9-8-2020

Development and implementation of a coarse-grained Markov State Model

Brian Chen
Rowan University

DEVELOPMENT AND IMPLEMENTATION OF A COARSE-GRAINED MARKOV STATE MODEL

by

Brian Chen

A Thesis

Submitted to the
Department of Bioinformatics
College of Science and Mathematics
In partial fulfillment of the requirement
For the Degree of
Master of Science in Bioinformatics
at
Rowan University
June 30, 2020

Thesis Advisor: Chun Wu, Ph.D.

Acknowledgements

Foremost, I would like to thank Dr. Chun Wu, my graduate advisor, teacher, and mentor. This work would not have been possible without his constant support and invaluable insights. Additionally, I would like to express my gratitude to all the members of the Wu Lab, including but not limited to Holli-Joi Sullivan, Griffin Fountain, and Nicholas Paradis for their help in many collaborative works. Lastly, I would like to thank my mother and father for their unwavering support and dedication.

Abstract

Brian Chen
DEVELOPMENT AND IMPLEMENTATION OF A COARSE-GRAINED MARKOV STATE MODEL
2019-2020
Chun Wu, Ph.D.
Master of Science in Bioinformatics

Markov State Models (MSMs) are constructed from Molecular Dynamics (MD) simulation data, high-resolution spatial and temporal information stored in the form of trajectories, of biological processes, such as ligand-receptor bonding, as a model to understand detailed kinetic information. Traditional MSM implementations involve a clustering step that clusters MD trajectories into thousands of experimentally unverifiable clusters known as “microstates” before lumping them together into “macrostates”. This work details a novel software implementation, using a combination of R, Python, and Tcl, that I have created for the purpose of creating a coarse-grained MSM that directly clusters the MD trajectories into a handful of experimentally verifiable clusters while maintaining the Markovian property. The coarse-grained MSM implementation was designed to require minimal technical experience while still being robust enough for usage in studying a variety of biological processes. In addition, this coarse-grained MSM implementation has already been used as part of several works to explore the binding mechanisms of various ligand-receptor complexes that have shown potential in the treatment of neurodegenerative diseases and various cancers.

Table of Contents

Table of Contents (continued)

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Ligand-receptor binding is a major biological process in which a ligand, usually some sort of small molecule, binds to a receptor, some sort of target molecule (Changeux & Edelstein, 2011). Molecular Dynamics (MD) simulations are a powerful tool that can be used to obtain high-resolution information about many biological processes and puts them in a series of snapshots, otherwise known as trajectories (Jirí Sponer & Spacková, 2007). Markov State Model (MSM) analysis, a method to further analyze the kinetic information provided by MD simulations, traditionally separates these snapshots into thousands of “microstates” via clustering before performing its analysis (Pande, Beauchamp, & Bowman, 2010). However, these “microstates” are experimentally unverifiable. In this thesis, the development of software to perform a modified version of MSM analysis, hereafter referred to as “coarse-grained MSM analysis”, that uses “macrostates”, created by clustering those snapshots into a handful of experimentally verifiable clusters, is detailed and applied to several ligand-receptor systems, providing invaluable kinetic information.

1.2 Biomolecule Dynamics

Folding and binding are two major biochemical processes essential to biological activity. Protein folding is the process in which a protein structure determines its 3-dimensional shape, otherwise known as its tertiary structure (Dill, Ozkan, Shell, & Weikl, 2008). This tertiary structure plays a key role in determining the protein’s biological function. On the other hand, binding can apply to several biological mechanisms such as but not limited to enzyme-substrate binding or ligand-receptor binding (Changeux & Edelstein, 2011). Of particular importance to this thesis is ligand-receptor binding, usually a process in which a small molecule binds to a target structure.

1.3 MD Simulations

Molecular dynamics simulations provide high spatial and temporal resolution in the form a series of snapshots, otherwise known as a trajectory. MD simulations are powerful tools that have been used to offer insights into many biological processes such as protein folding, ligand-receptor binding, and other dynamic processes that cannot be adequately captured by static methods such as NMR or X-ray crystallography. MD simulations have become a powerful and valuable tool over the past two decades for discovering details of biological processes (J. Sponer, Cang, & Cheatham, 2012; Jirí Sponer & Spacková, 2007; Zhu, Xiao, & Liang, 2013). Molecular modeling techniques are widely used to understand the binding of small molecules and provide a good structure model of these complexes. Several studies have used molecular modeling techniques in exploring the anti-cancer effect of various small molecules (Bhat, Mondal, Sengupta, & Chatterjee, 2017; Buket, Clement, & DanZhou, 2014; Dai, Carver, Hurley, & Yang, 2011; Deng, Wickstrom, Cieplak, Lin, & Yang, 2017; Kang & Park, 2009; Ma et al., 2012).

1.4 Clustering

Clustering is the act of grouping a set of *n* samples using *m* features, parameters of the samples that can be measured for similarity, into *k* groups, otherwise known as clusters. Various clustering algorithms can be used for any given cluster analysis task and there currently is not one best algorithm for all tasks. Clustering algorithms can be divided into two primary categories: supervised and unsupervised. Supervised clustering uses training data, data where the correct clusters have already been determined, to learn how to cluster the experimental data. Unsupervised clustering does not have training data and simply applies the algorithm directly to the experimental data. While there are other subdivisions for clustering algorithms, they do not affect the main topic of this work significantly and thus they will be excluded from mention. Typically, with MD simulations the clustering algorithms that are used are unsupervised clustering algorithms; there are likely many reasons for this but one of them is that there is simply not enough suitable training data for most systems (Pande et al., 2010). Three unsupervised clustering algorithms are frequently used in conjunction with MD simulations: single-linkage, spectral clustering, and *k*.

Single-linkage clustering performs an analysis of each individual sample compared against every other individual sample within a given cutoff in order to cluster the samples. Though typically very accurate, the long runtime of this sort of clustering limits the amount of clustering problems to which it can be applied to problems that have a small sample size. Spectral clustering is often an even slower algorithm than the single-linkage clustering and clusters based on the eigenvalues of a similarity matrix that is constructed from the original data. Similar to single-linkage clustering, spectral clustering is often seen as impractical unless the sample size is small. On the other hand, *k*-means clustering partitions each sample into one of *k* clusters such that within-cluster variance is minimized. Comparatively, *k*-means clustering is a significantly faster algorithm that can be run even on relatively large datasets, but may be slightly lesser in terms of accuracy for clustering MD trajectory frames.

1.5 MSM

Markov State Models are constructed from MD simulation trajectories and are used to understand detailed kinetic information about biological processes (Pande et al., 2010). MSMs divide the simulation data into various clusters called “states” and provide the kinetic information by analyzing the stochastic process of the transition between states at equilibrium. It maintains the Markovian property in that the model is memoryless such that the probability to transition to a state is only dependent on the present state. One of the major issues with the traditional MSM implementation methodology is that it uses thousands of states called “microstates” that are not experimentally verifiable.

Chapter 2

Coarse-Grained MSM Analysis

2.1 Introduction

Chapter 2 will discuss the theory and implementation of a coarse-grained MSM analysis using the software I developed. The general principles of the code and relevant equations are provided. All MSM analyses were predominantly performed in R, with some usage of VMD and Python. Some basic requirements to using this software will be a basic understanding of how to use R graphical interfaces, such as RStudio or RGui, as well as a basic understanding of VMD and its atom selection language.

2.2 Installation & Setup

This coarse-grained MSM program was developed to run on the Ubuntu operating system version 18.04 but should be compatible with newer versions. Support for Windows and Mac operating systems has not been developed but the coarse-grained MSM program should be compatible assuming the requisite packages support those operating systems. Development in R was performed using R version 3.6.0 but may require newer versions based on the requisite R packages. Similarly, Python version 3.5.1 was used in the development of this software but newer versions may be required based on the requisite Python packages. The list of R packages used in this software include ggplot2, reticulate, foreach, doParallel, igraph, expm, and R.utils. Installation of these R packages will be handled by the coarse-grained MSM software and no extra user installation is required outside of the R programming environment. In addition to installation of the Python environment, user installation of the following Python packages, either through pip or Anaconda, and their prerequisites is required: mdtraj, numpy, os, scikit-learn. It is recommended to install the latest version of VMD. After successful installation of requisite software, create a directory and extract the software from the archive. The coarse-grained MSM software can be run from the file “main.R” (see Appendix A). The code must be run as instructed in the subsequent parts of this chapter and general overviews of each line of code in “main.R” will be discussed. Some user modifications to the input values are required and these will be noted. Optional modifications to user input will not be discussed in this thesis but can be derived from the source code (Appendices A-D). Commented lines, denoted by the symbol “#”, are not part of the code but may contain helpful user instructions. While the user runs the coarse-grained MSM analysis in main.R (Appendix A), most of these call functions in functions.R (Appendix B) or functions.py (Appendix C). Trajectory-related calculations were performed in Python because most trajectory information is handled fairly robustly by the Python package “mdtraj”.

Configuration settings can be found in “config.txt” (Appendix E). Line 1 declares the selection to be defined as the ligand using the mdtraj atom selection language, which is fundamentally similar to the VMD atom selection language. Edit the text inside the quotations, “resname SPR”, to the desired selection. Lines 2-4 declare the type of trajectories being used as inputs, the name of the outputted merged trajectory, and the merged trajectory’s file type respectively. Edit the text inside the quotations as desired but note that the trajectory file types must be supported by mdtraj. Line 5 declares the topology file name and should be edited to fit the user’s system. This software was only designed to run with PDB topology files, but should also work with any file types supported by mdtraj. Line 6 can be set to “TRUE”, without the quotation marks, to also calculate the Root Mean Square Deviation (RMSD) values, the measure of the distance between the atoms of each frame, for each atom of the ligand individually. This does not impact the RMSD calculation of all ligand heavy atoms combined. Line 7 can be set to “FALSE”, without the quotation marks, to not calculate the center of mass. Line 8 does not require editing unless the user desires to change the features file’s filename. Lines 9-10 declare the minimum *k* value and the maximum *k* value respectively, for which the *k*-means clustering will be run, inclusively. Line 11 can be set to “FALSE”, without the quotation marks, to not normalize the data in the features file prior to clustering. Line 12 declares the reference frame to be used in the RMSD calculations and does not necessarily need to be changed. It is recommended from my observations to use a frame in which the ligand is positionally centralized. Line 13 declares the number of cores to use for the transition matrix calculations and can be left as 1 if the number of cores is unknown. Line 14 declares the filename for the finalized cluster file and does not need to be changed.

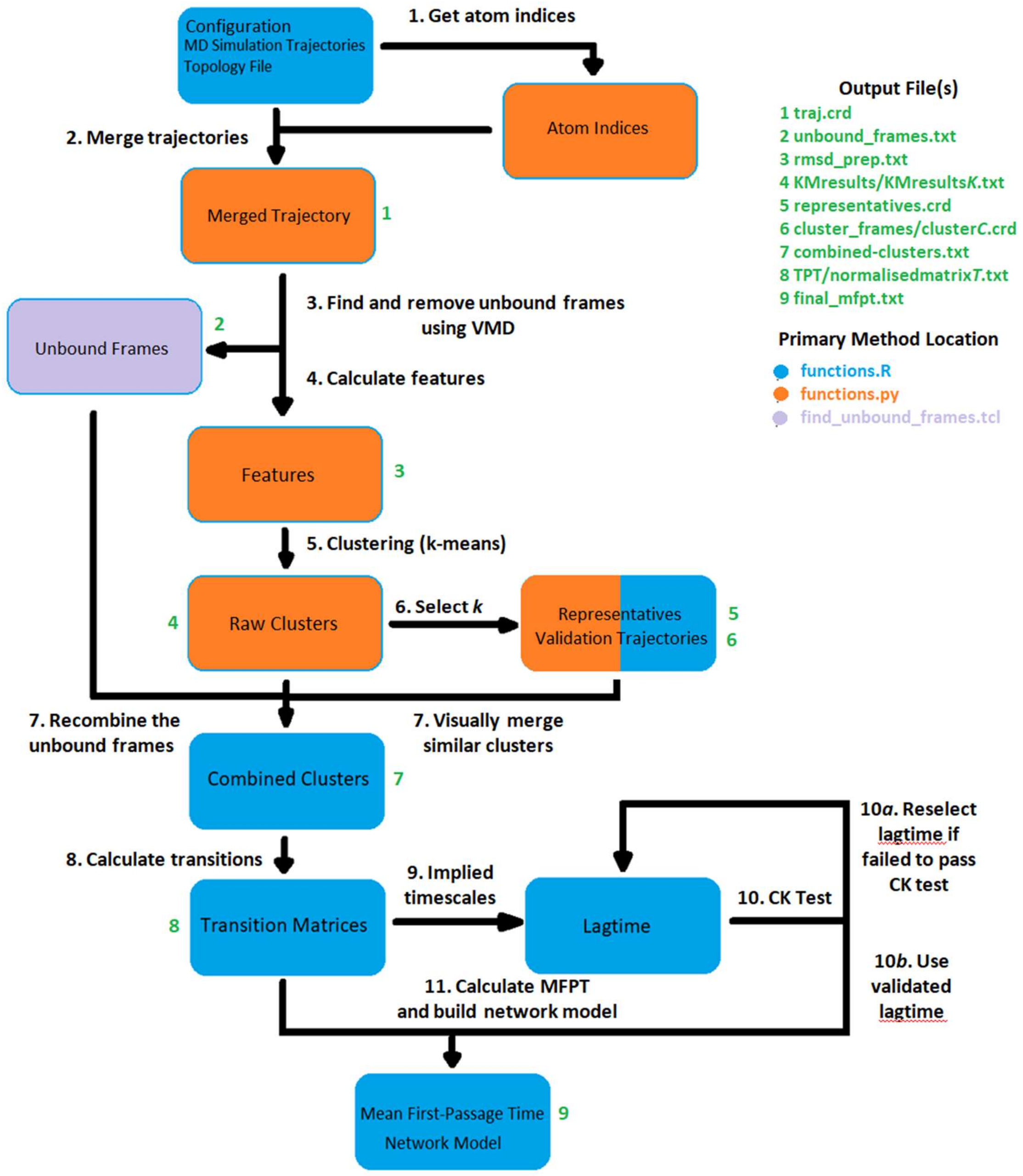


Figure 1. Coarse-grained MSM construction flowchart. The flowchart follows sequence of numerically listed instructions along the arrows. Instructions with the same number are considered to be performed in the same step. The location of the code responsible for each task is color coded for functions.R (blue), functions.py (orange), and find_unbound_frames.tcl (purple). Output files are labelled in green as numbers corresponding to files in the legend.

