: The PDB file to extract the sequence of

      Returns:
      An array containing the three-letter amino acid sequence
      """
      #Stores the chain of amino acids
      residues = []

      #Used to keep track of the index of the current residue
number so that
      #duplicates are not accidentally added
      currResNum = 0

      with open(fileName) as file:
      for line in file:
            if line[:4] == "ATOM": #if this line corresponds to an
atom in a residue, excludes TER, HETATM, etc
            thisLinesResNum = line.split()[4]
            if line.split()[4] != currResNum:
                  currResNum = thisLinesResNum
                  residues.append(line.split()[3])
      return residues

def add_number_to_list_with_code(input_list: list,
pdb_residue_list: list, number: int):
      """
      This function takes the pdb_list and takes the sector list,
and adds the specified number to
      the residue in the sector, then concatenates that onto the
corresponding residue.
      """
      modified_list = []
      for item in input_list:
      modified_list.append(pdb_residue_list[int(item) - 1] +
str(int(item) + number))

      return modified_list

def turn_residues_to_correct(output_directory: str, sector_labels :
np.ndarray, pdb_file_path: str) -> str:
      """
      This function takes the output directory to save the correct
sector residue list to.
```

```
      Without this function, if the pdb file does not start at the
same index of the .dat file then it will
      output the wrong residues. This function takes the pdb file
and the sector labels, and adds the correct number to
      sector labels.

      It outputs the correct residue list to a text file in the
output directory and as a string to use for copy and pasting.
      """
      # first get rid of first char for each residue in
residue_labels
      # one of the previous implementations added a c to the
beginning
      # of each residue number
      if sector_labels[0][0] == 'c':
      sector_labels = np.array([s[1:] for s in sector_labels])

      pdb_residue_list = GetResidueList(pdb_file_path)
      correct_residue_list =
add_number_to_list_with_code(sector_labels.tolist(),
pdb_residue_list, 95)
      residue_list_file_path = os.path.join(output_directory,
"sector_residues.txt")
      with open(residue_list_file_path, 'w') as file:
      for item in correct_residue_list:
            file.write(f"{item}\n")

      residue_list_str_form = ""
      for item in correct_residue_list:
      residue_list_str_form += f"{item}, "
      residue_list_str_form = residue_list_str_form[:-2]
      print(f"Correct residue list:\n {residue_list_str_form}")
      print(f'Correct residue list saved to
{residue_list_file_path}')

      return residue_list_str_form

def main():

      base_directory = 'path to .dat file you want to have
analyzed'
      ##### examples
      base_directory =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Inputs/Y220Cp5
311-30-23CorrelationMatrixHeaderModified.dat'
      # base_directory =
"/Users/joshp/Desktop/Summer24Research/Optimization1/Inputs/WildTyp
ep5311-30-23CorrelationMatrixHeaderModifdied.dat"
```

```
      # base_directory =
"/Users/joshp/Desktop/Summer24Research/Optimization1/misc_data/NEWP
K11000Cp5311-30-23CorrelationmatrixHeaderMODIFIED.csv"

      # output_directory = 'path to folder where you want outputs
to go'
      ##### example
      # output_directory =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Outputs/numpy'
      output_directory =
"/Users/joshp/Desktop/thesis/Figures/Y220C"

      # get the sector residues in the form of np.ndarray
      sector_labels = run_everything_np(base_directory,
output_directory)

      pdb_file_path = 'path to .pdb file you want to have analyzed'
      ##### examples
      pdb_file_path =
"/Users/joshp/Desktop/Summer24Research/Optimization1/dataframes/p53
_DBD_Y220C_ff14SB.pdb.txt"
      # pdb_file_path =
"/Users/joshp/Desktop/Summer24Research/Optimization1/dataframes/p53
_DBD_WT_ff14SB.pdb.txt"
      # pdb_file_path =
"/Users/joshp/Desktop/Summer24Research/Optimization1/dataframes/p53
_DBD_PK11000_ff14SB.pdb.txt"

      modified_sector_labels =
turn_residues_to_correct(output_directory, sector_labels,
pdb_file_path)
      return

if __name__ == "__main__":
      main()
```

Appendix B: run_sectors_pandas.py

```
# -*- coding: utf-8 -*-
"""