Run line 4 of “main.R” to load some of the created functions that are necessary to perform the coarse-grained MSM analysis and continue running the following lines as instructed. Line 5 checks to make sure all required R packages have been installed and proceeds to install any that are missing. Line 6 loads the package “reticulate” which is necessary for interactions with the Python code. Line 7 loads the created Python functions that are necessary for the coarse-grained MSM analysis into the R environment. Lines 10-11 create the necessary directories and load the configuration file. Move all desired MD simulation trajectories into the directory named “trajs”. It is important to note that while the software is robust enough to handle trajectories of differing lengths, it is still highly recommended and more convenient for the user to use trajectories consisting of the same length. Move the topology file into the working directory and confirm that the name of the topology file is consistent with the one listed in the configuration file. Lines 14-15 will extract the atom indices of the selections specified in the configuration file and merge all the trajectories into a single trajectory that will be created in the working directory.

2.3 Clustering & Featurization

2.3.1 Clustering theory I. Standard MSM construction typically involves initially clustering trajectory data into thousands of individual clusters termed “microstates”. Subsequent analysis is then performed on these microstates before they are hierarchically lumped into “macrostates”. As there does not exist any experimental evidence indicating that these thousands of microstates all actually exist, we chose to directly cluster into experimentally verifiable macrostates. This variance in procedure leads to a more coarse-grained model which trades finer detail for more experimental testability and greater human understanding. The clustering for coarse-grained MSM construction is divided into three steps. First, the unbound frames are identified and subsequently removed based on the number of atom contacts between the ligand and target molecule. The unbound frames must be removed prior to the clustering via *k*-means because the distribution of the unbound frames will likely result in hundreds or thousands of clusters (Figure 2). Second, the remaining frames are clustered based on RMSD and optionally center of mass via *k*-means clustering. Lastly, the clustering results are analyzed and validated before subsequent merging based on visual expertise.

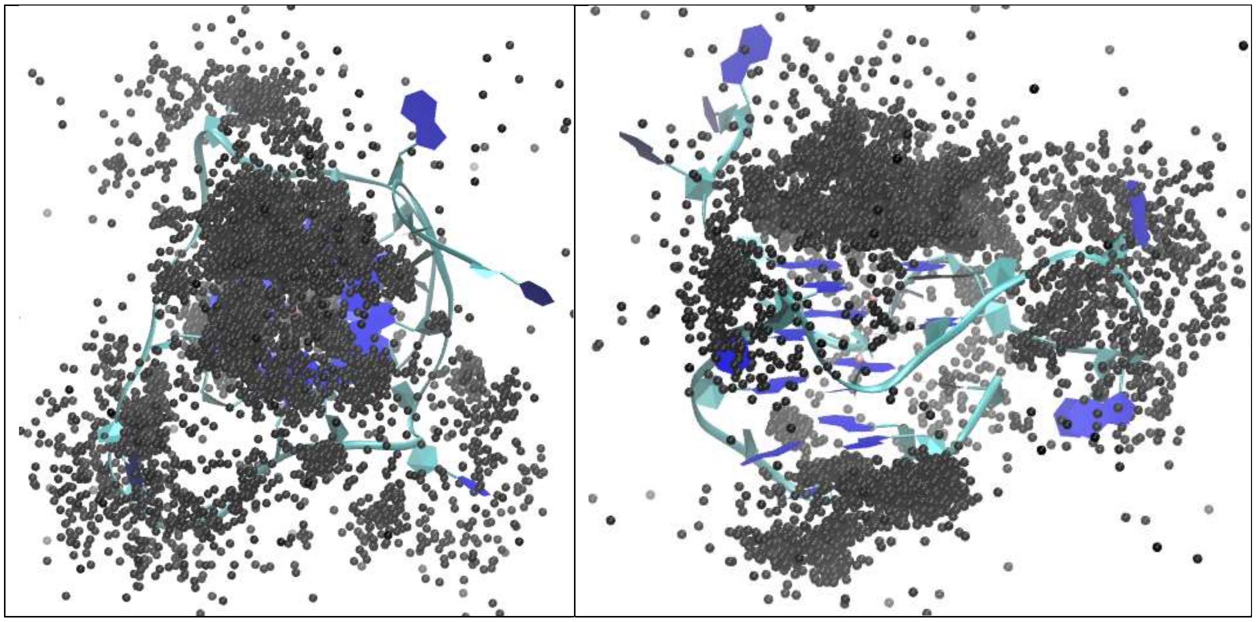


Figure 2. Distribution of ligand. Top (left) and side (right) views of the distribution of the ligand, DBD1, positions over the merged free ligand binding trajectories. The less dense outer distributions are indicative of the unbound frames.

2.3.2 Clustering implementation I. Using VMD, load the topology file and the merged trajectory file. The file “find_unbound_frames.tcl” (Appendix D) can be loaded through the Tk Console in order to output all frames in which there are less than a user-defined number of atom contacts within a user-defined cutoff between the ligand and receptor are to a separate file called “unbound_frames.txt”. Instructions for editing the “find_unbound_frames.tcl” script can be found in Appendix D. The script sequentially searches through all frames of the merged trajectory and stores a list of all frame numbers in which the cutoff was not met. This definition of the unbound frames by cutoff is part of what makes this implementation a coarse-grained model, as traditional MSM clustering directly clusters all of the trajectory data, including the unbound frames, into thousands of “microstates”.

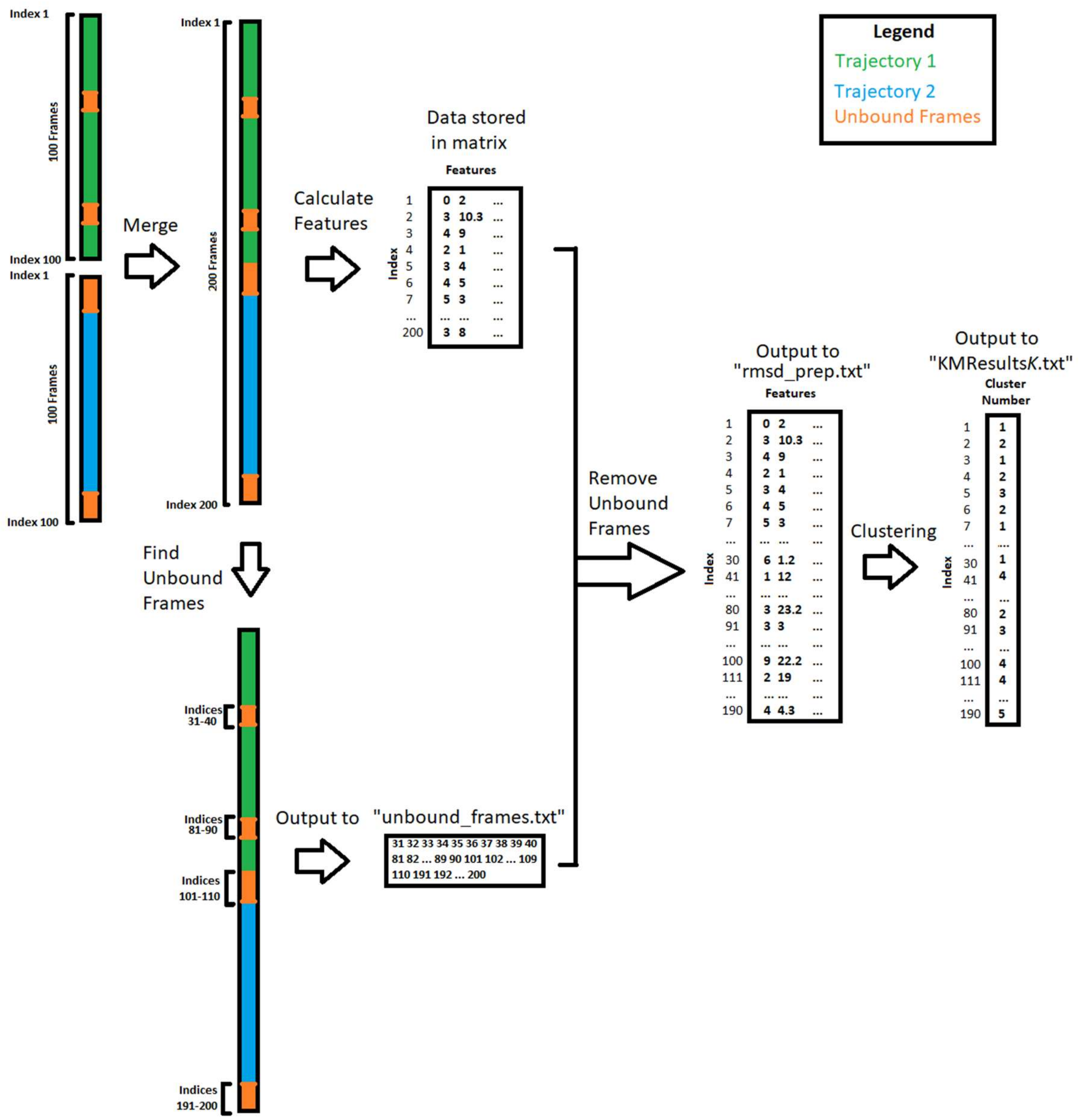


Figure 3. Representation of initial clustering. Trajectories are depicted as bars with color coding to indicate the trajectory number and unbound frames.

2.3.3 Featurization theory. Root mean square deviation (RMSD), in terms of MSM analysis, is a measure of the difference between the positions of a selection of atoms in a given frame compared to a reference frame. RMSD is calculated according to the equation:

$$RMSD = \sqrt{\frac{\sum_1^N (\hat{\theta} - \theta)^2}{N}}$$

For a given atom, $\hat{\theta}$ is a coordinate position while θ is the reference position. Center of mass is a measure of the point where the weighted relative position of a distribution of mass in space equals zero and is calculated according to the equation:

$$Center\ of\ Mass = \frac{\sum_1^N (x * m)}{\sum_1^N m}$$

For a given atom, *x* is a coordinate position and *m* is the mass.

2.3.4 Featurization implementation. From this point on, the user will only need to run code in main.R. Line 25 is optional and is only necessary if the user wishes to change the RMSD reference frame prior to feature calculation. Run line 28 to call a function (lines 44-71 of Appendix C) to calculate the RMSD of the ligand heavy atoms and possibly also the RMSD of each individual heavy atom of the ligand as well as the center of mass of the ligand heavy atoms (lines 73-121 of Appendix C) over all the frames of the merged trajectory sequentially based on the user settings in the configuration file. Heavy atoms are defined as non-hydrogen atoms. The RMSD calculations are performed according to the equation listed in section 2.3.3. Similarly, the center of mass calculations for each of the x, y, and z axes are performed according the equation for center of mass listed in section 2.3.3. The RMSD and center of mass values are outputted in Angstroms. The center of mass calculations can only be performed for molecules with heavy atoms consisting of carbon, nitrogen, oxygen, phosphorous, or sulfur atoms. Line 29 combines the feature information obtained in line 28 into a single data table. Rows denote the frame number while columns are ordered such that the overall ligand RMSD is in the first column followed sequentially by a separate column for the individual RMSD values of each of the ligand heavy atoms followed by three columns for the x, y, and z coordinates of the center of mass if these options were selected in the configuration file. Next, it deletes the rows corresponding to the list of frame numbers in the "unbound_frames.txt" file and outputs the remaining data to the file under "rmsdfile" in the configuration settings. This step effectively removes all of the previously defined unbound frames prior to the actual clustering step and is a major part of how this implementation is a coarse-grained model.

2.3.5 Clustering theory II. It is important to note that this analysis was initially designed for usage in analyzing G-quadruplex complexes, but is robust enough to be used for the analysis of many biological processes. The clustering used in this coarse-grained MSM analysis is *k*-means clustering. There are several reasons for choosing to use *k*-means clustering over the potentially more accurate single-linkage clustering or spectral clustering. Firstly, the sample size for the clustering of trajectory frames tends to be in the thousands or tens of thousands, which makes single-linkage and spectral clustering highly impractical without significant computational resources and hefty hardware. Secondly, single-linkage clustering implementations are traditionally run on a single feature while the inherently visible shape of many G-quadruplex structures proved indicative that an additional set of features, center of mass, would prove to be highly valuable in determining how to cluster the data. This can be seen in Figure 2, which shows the distribution of the ligand simulated with a G-quadruplex structure over the entirety of the frames. Lastly, a practical test using *k*-means clustering with the aforementioned features resulted in the same number of clusters and approximately similar percent abundance values for each cluster as a single-linkage algorithm by GROMACS using only RMSD after the results of both clustering algorithms were separately visually merged.