"""

import pandas as pd
import seaborn
import matplotlib.pyplot as plt
import time
import csv
import os
import numpy as np

def time_func(func):
      def wrap_func(*args, **kwargs):
      t1 = time.time()
      result = func(*args, **kwargs)
      t2 = time.time()
      delta = t2 - t1
      print(f'Function {func.__name__!r} executed in
{(delta):.4f}s')

      output_dir =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Outputs/Y220C_
isoforms_w_new_headers'
      if not os.path.exists(output_dir):
            os.makedirs(output_dir)

      with open(os.path.join(output_dir, 'function_times.csv'),
'a', newline='') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow([func.__name__, delta])

      return result
      return wrap_func


def output_csv(dataset : pd.DataFrame, directory: str, filename :
str):
      """
      Parameters:
      dataset : DataFrame
      directory : String
      filename : String

      Output:
```

```
      nothing just prints the file name and the path it was saved
to
      also saves dataset on computer
      """
      file_path = os.path.join(directory, filename)
      dataset.to_csv(file_path, index=True)

      print(f'{filename} saved to {file_path}')

def zero_bonded_residues(dataset : pd.DataFrame, notInitial : bool)
-> pd.DataFrame:
      """
      This function takes a dataset from the pd.read_csv, and finds
the ones in the diaganol
      and converts it into a zero, along with any residue above or
below it. This is because
      the residues will have an 100% correlation with itself.

      Parameters:
      dataset : DataFrame
      The Input is a Pandas DataFrame, and relies on a loaded in
CSV file from Excel.

      Output:
      An n x n matrix where the 1.0's on the diagnol are replaced
with 0's.
      If notInitial = True then
      the values touching the diagnols are replaced with 0's
      Else if notInitial = False then nothing changes
      This is so it doesn't screw up the lookup table which is
calculated based off the zeroed dataset

      """

      data_modified = dataset.copy()

      # Iterate through each column
      for col in data_modified.columns:

      # Identify rows with 1
      ones_indices = data_modified[col] == 1

      # Set the diag = 0
      data_modified.loc[ones_indices, col] = 0

      # Use shift to handle previous and next rows if notInitial is
True
      if notInitial:
```

```
            data_modified.loc[ones_indices.shift(1,
fill_value=False), col] = 0
            data_modified.loc[ones_indices.shift(-1,
fill_value=False), col] = 0

      return data_modified

def get_column_sums_and_sort(dataset : pd.DataFrame) -> pd.Series:
      """
      Parameters:
      dataset : DataFrame

      Output:
      sorted_column_sums : Series

      This function takes a dataset and returns the sum of each
column, and then sorts the columns in a series
      """

      # get sums of columns
      column_sums = dataset.sum()

      # sort the columns
      sorted_columns_with_sums =
column_sums.sort_values(ascending=False)

      return sorted_columns_with_sums

def rearrange_matrix(original_dataset : pd.DataFrame,
column_order_index : pd.Index) -> pd.DataFrame:
      """
      Parameters:
      original_dataset : DataFrame
      column_order : Series

      Original Dataset replaces the lookup dictionary

      Output:
      rearranged_dataset : DataFrame

      Steps:
      Apply column ranking to dataset
      Switch column values based on ranking, while also switching
the corresponding row
      Make sure matrix is symmetric
      """

      # Create an empty DataFrame w labels from sorted_columns
```

```
      dataset_outline = pd.DataFrame(index=column_order_index,
columns=column_order_index)

      # Fill the outline with values from the original dataset
      for col in dataset_outline:
      for row in dataset_outline:
            dataset_outline.at[col, row] = original_dataset.at[col,
row]

      return dataset_outline

def fill_matrix(original_dataset : pd.DataFrame, dataset_outline :
pd.DataFrame) -> pd.DataFrame:
      """
      Parameters:
      original_dataset : DataFrame