2.3.6 Clustering implementation II. Line 32 of main.R (Appendix A) will call a function (lines 123-142 in Appendix C) to cluster the data in the "rmsdfile". The data is normalized prior to clustering and clustering is performed through a *k*-means algorithm where the user defined a minimum bound for K and a maximum bound for K such that clustering will be performed for each K from minimum to maximum inclusively. The silhouette indices for each of these cluster iterations is calculated and stored in the directory "KMresults" as "KMSIresults.txt". The results of each cluster iteration are also stored in the directory "KMresults" as separate "KMresults#.txt" files, where # is the value of K and the data is stored as a single column containing the cluster ID for each frame sequentially. Line 33 calls a function (lines 92-109 of Appendix B) to plot the silhouette indices in R as a measure of how accurate each clustering iteration was compared to the others. Typically, picking the K corresponding to the greatest silhouette index will result in a good clustering result. Line 36 requires the user to change the value to the K value picked in the previous step. This will allow the rest of the code to know which K value to use for subsequent validation and other analyses.

2.3.7 Clustering theory III. Clustering rarely produces absolutely perfect results for complex datasets; thus, validation and other analyses are required. Firstly, the identification of a representative of each cluster can be obtained by finding the frame for each cluster that has an RMSD value nearest the mean RMSD value for that cluster. Next, validation of the integrity of each cluster can be obtained by visually observing every frame for each cluster and comparing to see if the majority of each cluster is similar. Lastly, clusters can be merged based on visual expertise through comparison of the representatives for each cluster. This should result in a handful of “macrostates” that are mostly experimentally verifiable, unlike the unverifiable thousands of “microstates” that are used in traditional MSM analyses. The clustering theory detailed in these three clustering theory sections (2.3.1, 2.3.5, 2.3.7) shows why this MSM analysis is considered coarse-grained.

2.3.8 Clustering implementation III. Line 39 creates a trajectory containing the representative frames of each cluster for the previously selected K value. First, the representative frame number for each cluster is found via lines 118-199 in functions.R (Appendix B). This function (lines 118-199) loads all of the feature data and combines it with the cluster data for each frame (lines 119-133). It then adds all the frames that were removed as unbound frames back into the dataset and sets the unbound frames as a separate cluster (lines 136-162). It then iterates over the dataset to obtain the average RMSD value of each cluster (lines 165-179). It iterates again over the dataset while comparing the RMSD value of each frame against the mean value of the respective cluster to obtain the frame with the least difference from the mean RMSD value for each cluster (lines 182-195). Line 42 of main.R calculates the number of frames for each cluster. This function (lines 201-215 of Appendix B) iterates over the dataset and counts the number of times each cluster appears. The frames per cluster can be shown as a percentage or as a raw value based on whether the user declares the “percentage” as

“TRUE” or “FALSE”. The example given in Appendix A will show the result as a percentage.

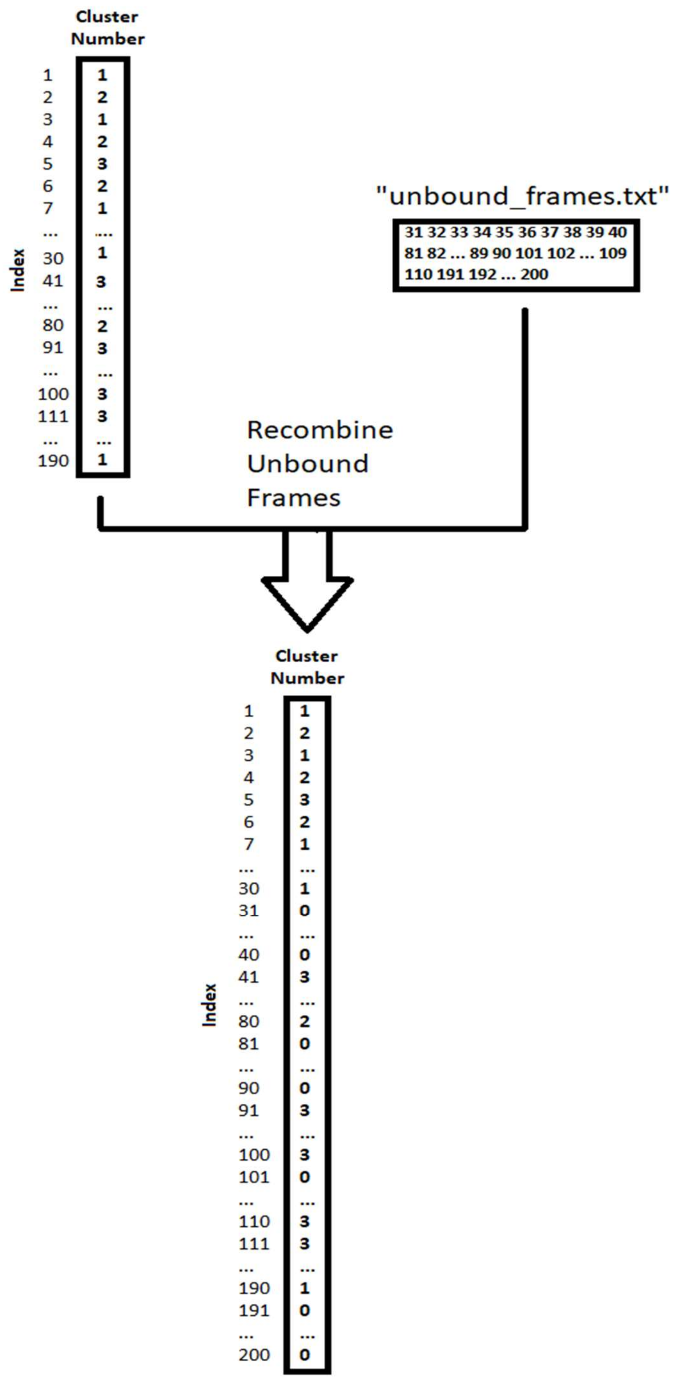
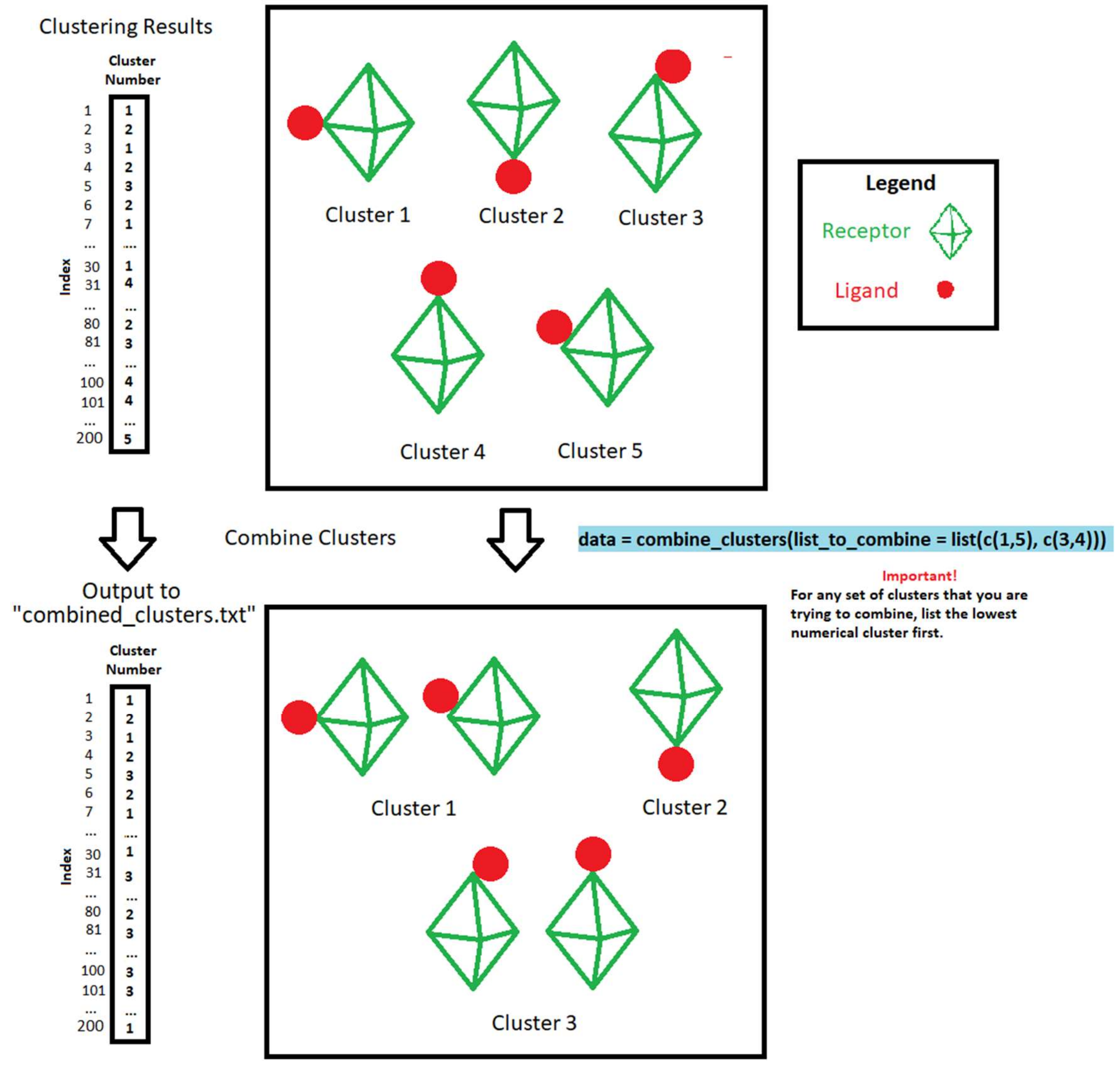


Figure 4. Representation of recombining unbound frames. Note that the unbound frames are inserted back into this representation as cluster 0 (this may not necessarily be the case for the actual software implementation).

Line 45 of main.R creates a trajectory for each cluster that contains all of the frames in the respective cluster. First, a function to find the frames that belong to each cluster (lines 223-245 of Appendix B) iterates over the dataset checking the cluster number and adds the frame number to the end of a list containing all the frames for that cluster. Each of these lists are then outputted to files in the directory “cluster_frames” by cluster number. Then, a function to create separate trajectories containing all the frames for each cluster (lines 163-186 of Appendix C) loads the merged trajectory and then subsets the list of frames for each cluster before outputting the trajectory into the directory “cluster_frames”. Visual analysis of the representatives and validation trajectories can be used to identify the integrity of the clustering results as well as the selected K value. If validation identifies that the clustering analysis produced significantly inaccurate results, changing the reference frame or the selected K value and then repeating the previous steps can improve the integrity of the clusters. After validating the clusters, running line 48 in main.R will recombine the unbound frames (lines 247-272 in Appendix B) that were previously removed back into the data set as cluster 1 (lines 253-270) and renumbering the other clusters to follow while still maintaining their identities (lines 248-250).



```
data = combine_clusters(list_to_combine = list(c(1,5), c(3,4)))
```

Index	Cluster Number
1	1
2	2
3	1
4	2
5	3
6	2
7	1
...	....
30	1
31	4
...	...
80	2
81	3
...	...
100	4
101	4
...	...
200	5

Index	Cluster Number
1	1
2	2
3	1
4	2
5	3
6	2
7	1
...	....
30	1
31	3
...	...
80	2
81	3
...	...
100	3
101	3
...	...
200	1

Figure 5. Representation of cluster combination. The actual data transformation (left) and a visual example (right) of how to merge the clusters are shown. The command shown in blue is line 57 of main.R (Appendix A). The visual example of the clusters depicts the receptor as a green crystal-like object in complex with the ligand, a red sphere.

Line 57 of main.R requires user input to merge similar clusters. This is very important as it helps reduce the number of clusters to an experimentally verifiable amount and is a major part of what makes this analysis a coarse-grained MSM. User input is required in similar syntax as shown in line 57 of Appendix A such that the lowest clusters to be combined are listed first. Detailed instructions are shown in the commented lines of Appendix A (lines 51-54). The function (lines 280-317 of Appendix B) iterates over the entire dataset multiple times as specified by the user input, searches for values matching the user input, and then sets those values equal to the lowest value given in the user input for that iteration (lines 282-291). The function then proceeds to clean up the values by finding the unique cluster values remaining (lines 294-300), sorting them, and then subsequently renumbering the values to increment from 2 onwards while maintaining the cluster identities and order (lines 303-313). Line 60 of main.R writes the data out to a file "combined_clusters.txt" for further use. Line 63 is an optional line that obtains the frames per cluster, representatives, and validation trajectories for the combined clusters. The theory and implementation remain the same and only the data has changed.

2.3.9 Transition path theory. Count matrices are created for specified lagtimes (τ), increasing in magnitude, by counting the number of observed transitions between discrete states such that the count of transitions from state *i* to state *j* (c_{ij}) is the sum of the number of times each of the trajectories were observed in state *i* at time *t* and in state *j* at time $t + \tau$, for all $t \leq t_{max} - \tau$ (Prinz et al., 2011). The count matrices were symmetrized (sym_{ij}) such that:

$$sym_{ij} = sym_{ji} = \frac{c_{ij} + c_{ji}}{2}$$

and then row-normalized ($norm_{ij}$) such that:

$$norm_{ij} = \frac{sym_{ij}}{\sum_{j=1}^{j=n} sym_{ij}}$$

For the purpose of determining the lag time at which the model has converged, an implied timescale of each cluster is calculated according to the equation:

$$\tau_k = -\frac{\tau}{\ln \mu_k(\tau)}$$

in which μ_k is an eigenvalue of the transition matrix, for all specified lagtimes and plotted. The implied timescale of the first cluster is not included in the plot as the eigenvalue is always 1 and thus contributes no information (Noé, Horenko, Schütte, & Smith, 2007). Further validation that the model had converged was performed through the Chapman-Kolmogorov (CK) test using the equation:

$$P_{MD}(n\tau) = [P_{MSM}(\tau)]^n$$

in which P is the transition probability at a specific time, n is a constant, τ is the lagtime. Thereafter, the mean first passage times (F_{if}) at a specified optimal lag time, found via the implied timescales and Chapman-Kolmogorov test, can be calculated according to the formula:

$$F_{if} = \tau + \sum_{j \neq f} P_{ij} F_{jf}$$

with the boundary condition $F_{ff} = 0$, where τ is the lag time used to construct the transition matrix $P(\tau)$. Standard deviations can be obtained by performing the same mean first passage time calculations on a range of values near the optimal lag time. A network model can then be generated based on the count matrix at the optimal lag time with a cutoff of a desired number of transitions.

2.3.10 Transition path theory implementation. Line 68 of main.R (Appendix A) is an optional line used to declare a series of lagtimes for the subsequent steps. The software has already generated a default series of lagtimes, incrementing by 1 from 1 to the length of the shortest MD trajectory prior to merging, in a previous step. However, the user can opt to set a larger increment thereby saving some time and detailed instructions for how to declare line 68 are shown in lines 66-67. Typically, it is necessary at minimum to set a series such that it includes 1, and steps by 10 from 10 to the length of the shortest MD trajectory rounded to the nearest 10. The example shown in Appendix A denotes a series of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, … 490, 500. Line 70 calculates the transition matrices according to the equations mentioned in section 2.3.9 and outputs the counted, symmetric, and normalized matrices into corresponding text files in the TPT directory. The functions (lines 353-386 and 388-434 of Appendix B) responsible for this use a parallel processing technique to heavily speed up the overall calculations based on the value set in line 13 of the configuration file. Lines 362-371 are performed for each lagtime in the series and iterate over the entire dataset while counting the number of observed transitions from *i* to *j* as detailed in section 2.3.9 for each individual trajectory prior to the merging. Lines 374-383 perform the same actions as lines 362-371 but are only used instead of lines 362-371 when the original input trajectories consist only of a single trajectory.

Line 73 of main.R (Appendix A) creates a plot of the implied timescales, an example is shown in Figure 6, for the previously declared lagtimes using the matrices calculated in the previous step. The implied timescales function can be found in lines 436-500 of Appendix B and calculates the implied timescales according to the equation listed in section 2.3.9. The function uses singular value decomposition for the calculation of the eigenvalues as the matrices used in MSM analyses fulfill the conditions such that values obtained by singular value decomposition will be equal to that of eigen decomposition. The user can choose to use the eigen solver to calculate the eigenvalues by setting the variable f to any string other than “svd”. This is not recommended, however, as the eigen solver in R often has many issues when solving certain matrices. Lines 439-475 calculate the eigenvalues for each state at each of the previously specified lagtimes and stores the data in a matrix for further use. Lines 477-482 are responsible for performing the mathematical equation used for calculating implied time detailed in section 2.3.9 and then preprocessing the data for plotting. Lines 484-499 are responsible for the automated creation of a customized plot that shows the implied timescales with color coded states and optional logarithmic axis representations. Lines 76-77 of main.R are optional but recommended lines that can be used to refine the implied timescales plot for use in generating a good figure for use in publications. Line 76 is very similar to line 68 and is used to redefine the lagtimes in order to remove any lagtimes where the matrices may prove unreliable. These lagtimes are typically closer towards the maximum lagtime value as there is significantly less data to work with when generating the count matrices. Line 77 adds an option, log_y, to the implied timescales function that is set to TRUE by default to show a logarithmic y-axis which is often more useful when looking at implied time, but provides less information on finding unreliable data.

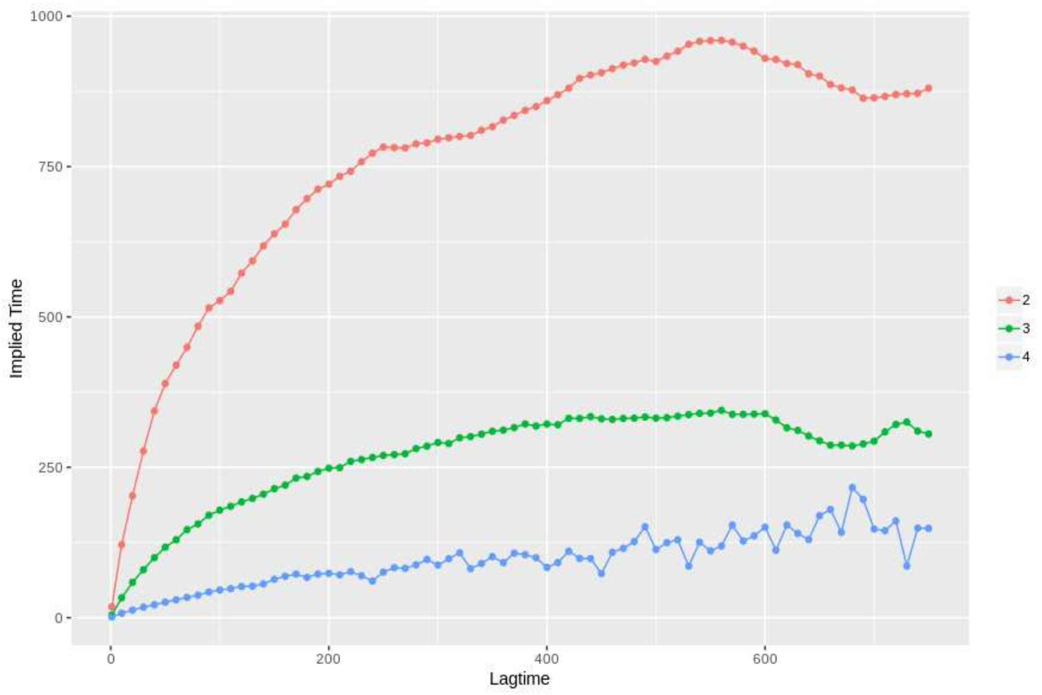


Figure 6. Plot of the implied timescales. States 2, 3, and 4 (side, top and bottom binding respectively) are shown. The lines have stabilized and all three bound states are converged by roughly 250 ns. Timescales are only shown up until 750 ns as the transition matrices after that are significantly less reliable.

Line 81 of main.R calls a function (lines 502-627 of Appendix B) that is responsible for creating the plot of the CK test. An example of the CK test is shown in Figure 7. User input is required for the option x and it must be set a value in the series of lagtimes declared in line 68. It is typically recommended to pick a value that has a large number of divisors and thus values that are multiples of 100 are highly recommended. Choosing a value that is not a multiple of 100 may lead to inaccuracies, insufficiencies, or legend errors when viewing the plot. User input is also required for the option cluster and it will affect which cluster the CK test is performed on. The recommended values for cluster are either the number of the most abundant cluster that is not unbound or the number of the unbound cluster if it is the most abundant. There are several optional variables that the user can choose to declare: final, steps, user_base_lagtimes, base_lagtimes, useBasin, and basin. These values are respectively used to cut off unreliable data, declare step size, declare whether or not the user intends to declare their own base lagtimes, the series of base lagtimes the user wants to use, declare whether or not to use basins instead of clusters, and the basin the user wants to use. It is not recommended to change the optional values, other than final, unless the user has an advanced understanding of the code used in lines 502-627 of Appendix B. Final can be changed to any value less than the value set for x so as to set the maximum value of the x-axis. Lines 506-534 are responsible for preprocessing and calculating the base lagtimes if the user did not declare any. Lines 535-551 are responsible for performing the Chapman-Kolmogorov equations detailed in section 2.3.9 on each of the matrices from the base lagtimes and storing the data. Lines 552-626 are responsible for preparing the data for plotting and the subsequent plotting of the CK test results.

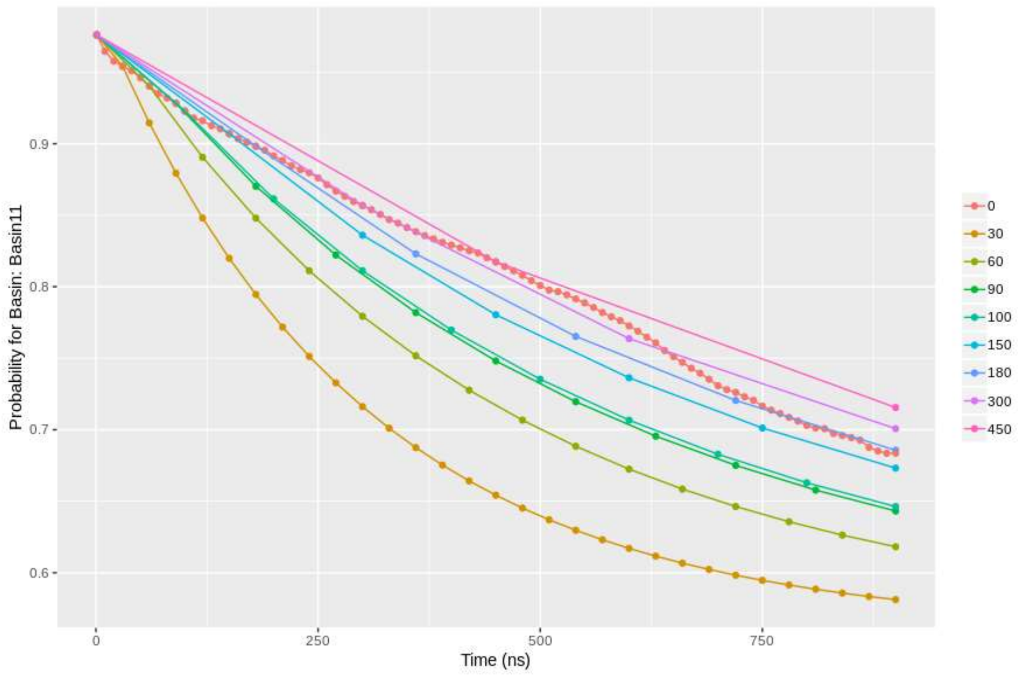


Figure 7. Plot of Chapman-Kolmogorov test. Shows the probability for Basin 11. Basin 11 is indicative of the probability that the simulation will remain in cluster 3, the most stable state (top binding mode). Each line denotes the approximation of a model for that given lag time. The line for 0 ns is the actual simulation data. As the line for 0 ns falls between the lines from 180 ns and 450 ns, it is clear that the model closely approximates the simulation between lag times of 180 ns and 450 ns.

The plots of the implied timescales and CK test should be interpreted together. The implied timescales plot allows the user to find an approximation of an optimal lagtime at which the system has reached equilibrium. This is defined by an area at which all states have begun to either plateau or oscillate between near constant boundaries and can be seen in Figure 6. The CK test allows the user to validate that the model is a sufficient representation of the actual data. This can be confirmed by noting which lines the line for lagtime 0, which represents the raw trajectory data, falls between and can be seen in Figure 7. The lines which the line for the raw trajectory is consistently between represents the minimum (lower line) that the optimal lagtime can be and an upper bound showing that the system has sufficiently relaxed. If the line for the raw trajectory does not fall between at least two lines, then either the user is using the wrong cluster for the CK test or the model is not sufficiently representative of the data.

Line 83 of main.R (Appendix A) requires user input to set the option optimal_lagtime to the value of the optimal lagtime obtained from the implied timescales and CK test. The function (629-700 of Appendix B) then proceeds to calculate the mean first-passage times for the given lagtime according to the equation listed in section 2.3.9. Lines 632-681 consist of a complex algorithm that automatically builds the system of equations necessary for the mean first-passage time calculations regardless of the number of states. Lines 683-699 are responsible of solving the system of equations, storing the results for further use, and returning the results to the user. Line 84 of main.R calls a function (lines 708-729 of Appendix B) to calculate a standard deviation for the mean first-passage times and requires the user to input 3 numbers such that the first is a lower bound, the second is an upper bound, and the third is the step size. In the example shown in Appendix A, this calculates the standard deviations between lagtimes of 200, 210, 220, 230, 240, 250, 260, 270, 280, 290, and 300. It is recommended that the user use two equidistant values from the optimal lagtime with a step size of 10.