      Output:
      rearranged_dataset : DataFrame

      """
      for col in dataset_outline:
      for row in dataset_outline:
            dataset_outline.at[col, row] = original_dataset.at[col,
row]

      return dataset_outline

def get_sector(dataset : pd.DataFrame, sector_size: int) ->
pd.DataFrame:
      """
      Parameters:
      dataset : DataFrame
      sector_size : Int

      Output:
      sector : DataFrame

      Takes a dataset and a sector size and returns first n x n
sector as a DataFrame
      """
      copy_of_dataset = dataset.copy()
      copy_of_dataset = copy_of_dataset.iloc[:sector_size,
:sector_size]
      return copy_of_dataset

def get_sector_outside(dataset : pd.DataFrame, sector_size: int) ->
pd.DataFrame:
      """
```

```
        Parameters:
        dataset : DataFrame
        sector_size : Int

        Output:
        sector : DataFrame

        Takes a dataset and a sector size and returns everything but
the n x n sector as a DataFrame
        """

        copy_of_dataset = dataset.copy()
        copy_of_dataset = copy_of_dataset.iloc[sector_size:,
sector_size:]
        return copy_of_dataset

def get_sector_values(dataset : pd.DataFrame) -> pd.Series:
        """
        Parameters:
        dataset : DataFrame

        Output:
        sector_values : Series

        Takes a dataset and a sector size and returns the values of
the sectors
        as a series

        """
        # get the sum of the columns this produces a series with
indexs corresponding to the sum of the columns
        sector_values = dataset.sum()

        # get the sum of the sum of the columns
        total_sector_value = sector_values.sum()

        return total_sector_value

@time_func
def switch_values(dataset : pd.DataFrame, outside_column_values :
pd.Series, dataset_zeroed_lookup : pd.DataFrame) -> pd.DataFrame:
        """
        Now we have to switch the sector values to see if other
columns have more cohesion than the current ones
        Then we will apply rearrange_matrix to it again
        We will do this until we have checked every single
combination

        Parameters:
```

```
        dataset : DataFrame
        outside_column_values : Series
        original_dataset : DataFrame

        Output:
        modified_dataset : DataFrame
        The dataset with the highest total sector value.
        """
        i = 1 # iterator for names of csv files

        copy_of_dataset = dataset.copy()

        # get sector size, then the sector, then the
total_sector_value, which we will refer to as curr
        sector_size = int(dataset.shape[1] / 5)
        sector = get_sector(dataset, sector_size)
        curr = get_sector_values(sector)

        index_of_col = sector_size
        for col in outside_column_values:

        labels = copy_of_dataset.columns
        new_labels = labels.tolist()

        # get column that will be swapped with col
        swapped_col = new_labels[sector_size - 1]

        # replace the last col in sector with new col
        new_labels[(sector_size - 1)] = col

        # replace where the col was with the swapped_col
        new_labels[index_of_col] = swapped_col
        index_of_col+=1

        # turn into index so it can be used in rearrange function
        new_index = pd.Index(new_labels)

        # get the rearranged dataset
        rearranged_dataset = rearrange_matrix(dataset_zeroed_lookup,
new_index)

        # get total sector val of rearranged_dataset
        new_sector = get_sector(rearranged_dataset, sector_size)
        temp = get_sector_values(new_sector)

        # if the new value is greater than the current ones apply the
changes
        print("here is temp for col ", col, " val:", temp)
```

```
      # output_csv(rearranged_dataset, output_dir,
f'swap{i}_{swapped_col}_for_{col}.csv')
      i += 1

      if temp > curr:
            print(f"Here is the new sector value: {temp}, Here is
the old one {curr}")
            print("Swapped", swapped_col, "with", col)
            curr = temp
            copy_of_dataset = rearranged_dataset
      print()

      return copy_of_dataset

@time_func
def run_everything(base_dir_dataset : str, output_directory : str):
      """
      Parameters:
      base_dire_dataset : str is the file path to the .dat file,
      which can be a symmetric matrix. In this case it should be a
correlation matrix
      output_directory: str is the file path to the directory
      where the output files will be saved
      """

      dataset = pd.read_csv(base_dir_dataset, delimiter='\s+')

      dataset.index = dataset.columns

      # Get rewritten matrix that replaces 1's w 0's
      notInitial = False
      dataset_zeroed_lookup = zero_bonded_residues(dataset,
notInitial)