Line 85 of main.R calls a function (lines 731-774 of Appendix B) to generate a network model for a given lagtime. The user must set the lagtime equal to the optimal lagtime, declare a cutoff for the minimum number of transitions necessary to show an edge, and optionally declare a seed that affects only the graphical representation of the model. The cutoff is typically recommended to be set as 1 so that all observed transitions are shown while the seed does not matter except to get the same graphical representation for the same seed. The network model is built using the R package igraph and represents each state as a node (circle) and the transitions as edges (lines). The edges are labelled with the number of transitions observed from a source state to a sink state. These labels are color coded such that the color of the label matches the color of the source node. An example of a network model is shown in Figure 8. Lastly, line 87 of main.R calls a function (lines 776-788 of Appendix B) to output the results of the mean first-passage time calculations to a text file containing a matrix such that the values are the mean first-passage times, the row number is the source state and the column number is a sink state. There are two optional values for the user to declare: divisor is the number by which to divide the mean first-passage time values if the user desires to convert units and decimals is the number of decimal places to which the values should be rounded. The data is outputted to a file called “final_mfpt.txt”. This concludes the coarse-grained MSM analysis.

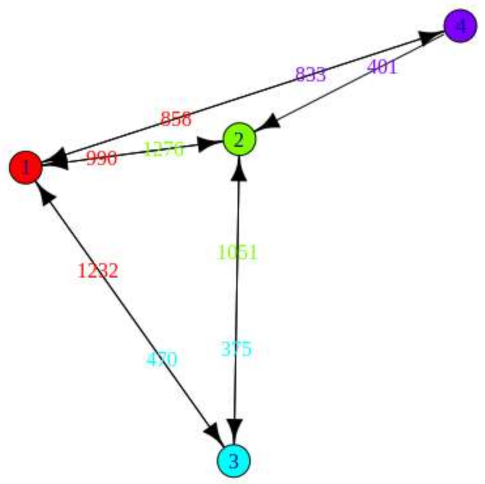


Figure 8. Network model. Shows the transitions between the (1) unbound, (2) side, (3) top, and (4) bottom states. Edge labels indicate the number of observed transitions and are color coded such that the color of the label matches the source of an outgoing transition.

Chapter 3

Coarse-grained MSM Analysis of G-quadruplex Systems

3.1 Chapter Overview

Chapter 3 serves as a demonstration of the real application of the coarse-grained MSM analysis for a ligand binding to a G-quadruplex. The following are several slightly modified excerpts from a soon-to-be submitted manuscript that was developed from an older version of the coarse-grained MSM software (Chen, Fountain, Sullivan, Paradis, & Wu, 2020). The final section of chapter 3 will also include several sets of results obtained from performing the coarse-grained MSM analysis (Mulholland et al., 2020; Sullivan, Chen, & Wu, 2020).

3.2 Introduction

A G-quadruplex consists of four guanine base pairs (G-tetrads) stabilized by Hoogsteen hydrogen binding and pi-pi stacking interactions (Suntharalingam, White, & Vilar, 2009). The G-tetrads form a planar arrangement within the structure and are further stabilized by the presence of monovalent cations, such as K^+ and Na^+ ions, interacting with guanine O6 carbonyl groups (Hsu et al., 2009). G-quadruplexes have been found in the promoter regions of numerous genes, including various oncogenes such as MYC, c-KIT, VEGF, and KRAS (Agarwal et al., 2011; Agrawal, Hatzakis, Guo, Carver, & Yang, 2013; Harikrishna, Kotaru, & Pradeepkumar, 2017). Investigations of new quadruplex-binding ligands that can stabilize the quadruplex structure have gained interest over the past two decades (Phan, Modi, & Patel, 2004; Ruggiero & Richter, 2018). Ligand binding-induced stabilization of the promoter G-quadruplex is being explored as a cancer therapy because of its ability to downregulate oncogene expression (Ambrus, Chen, Dai, Jones, & Yang, 2005; Neidle, 2016; Tawani, Mishra, & Kumar, 2017).

Overexpression of the MYC oncogene has gained a lot of attention due to its common genetic aberrations found in various types of human cancer cells, including breast (Watson, Safneck, Le, Dubik, & Shiu, 1993), prostate (Hawksworth et al., 2010), lung (Rapp et al., 2009), cervical (J Wu, 1996), colon (Smith, Myint, & Goh, 1993), small-cell lung cancer (Barr et al., 2000), and lymphoma (Kim, Evans, Dubins, & Chalikian, 2011; Magrath, 1990). The MYC gene encodes transcription factors, which regulate gene expression in many important biological processes such as cell growth, apoptosis and proliferation (Boddupally et al., 2012). Within the MYC promoter region, an important element, termed nuclease-hypersensitivity element III_1 (NHE III_1), is required for 80-95% of MYC transcription (Cooney, Czernuszewicz, Postel, Flint, & Hogan, 1988; T. L. Davis, Firulli, & Kinniburgh, 1989; Mathad, Hatzakis, Dai, & Yang, 2011). Pu27, the major purine rich strand of NHE III_1, (Ambrus et al., 2005; Chung, Heddi, Hamon, Teulade-Fichou, & Phan, 2014; J. T. Davis, 2004) forms G-quadruplex structure and ligand-binding induced stabilization of the MYC G-quadruplex has been shown to downregulate MYC expression as a cancer therapy (Ambrus et al., 2005; Tawani et al., 2017). In one study, suppression of MYC expression was observed when a Burkitt's lymphoma cell line was treated with TMPyP4, which stabilized the G-quadruplex (Tawani et al., 2017). Two other studies have also shown that quindoline derivatives stabilize the MYC G-quadruplex and reduce expression of MYC in cancer cells (Che et al., 2018; Deng et al., 2017). Therefore, the use of small molecules to stabilize the MYC G-quadruplex and consequently decrease MYC expression is an attractive anti-cancer therapeutic approach.

In an attempt to identify a new class of MYC G-quadruplex stabilizing ligands, Felsenstein et al. employed a small molecule microarray to screen 20,000 compounds from ChemBridge and ChemDiv repositories (Felsenstein et al., 2011), which yielded compound 1 (ChemDiv No. D089-0563), a crescent-shaped structure containing a G-quadruplex binding disubstituted benzofuran scaffold (hereafter referred as DBD1). Measurement of the binding affinity of DBD1 by surface plasmon resonance gave a K_d value of 4.5 ± 1.4 μM, which is sufficient to elicit a biological response (Neidle, 2016). DBD1, a G-quadruplex stabilizing ligand, also exhibited the ability to selectively inhibit MYC gene expression through stabilization of Pu27 in the MYC promoter region and induced apoptosis in cancer cell lines while having minimal toxicity on normal peripheral blood mononucleocytes (Neidle, 2016; Pany, Bommisetti, Diveshkumar, & Pradeepkumar, 2016). Although the high-resolution complex structure of DBD1 with the MYC G-quadruplex of Pu27 has not been obtained due to the polymorphic structure of Pu27, the NMR structure of a derivative, Pu24, (PDB: 2MGN) in complex with a different ligand has been solved.

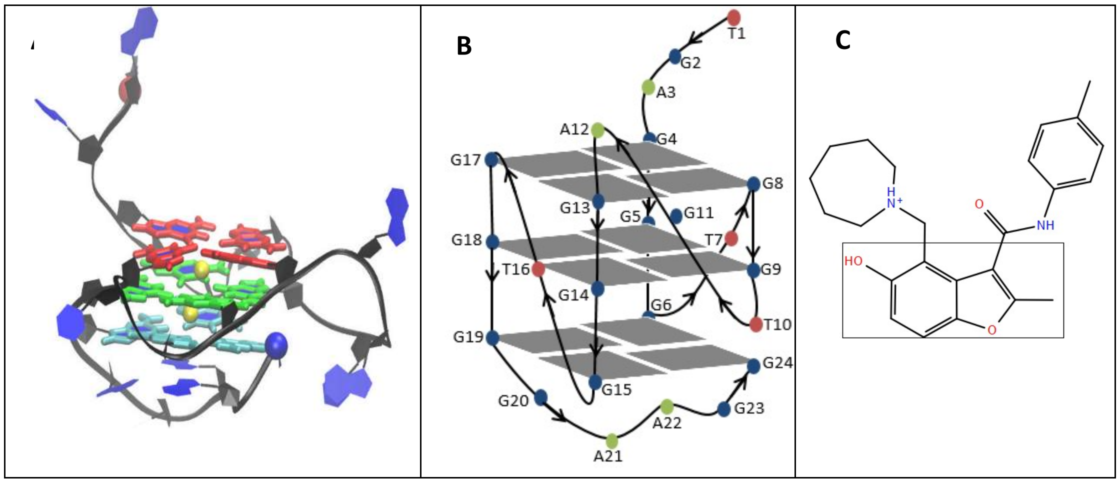


Figure 9. Structure of DBD1 and Pu24. Side view **(A)** and cartoon representation **(B)** of the Pu24 G-quadruplex (PDB ID: 2MGN) and the 2D structure of DBD1 (**C**). The G-tetrads are highlighted in red, green and cyan. The 5' and 3' ends are represented by red and blue spheres, respectively. K^+ cations are represented by yellow balls. The planar core region of the DBD1 structure is shown by a black box.

In this study, we employed extensive free ligand binding MD simulations to probe the binding of DBD1 to a MYC G-quadruplex derivative, Pu24 (PDB: 2MGN). Subsequent *k*-means clustering analysis of the MD simulation data were evaluated for determining the major binding modes. Molecular Modeling Poisson-Boltzmann Surface Area (MM-PBSA) analysis was evaluated for determining the energetics of the major binding modes. MSM analysis is performed to examine the binding pathway and kinetic rate information. Order parameters are calculated for representative trajectories.

3.3 Methods

3.3.1 MD simulation systems. A total of three systems were constructed: one ligand-only system, one DNA-only system, and one unbound DNA-ligand system. Each system was solvated in a water box of truncated octahedron with 10 Å water buffer plus Cl^- or K^+ as counter ions to neutralize the system and 0.1 M KCl. A refined OL15 version of the AMBER nucleic acid force including corrections of several backbone torsion angle parameters (i.e. parm99bsc0 + χ_{OL4} + ε/ζ_{OL1} + β_{OL1}) was applied to represent the DNA fragment. The TIP3P water model was used to represent water molecules, and the K^+ model developed by Cheatham group was used to represent the K^+ ions (Joung & Cheatham, 2008). The force field for DBD1 ligand was obtained using standard AMBER protocol: the molecular electrostatic potential (MEP) of DBD1 was calculated at the HF/6-31G* level after its geometry optimization at the same theory level; then MEP was used to determine the partial charges of DBD1 atoms using Restrained Electrostatic Potential/RESP method with two stage fitting; and other force field parameters were taken from the AMBER GAFF2 force field.

The simulations for each system were run using the AMBER 16 simulation package. The simulation protocols followed our early studies which are briefly described here. The starting points of the MD simulations for DBD1 to the G-quadruplex involved two different initial starting unbound points, top and bottom, with a separation of ~15 Å. Each unbound DNA-ligand system underwent an additional 1000 ps pre-run at 500 K to ensure that the position and orientation of the free ligand were randomized before a production run at 300 K; during this pre-run, the receptors position remained fixed. Thirty-three independent runs at 300 K were carried out using random initial velocities. A run at 300 K, included a short 1.0 ns molecular dynamics in the NPT ensemble mode (constant pressure and temperature) to equilibrate the system density and production dynamics in the equivalent NVT ensemble mode (constant volume and temperature). SHAKE was applied to constrain all bonds connecting hydrogen atoms, enabling a 2.0 fs time step in the simulations. The particle-mesh Ewald method was used to treat long-range electrostatic interactions under periodic boundary conditions (charge grid spacing of ~1.0 Å, the fourth order of the B-spline charge interpolation; and direct sum tolerance of 10^{-5}). The cut off distance for short-range non-bonded interactions was 10 Å, with the long-range van der Waals interactions based on a uniform density approximation. To reduce the computation, non-bonded forces were calculated using a two-stage RESPA approach where the short range forces were updated every step and the long range forces were updated every two steps. Temperature was controlled using the Langevin thermostat with a coupling constant of 2.0 ps. The trajectories were saved at 50.0 ps intervals for analysis.

3.3.2 Coarse-grained MSM analysis. 33 trajectories (1000 ns each) of the DNA-ligand system were combined into one trajectory. Using VMD, all frames in which there were less than 13 atom contacts, at a distance less than 3Å, between the G-quadruplex and the ligand were separated as the unbound state (Humphrey, Dalke, & Schulten, 1996). The trajectory was then superimposed based on the nucleic backbone using MDtraj and calculations for RMSD as well as center of mass of the ligand heavy atoms were performed (McGibbon et al., 2015). *k*-means clustering, performed using scikit-learn, was then used to classify the remaining frames into various states (Pedregosa et al., 2011). Clustering was performed for *k* between 2 and 30 inclusively, using the silhouette index as the metric for similarity of clusters (Pedregosa et al., 2011; Rousseeuw, 1987). It was determined that k=4 had the greatest silhouette index and the most representative frame for each cluster was determined by calculating the mean RMSD for each cluster and finding the frame with the least difference from the mean. Further validation of the clustering was performed by creating a trajectory for each of the clusters containing all of the frames in each cluster and visually confirming the similarity within each cluster. Through visual analysis of the cluster representative frames, two clusters were determined to be highly similar and were thus combined. The unbound frames were then reintroduced as a single cluster resulting in a total of four clusters, observed to be top binding pose, side binding pose, bottom binding pose, and unbound.

Count matrices were then created for lagtimes (τ) of 1, 10, 20, 30 … 1000 ns by counting the number of observed transitions between discrete states such that the count of transitions from state *i* to state *j* (c_{ij}) is the sum of the number of times each of the trajectories were observed in state *i* at time *t* and in state *j* at time $t + \tau$, for all $t \leq t_{max} - \tau$ (Prinz et al., 2011). The count matrices were symmetrized (sym_{ij}) such that $sym_{ij} = sym_{ji} = \frac{c_{ij}+c_{ji}}{2}$ and then row-normalized ($norm_{ij}$) such that $norm_{ij} = \frac{sym_{ij}}{\sum_{j=1}^{j=n} sym_{ij}}$. For the purpose of determining the lag time at which the model has converged, the implied timescale of each cluster was calculated for all lagtimes and plotted (Figure 6). The implied timescale of the first cluster is not included in the plot as the eigenvalue is always 1 and thus contributes no information (Noé et al., 2007). Further validation that the model had been converged was performed through the Chapman-Kolmogorov test (Figure 7) (Prinz et al., 2011). A network model (Figure 8) was then generated based on the count matrix at a lag time of 250 ns with the cutoff for a directed edge in the network being set at 300 transitions (Csardi & Nepusz, 2006). Thereafter, the mean first passage times (F_{if}) at a lag time of 250 ns and the standard deviations from lag time 250 ns to 750 ns were calculated according to the formula

$F_{if} = \tau + \sum_{j \neq f} P_{ij} F_{jf}$, with the boundary condition $F_{ff} = 0$, where τ is the lag time used to construct the transition matrix $P(\tau)$.

3.4 Results

The distribution of DBD1 over the course of all 33 binding simulations, represented by a single atom, was calculated. From the top view and side view, almost every surface of the DNA G-quadruplex was sampled by the ligand, suggesting that a good position sampling has been achieved by our simulation protocol. The Root Mean Square Deviations (RMSD) of both the DNA backbone and ligand were calculated for all the runs of the free ligand binding simulations. Atom contacts between the DNA structure and the drug molecule were calculated using an atom-to-atom distance cutoff of 3.0 Å. The flat RMSDs and atom contacts after 250 ns were observed in the most of trajectories, indicating the convergence of the binding simulations. The last snapshots can be found in Appendix F and of the 33 trajectories, final binding poses at 1001 ns were 23 top binding, 6 bottom binding, and 4 side binding.

3.4.1 Clustering. Clustering analysis was performed as discussed in the methods section in order to identify three major binding modes (top stacking, groove binding, and bottom stacking). The first cluster, the most abundant, is a top binding pose that consists of 55.9% of the simulation. DBD1, resting above the first G-tetrad layer (G4, G8, G13, G17), exhibits intercalation at the 5'-end of the G-quadruplex. The second cluster is a side binding pose that consists of 15.6% of the simulation and only exhibits minor intercalation between A21 and G23. The third cluster, consisting of 12.1% of the simulation, is a bottom binding pose. Altogether, they encompassed 83.6% of all the trajectories. The remainder of the simulation consisted of the unbound state.

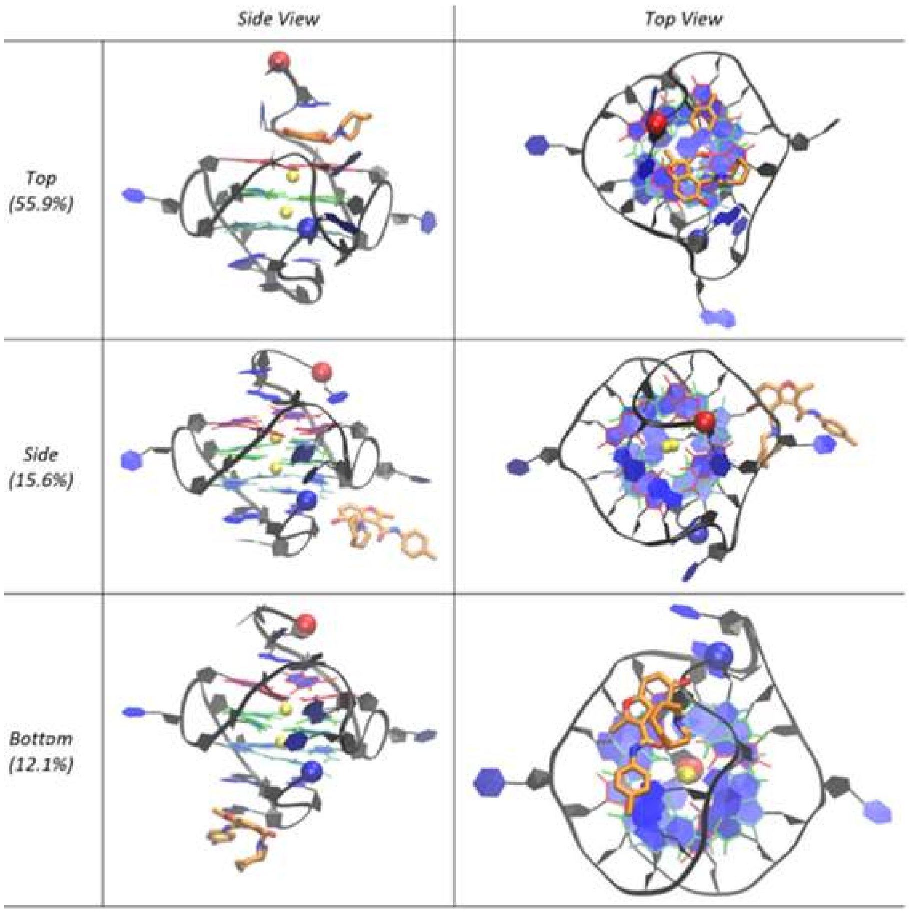


Figure 10. Clustering results. Representative structure of populated bound clusters from the *k*-means clustering analysis. The three G-tetrad layers in the G-quadruplex are in red, green, and light blue for the top, middle and bottom layers, respectively. The ligand (DBD1) is in orange and K^+ cations are represented by yellow balls. The unbound cluster was not shown (16.4%).

3.4.2 MM-PBSA. MM-PBSA binding energy calculations were conducted for the three ligand binding modes in order to determine the relative stability of the three major binding modes. The binding energy calculations indicated that the most energetically favorable binding pose was the top binding mode (-42.9 ± 4.5 kcal/mol) followed by both the bottom (-16.9 ± 1.8 kcal/mol) and side (-16.9 ± 2.6 kcal/mol) binding modes which had similar binding energies. van der Waals forces play a major role in the stability of the binding of DBD1 to Pu24 as can be seen when comparing the van der Waals forces of the top binding mode (-52.0 ± 0.3 kcal/mol) to that of the bottom (-14.7 ± 1.0 kcal/mol) or side (-14.3 ± 0.5 kcal/mol) binding modes. The change in binding energy ($\Delta\Delta E_{TOT}$) between the top binding mode and the other two binding modes is less than the difference in van der Waals energy, indicating that the van der Waals interactions make up the majority of the total MM-PBSA binding energy for all three binding poses. We see that the difference in PBTOT, PB solvation and gas phase energy, also plays a lesser but non-negligible role in the difference between the binding energies of top (-25.9 ± 0.4 kcal/mol), bottom (-4.5 ± 0.8 kcal/mol), and side (-4.5 ± 0.5 kcal/mol) binding modes.

Table 1

MM-PBSA binding energies.

Binding energies (kcal/mol) of DBD1 in the Top, Bottom, and Side binding modes.

Position	**ΔE_{VDW}**[a]	**ΔE_{SUR}**[b]	**ΔE_{PBELE}**[c]	**ΔE_{PBTOT}**[d]	**ΔConf**[e]	**ΔE_{TOT}**[f]	**$\Delta\Delta E_{TOT}$**[g]
Top	-52.0±0.3	20.3±0.3	5.9±0.4	-25.9±0.4	-16.9±4.6	-42.9±4.5	0
Bottom	-14.7±1.0	8.3± 1.2	1.9±1.0	-4.5±0.8	-12.4±1.9	-16.9±1.8	26.0
Side	-14.3±0.5	8.4±1.9	1.4±0.2	-4.5±0.5	-12.4±2.7	-16.9±2.6	26.0

[a] Gas phase van der Waals energy (VDW)
[b] Nonpolar solvation (SUR=PBSUR+PBDIS)
[c] Solvation and gas phase electrostatic energy (PBELE=PBCAL + ELE)
[d] PB Solvation and gas phase energy (PBTOT=VDW+SUR+PBELE)
[e] Conformation energy change upon complex formation (Conf)
[f] Total binding energy in water (PBTOT + Conf)
[g] Relative binding energy

3.4.3 MSM results. The clustering identified four macrostates (unbound, top, side, and bottom binding) and MSM analysis was performed on those states using transition path theory as mentioned in the methods section to obtain binding pathway information. Identification and verification of the optimal lagtime were performed using the implied timescales and Chapman-Kolmogorov test as discussed in the methods section (Figures 6-7). A network model with the optimal lagtime (250 ns) was presented with the transition counts (Figure 8): the approximate ratios of the interstate fluxes were 1:3 for unbound to top binding, 4:3 for unbound to side binding, 1:1 for unbound to bottom binding, 1:3 for side binding to top binding, and 1:2 for bottom binding to side binding. To simplify interpretation, the mean first passage time between each of the two connected states, connections being defined as any two states that had at least 1 transition, was calculated. The transitioning of the other states towards the top binding mode, the most thermodynamically stable state, was analyzed and presented in a reorganized MSM of DBD1 binding to the Pu24 G-quadruplex (Figure 11). Figure 11 shows organization of the states from top to bottom begins with unbound on top and then from least abundant to most abundant (abundance is displayed in parentheses). Overall transition times for each given path are organized from fastest (left) to slowest (right). It can be clearly seen that the transition from the unbound state directly to the final binding pose is the fastest while transitions involving transition states are significantly slower. Transition to the bottom transition pose requires transition to the side transition pose in order to reach the final binding pose. Interestingly, three distinct binding pathways to the most stable binding mode (top binding) were obtained. The first major binding pathway was the direct transition from unbound to the final top binding state. The second and third binding pathways involve additional transitions through the side binding transition state and both the bottom binding transition state and the side binding transition state respectively. We can see from our results that the ligand has multiple pathways, some more favored than others, to reach the final binding pose and these pathways are observable in the original trajectories, further supporting our results.

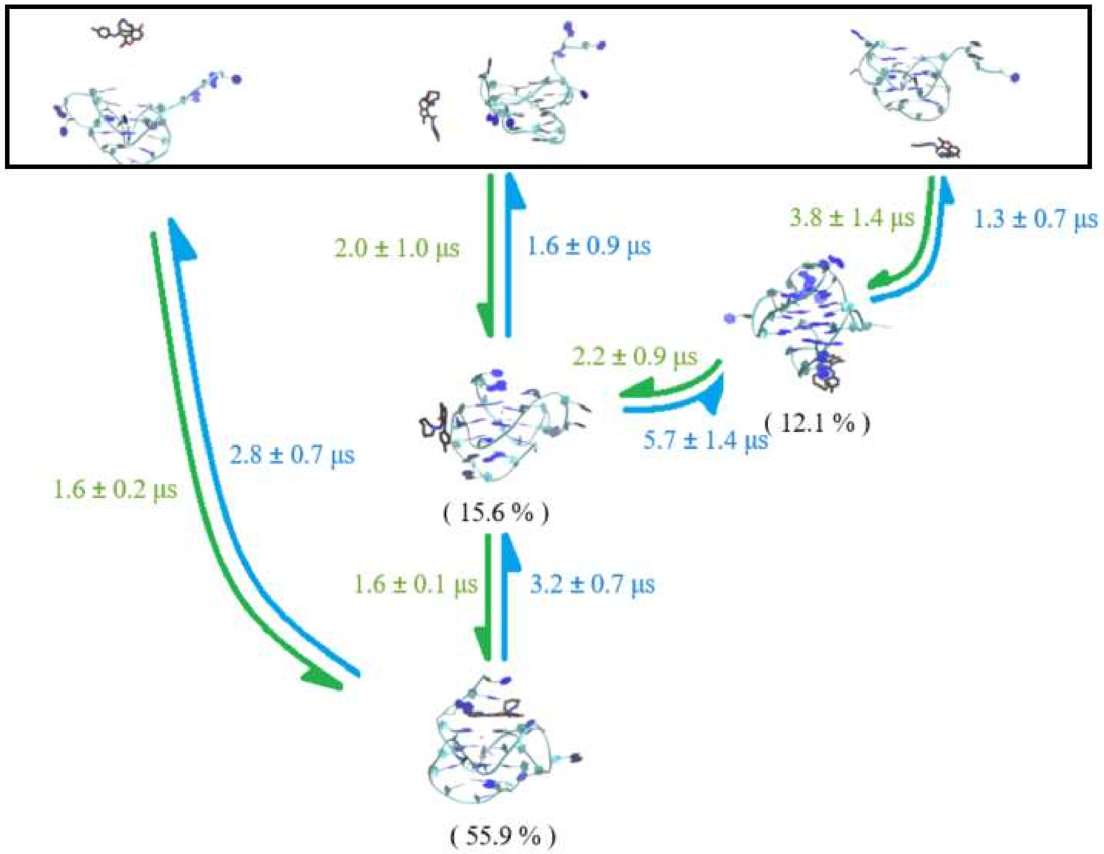


Figure 11. MSM of DBD1/Pu24 complex. The top row consists of representative structures of the unbound state. The middle row consists of the two intermediate states, side and bottom. The bottom row consists of DBD1 binding to the top site of the G-quadruplex. The mean first passage times between the four states (unbound, bottom, side, and top) are annotated in the same color as the arrow directing the transition. DBD1 and the Pu24TT G-quadruplex are colored black and blue/cyan respectively.