      # rezero the dataset so all the numbers touching diagnol are
zero as well
      notInitial = True
      datasetzeroed = zero_bonded_residues(dataset_zeroed_lookup,
notInitial)

      # Now get ranking of columns greatest to least based on sum
of columns
      sorted_columns = get_column_sums_and_sort(datasetzeroed)

      # Turn it into an index
      sorted_columnns_index = sorted_columns.index

      # Rearrange the dataset based on sorted_columns, making sure
it stays symmetric
```

```
      rearranged_dataset = rearrange_matrix(datasetzeroed,
sorted_columnns_index)

      # Get 20% of the length
      sector_size = int(rearranged_dataset.shape[1] / 5)

      # Chop out n x n of the dataset
      sector = get_sector(rearranged_dataset, sector_size)

      # get outside sector
      outside_sector = get_sector_outside(rearranged_dataset,
sector_size)

      # do the switch
      swapped_matrix = switch_values(rearranged_dataset,
outside_sector, dataset_zeroed_lookup)

      #output the matricies
      output_csv(dataset, output_directory,
f'(name_of_dataset)_orig_matrix_.csv')
      output_csv(dataset_zeroed_lookup, output_directory,
f'(name_of_dataset)_zeroed_matrix_.csv')
      output_csv(rearranged_dataset, output_directory,
f'(name_of_dataset)_rearranged_matrix_.csv')
      output_csv(swapped_matrix, output_directory,
f'(name_of_dataset)_swapped_matrix_.csv')

def main():
      ##### Example
      base_directory =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Inputs/Y220Cp5
311-30-23CorrelationMatrixHeaderModified.dat'
      # output_directory =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Outputs/test_m
atrix'

      #### Example
      output_directory =
'/Users/joshp/Desktop/Summer24Research/Optimization1/Outputs/presen
tation'
      run_everything(base_directory, output_directory)



if __name__ == "__main__":
      main()
```

Appendix C: get_correlation_matrix.sh

```
#!/bin/bash

export PATH=/share/apps/CENTOS7/gcc/6.5.0/bin:$PATH
export
LD_LIBRARY_PATH=/share/apps/CENTOS7/gcc/6.5.0/lib64:$LD_LIBRARY_PAT
H

### AMBER22 ###
source /share/apps/CENTOS7/amber/amber22/amber.sh

# Enable strict error handling
set -euo pipefail

# Usage message
usage() {
      echo "Usage: $0 [-p] <myfile.prmtop> <myfile.mdcrd> <Lower
Bound> <Upper Bound> <output_name> <temp_input_file>"
      echo "Options:"
      echo "  -p   Use parallel execution with cpptraj.MPI"
      exit 1
}

# Check for the parallel option
USE_MPI=false
if [ "$1" == "-p" ]; then
      USE_MPI=true
      shift
fi

# Check if the correct number of arguments is provided
if [ $# -ne 6 ]; then
      usage
fi

# Assign variables
parmfile=$1
mdcrdfile=$2
lower_bound=$3
upper_bound=$4
output_name=$5
temp_input_file=$6

# Check if input files exist
if [ ! -f "$parmfile" ]; then
      echo "Error: Parameter file '$parmfile' not found."
      exit 1
```

```
fi

if [ ! -f "$mdcrdfile" ]; then
      echo "Error: Trajectory file '$mdcrdfile' not found."
      exit 1
fi

# Determine whether to use cpptraj or cpptraj.MPI
CPPTRAJ_CMD="cpptraj"
if $USE_MPI && command -v cpptraj.MPI &> /dev/null; then
      CPPTRAJ_CMD="cpptraj.MPI"
elif $USE_MPI; then
      echo "Warning: cpptraj.MPI not found. Falling back to
single-threaded cpptraj."
fi