3.4.4 Simulation results. The 33 simulated trajectories can be further classified into different binding pathways. First, there are 16 trajectories that exhibit the transition shown from unbound directly to the top binding state (Appendix G). Second, nine trajectories show the ligand going from the unbound state to the side transition state to the final top binding state (Appendix G). Third, three trajectories indicate a transition from either top or bottom to the side binding state (Appendix G). Fourth, there are two trajectories that indicate the transition from unbound to bottom, which combined with the aforementioned transitions shows the possibility of an unbound to bottom to side to top transition if the trajectories were to be extended (Appendix G). Fifth, some of the reverse pathways can be observed such as 3 trajectories exhibiting the transition from side back to bottom binding (Appendix G). Clearly, these observed pathways support our MSM (Figure 11).

A representative trajectory for the first three pathways was chosen for further characterization using some order parameters (Figures 12-14). We measured hydrogen bonds, center-to-center distance (D), drug-base dihedral angle, receptor and ligand RMSD and MM-PBSA binding energy (ΔE). The top stacking mode was the most energetically favorable and stable structure according to the RMSD and MM-PBSA binding energy.

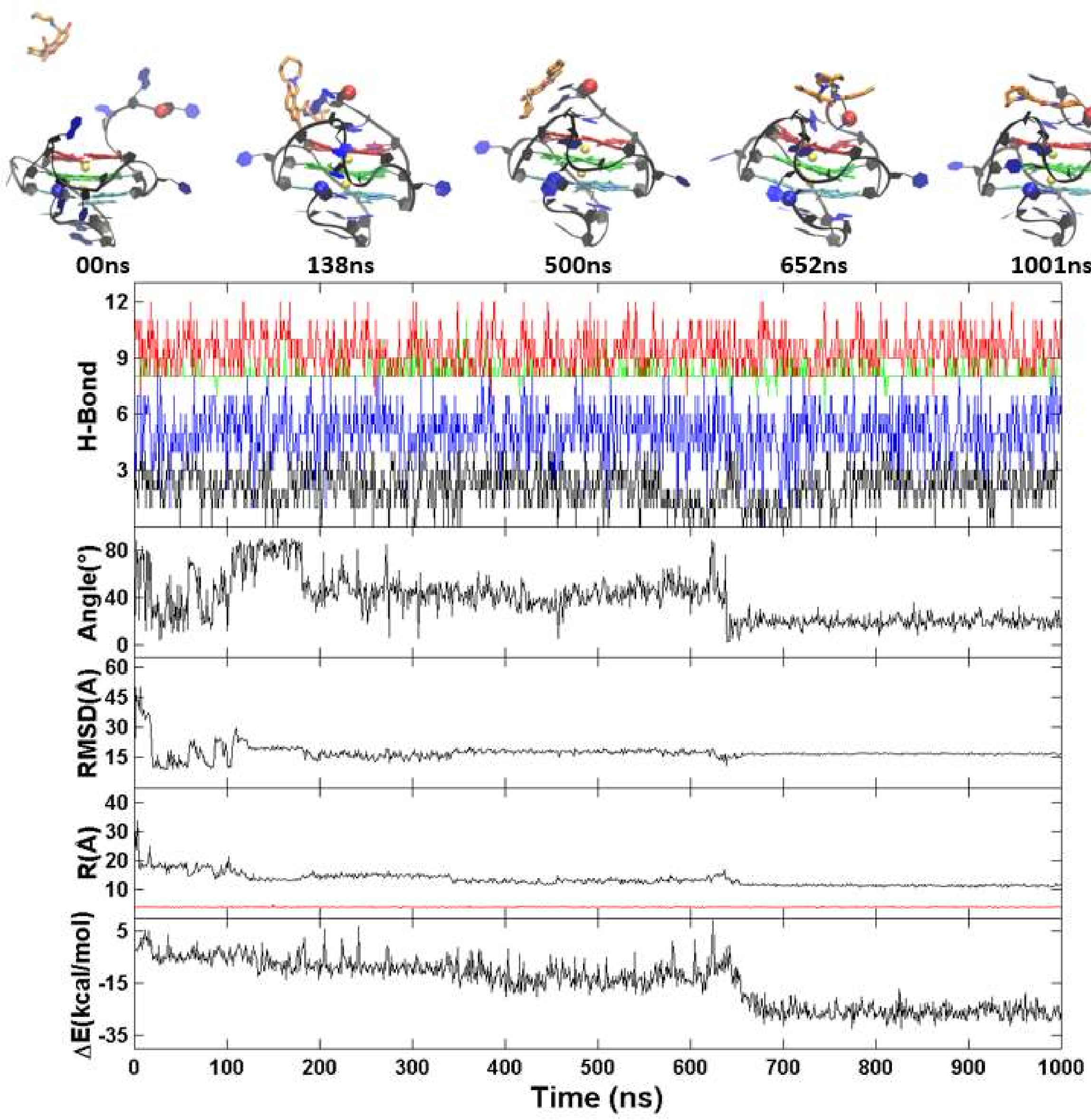


Figure 12. Order parameters of top binding representative trajectory. Results are calculated from a representative trajectory of the primary binding pathway of DBD1 to the top position of the G-quadruplex. Top-bottom: Representative structures with time annotation. 5' and 3' are indicated by a red and blue ball, respectively. K^+ ions are represented in yellow. Hydrogen bonds in the first (red), second (green), third (blue) G-tetrad and fourth (black) of G-triad layer of quadruplex (H-bond), drug-base dihedral angle, ligand RMSD, center-to-center distance (R/black) and K^+-K^+ distance (R/red) and MM-PBSA binding energy (ΔE).

Figure 12 shows unbound DBD nearly reaching the top binding position at 138 ns and transitioning fully into this position by 652 ns. Hydrogen bonding remains relatively stable throughout the simulation with about 10, 9, 6, and 3 hydrogen bonds in the first, second, third G-tetrad, and fourth triad layers of the G-quadruplex, respectively, suggesting little change in the G-quadruplex scaffold. The drug-base dihedral begins at ~80 degrees, decreases to ~40 degrees at 200 ns, and stabilizes at 20 degrees at 650 ns and throughout the remaining simulation. This highlights DBD1's intercalation between the top G-tetrad layer and the 5'-end loop. The DBD1 ligand RMSD stabilizes at ~20 Å by 150 ns and remains stable throughout the simulation. Center-to-center potassium ion distancing between the ligand and the G-quadruplex stabilizes by 100 ns and the potassium ions distances remains stable throughout the simulation. MM-PBSA binding energy stabilizes at approximately -25 kcal/mol by 650 ns after DBD1 reaches the top binding site.

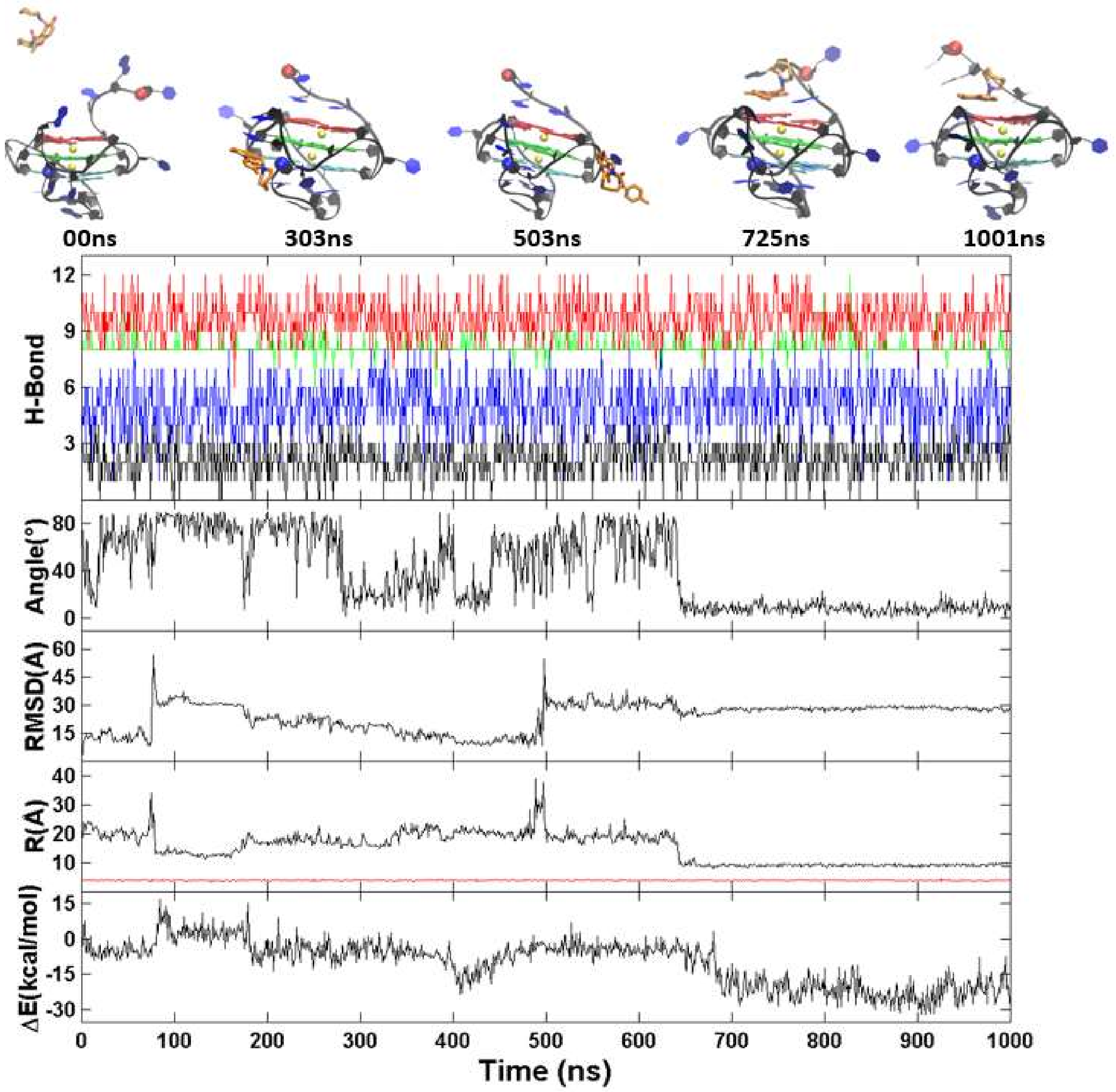


Figure 13. Order parameters of side to top representative trajectory. Results are calculated from a representative trajectory of the primary binding pathway of DBD1 to the top position of the G-quadruplex. Top-bottom: Representative structures with time annotation. 5' and 3' are indicated by a red and blue ball, respectively. K^+ ions are represented in yellow. Hydrogen bonds in the first (red), second (green), third (blue) and last (black) of G-tetrad layer of quadruplex (H-bond), drug-base dihedral angle, ligand RMSD, center-to-center distance (R/black) and K^+-K^+ distance (R/red) and MM-PBSA binding energy (ΔE).

Figure 13 shows unbound DBD1 that transitions to the side-binding site by 303 ns and to another side-binding site at 503 ns before transitioning to the top-binding position at 725 ns. The hydrogen binding analysis here shows little change in the G-quadruplex scaffold, as in Figure 12. The dihedral angle initially averages 80 degrees, decreases to ~20 degrees at 300ns, increases again to ~80 degrees between 300 ns and 650 ns and stabilizes at 10 degrees at 650 ns and throughout the remaining simulation. The DBD1 RMSD begins at 15 Å, increases sharply to ~30 Å at 100 ns, decreases slowly until 500 ns before spiking to ~30 Å again and then stabilizes with less fluctuation at 650 ns until the end of the simulation. Center-to-center distance stabilizes by 650 ns, while the distance between the two K^+ ions remains stable for the entire simulation. MM-PBSA energy stabilizes at approximately -25 kcal/mol by 650 ns when the ligand has reached the stable top binding position.