# Create cpptraj input file
CPPTRAJ_INPUT="${temp_input_file}_cpptraj.in"
cat <<EOF > "$CPPTRAJ_INPUT"
# Load topology file
parm $parmfile

# Read trajectory file
trajin $mdcrdfile

# Ensure molecules stay within the box
autoimage

# Align frames using C-alpha atoms
rms first .CA

# Compute cross-correlation matrix for selected frames
matrix :$lower_bound-$upper_bound@CA :$lower_bound-$upper_bound@CA
correl out tempmat.dat byres

# Run
go
EOF

# Run cpptraj and check for errors
echo "Running $CPPTRAJ_CMD..."
if ! $CPPTRAJ_CMD -i "$CPPTRAJ_INPUT"; then
      echo "Error: cpptraj failed."
      exit 1
fi

# Combine output files
if [ -f tempmat.dat ]; then
      cat tempmat.dat > "$output_name"
```

```
else
      echo "Error: tempmat.dat not found. Something went wrong with
cpptraj."
      exit 1
fi

# Clean up temporary files
rm -f "$CPPTRAJ_INPUT" tempmat.dat

# Print completion message
echo "CC_MD_SECTORS.sh is complete. Your output file is
'$output_name'."
```

Appendix D: app.py

```
from flask import Flask, request, render_template, url_for,
send_file
import os
import apply_sectors_numpy as a_s_n

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = './uploads'

@app.route('/')
def upload_form():
      return render_template('upload.html')

@app.route('/example')
def example_file():
      # path to example .dat file
      example_file_path =
'Y220Cp5311-30-23CorrelationMatrixHeaderModified_cleaned.dat'
      return send_file(example_file_path, as_attachment=False,
mimetype='text/plain')

@app.route('/upload', methods=['POST'])
def upload_files():
      dat_file = request.files.get('dat_file')
      pdb_file = request.files.get('pdb_file')

      if not dat_file or not pdb_file:
      return "Please upload both .dat and .pdb files", 400

      dat_file_path = os.path.join(app.config['UPLOAD_FOLDER'],
dat_file.filename)
      pdb_file_path = os.path.join(app.config['UPLOAD_FOLDER'],
pdb_file.filename)

      dat_file.save(dat_file_path)
      pdb_file.save(pdb_file_path)

      try:
      result = a_s_n.run_sectors(dat_file_path, pdb_file_path)
      except Exception as e:
      print(f"Error while processing files: {e}")
      return "An error occurred while processing the files.", 500

      return render_template('result.html', result=result)

if __name__ == '__main__':
```

```
os.makedirs(app.config['UPLOAD_FOLDER'], exist_ok=True)
app.run(debug=True)
```

Appendix E: upload.html

```
<!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="UTF-8">
	<title>Run Sectors</title>
</head>
<body>
	<h1>RUN SECTORS WEBSITE</h1>
	<h2>Upload .dat and .pdb files</h2>
	<form action="{{ url_for('upload_files') }}" method="post"
enctype="multipart/form-data">
	<label for="dat_file">Choose .dat file:</label>
	<input type="file" id="dat_file" name="dat_file"
accept=".dat" required><br><br>
	<label for="pdb_file">Choose .pdb file:</label>
	<input type="file" id="pdb_file" name="pdb_file"
accept=".txt, .pdb" required><br><br>
	<input type="submit" value="Upload">
	</form>
	<br>
	<button onclick="showExample()">Example</button>
	<pre id="example-content" style="display: none; border: 1px
solid #ccc; padding: 10px;"></pre>

	<script>
	function showExample() {
		fetch('{{ url_for("example_file") }}')
		.then(response => response.text())
		.then(data => {
			const exampleContent =
document.getElementById('example-content');
			exampleContent.style.display = 'block';
			exampleContent.textContent = data;
		})
		.catch(error => {
			console.error('Error fetching example file:',
error);
		});
	}
	</script>
</body>
</html>
```

Appendix E: result.html

```
<!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="UTF-8">
	<title>Sectors Results</title>
</head>
<body>
	<h1>Results</h1>
	<p>{{ result }}</p>
</body>
</html>
```