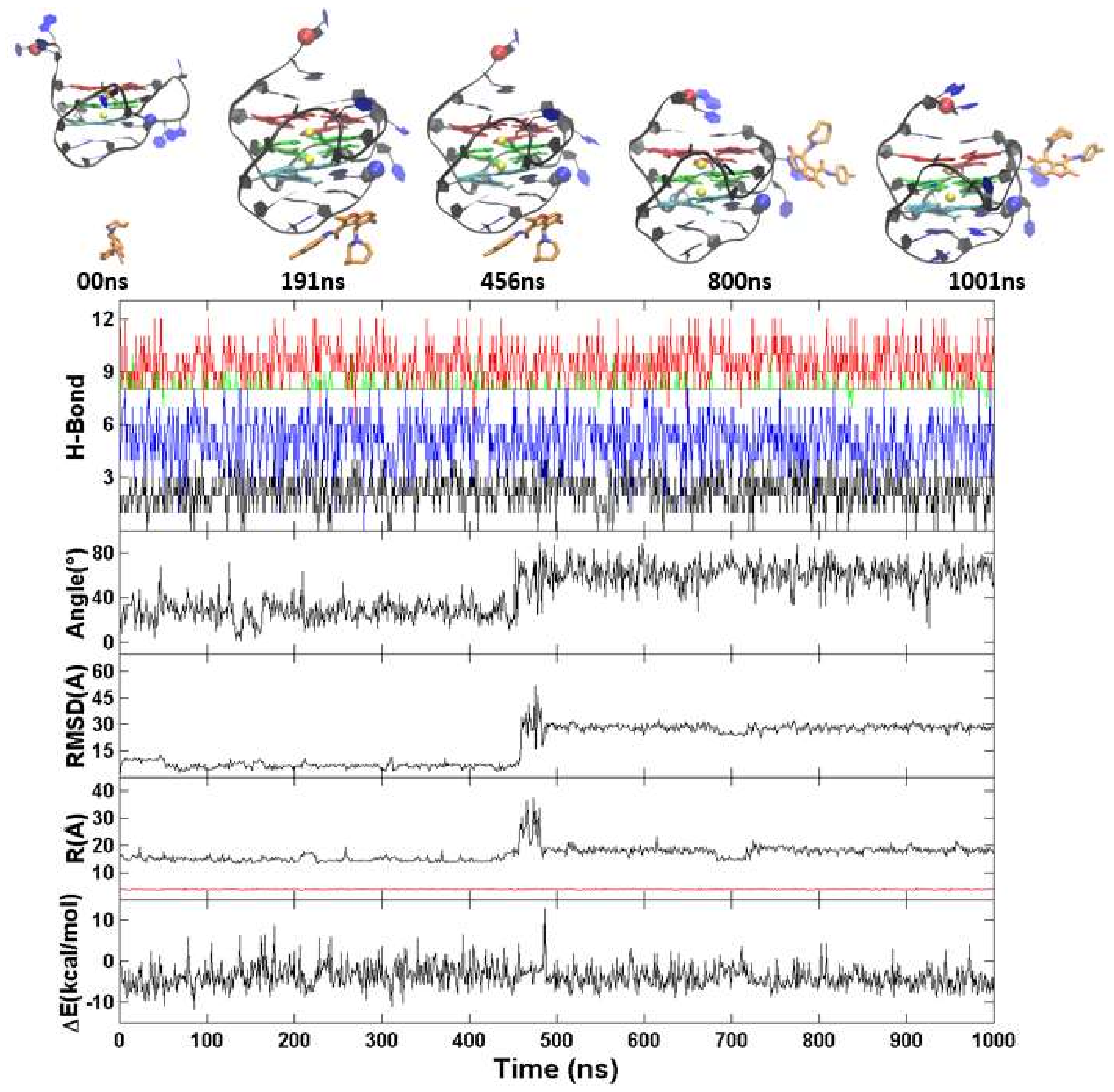


Figure 14. Order parameters of bottom to side representative trajectory. Results are calculated from a representative trajectory of the primary binding pathway of DBD1 to the top position of the G-quadruplex. Top-bottom: Representative structures with time annotation. 5' and 3' are indicated by a red and blue ball, respectively. K^+ ions are represented in yellow. Hydrogen bonds in the first (red), second (green), third (blue) and last (black) of G-tetrad layer of quadruplex (H-bond), drug-base dihedral angle, ligand RMSD, center-to-center distance (R/black) and K^+-K^+ distance (R/red) and MM-PBSA binding energy (ΔE).

Figure 14 shows unbound DBD1 before it reaches the bottom-binding position at 191 ns and transitioning to the side-binding position at 800 ns. The hydrogen binding analysis here shows little change in the G-quadruplex scaffold, as seen in Figures 12 and 13. The dihedral angle initially averages ~30 degrees, increasing and stabilizing to ~70 degrees at 450 ns throughout the remaining simulation. The DBD1 RMSD starts at 15 Å before increasing sharply to 30 Å at 450 ns and stabilizes for the remaining simulation. Center-to-center distance is stable throughout the entire simulation except for a small fluctuation at ~450 ns, while the distance between the two potassium ions remains stable throughout the simulation. MM-PBSA energy fluctuates between 0 kcal/mol and -10 kcal/mol for the entire simulation.

Trends in Figures 12-14 showed a lack of change in H-bonding within the G-quadruplex structure scaffolding; the G-quadruplex structure conformations were not significantly altered by ligand-binding. The center-to-center potassium ion distancing in the G-quadruplexes stays relatively stable in all three simulations, suggesting that the core structures do not undergo significant conformational change. The nucleotide sequence ($T^{10}G^{11}A^{12}$) underwent noticeable conformational changes in all three systems, where base flipping was apparent in the ($T^{10}G^{11}A^{12}$) sequence. The top-binding position in all three trajectories has a lower dihedral angle and binding energy after DBD1 intercalates between the 5'-loop and the top G-tetrad layer; this suggests that DBD1 has geometric-favorability within the top-binding site, resting just above the first G-tetrad layer. The top binding position exhibits planarity with the G-quadruplex scaffold, which is not seen in either the bottom- or side-binding positions. This planarity allows DBD1 to intercalate here, serving an important factor in determining the most stable binding position. Our findings suggest that the most favorable binding position is top-stacking while the intermediate positions, side and bottom ultimately transition to the top-binding position of the G-quadruplex.

The pathway from the unbound state directly to the final (top) binding state is indicative of the induced fit theory while the other two pathways that selectively bind to the side or the bottom pockets suggest that the system follows the conformational selection theory. We can see from the observed transitions that the number of transitions for both the induced fit and conformational selection pathways are relatively similar in abundance, thus we propose that the binding of DBD1 to 2MGN is a mixture of the two binding theories.

3.5 Discussion

Our clustering and MM-PBSA analyses suggest that the most favorable binding state is top stacking while the intermediate states, side and bottom, shift towards the top binding position of the G-quadruplex**.** This coincides with several other studies that show the top binding mode being the most energetically favorable for certain G-quadruplex systems (Machireddy, Sullivan, & Wu, 2019; Shen, Mulholland, Zheng, & Wu, 2017). The lesser energetic favorability exhibited by the side and bottom binding poses suggests that they are intermediate states. The side binding pose is not experimentally observed in either the 2MGN or 5W77 structure but has been observed in duplex structures (Machireddy et al., 2019). Major contributors to the binding energy were van der Waals forces, which had -52.0 ± 0.3 kcal/mol for the top binding mode, -14.7 ± 1.0 kcal/mol for the bottom binding mode, and -14.3 ± 0.5 kcal/mol for the side binding mode, and PBTOT, which had -25.9 ± 0.4 kcal/mol for the top binding mode, -4.5 ± 0.8 kcal/mol for the bottom binding mode, and -4.5 ± 0.5 kcal/mol for the side binding mode. Overall binding energies of the representative trajectories also showed that the top binding pose was the most stable as the two trajectories that ended in the top binding pose exhibited MM-PBSA values of -25 kcal/mol in comparison to the bottom or side binding MM-PBSA values of -10 kcal/mol. Additionally, intercalation at the 5'-end of the G-quadruplex structure was observed for the top binding pose. Thus, we propose that the ability for the ligand to bind in a planar orientation relative to the G-quadruplex is more energetically favorable thus making the top binding mode the most favorable.

Our coarse-grained MSM procedure, used in our previous work(Mulholland et al., 2020), clusters into a handful of “macrostates” directly and skips over the experimentally unverifiable thousand “microstates”. The expected convergence time of the implied timescales should be significantly greater than that of a model with a greater number of clusters. This results in a coarser grained model that trades finer detail for greater experimental testability and easier human understanding (Pande et al., 2010). It is likely that directly clustering into “macrostates” still maintains the integrity of the MSM as verification through the Chapman-Kolmogorov test (Figure 7) indicates that the model closely resembles the observed simulation data.

The interstate fluxes (Figure 8) indicate that the favored transition pathway is from unbound to top binding, though the pathways from unbound to side binding to top binding and unbound to bottom binding to side binding to top binding play a lesser but non-negligible role. This is supported by the mean-first passage time calculations which indicate that the unbound directly to top binding pathway is the fastest (1.6 ± 0.2 μs) while the other two pathways are several microseconds slower. The pathway from the unbound state directly to the final (top) binding state is indicative of the induced fit theory while the other two pathways that selectively bind to the side or the bottom pockets suggest that the system follows the conformational selection theory. We can see from the observed transitions (Figure 8) that the number of transitions for both the induced fit and conformational selection pathways are relatively similar in abundance, thus we propose that the binding of DBD1 to 2MGN is a mixture of the two binding theories.

3.6 Other Studies

The coarse-grained MSM analysis was performed on 40 simulation trajectories of the TMPyP4-G4C2 RNA G-quadruplex system (Mulholland et al., 2020). To decipher the kinetics pathways, a coarse-grained MSM was constructed from the 40 binding trajectories in a similar fashion as described previously in this thesis. Consistent with the thermodynamics analysis, there were three observed kinetic binding states: top, bottom, and side (groove). Parallel binding pathways toward stable top and bottom binding states were observed for the TMPyP4-RNA G-quadruplex complex system (Figure 15). We can observe that the transition from unbound directly to the bottom/top binding state is slightly faster than the parallel pathway that involves the side transition state. The transition time of unbound to bottom is the fastest of any pathway leading to a final binding state, while the transition time of unbound to top is only slightly slower. Transitioning from unbound to the transition state and then to a final binding state is approximately two-fold the time that it takes for a direct transition from unbound to a final binding state. The bottom and top binding poses are the final binding states which collectively make up approximately 75% of the simulation. The approximate interstate flux for unbound to bottom binding was 2:1, unbound to side binding was 1:1, unbound to top binding was 3:1, side binding to top binding or bottom binding were both unidirectional.

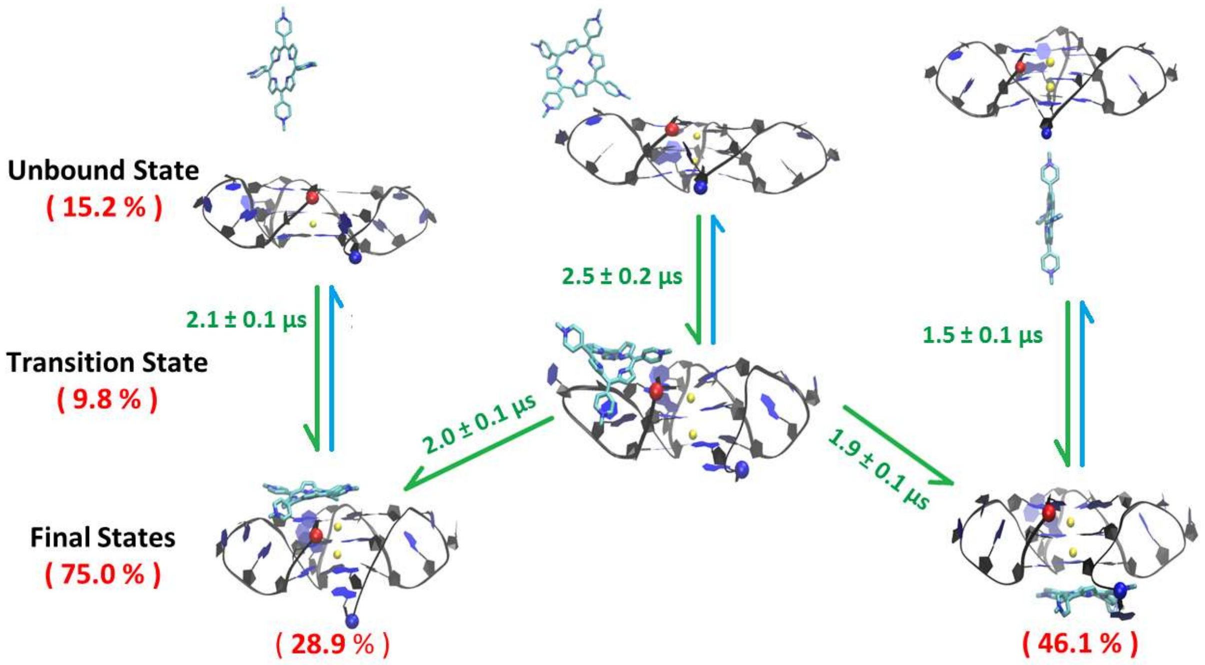


Figure 15. TMPyP4-RNA MSM. The mean first passage times between the four states (unbound, side transition, top, and bottom) of the TMPyP4-RNA G-quadruplex complex system.

In another study, four coarse-grained MSM analyses were performed in a similar method to what has been previously described on the ligand CX-5461 in complex with the MYC G-quadruplex, the c-KIT1 G-quadruplex, and human telomeric DNA, and duplex DNA (Sullivan et al., 2020). The MSM revealed multiple parallel pathways toward the most thermodynamically stable states (end stacking) in the human telomeric G4 system. For the human telomeric system, there were four major parallel pathways were observed for CX-5461: unbound to top binding, unbound to bottom binding, and unbound to side binding as an intermediate state before transitioning to either a top or bottom binding pose. The mean first passage times between the four states are shown in Figure 16 where green arrows indicate the more likely transition while blue arrows indicate a less likely transition. The top (37.8%) and bottom (20.8%) binding poses are the most thermodynamically favorable binding states and collectively make up approximately 58.5% of the simulation. Our calculated first mean passage times indicated that the pathway from unbound directly to the top binding state is slightly faster (3.3 μs) than unbound directly to the bottom binding state (4 μs) and both the transition states starts unbound and going from the side to top (1.2 μs + 3.1 μs = 4.3 μs) and side to bottom (1.2 μs + 5.7 μs = 6.9 μs) transition states. The approximate interstate flux for unbound to top binding was 1:15, unbound to side binding was 3:5, unbound to bottom binding was 2:3, side binding to top binding was unidirectional from side binding to top binding, and side binding to bottom binding was 20:1.

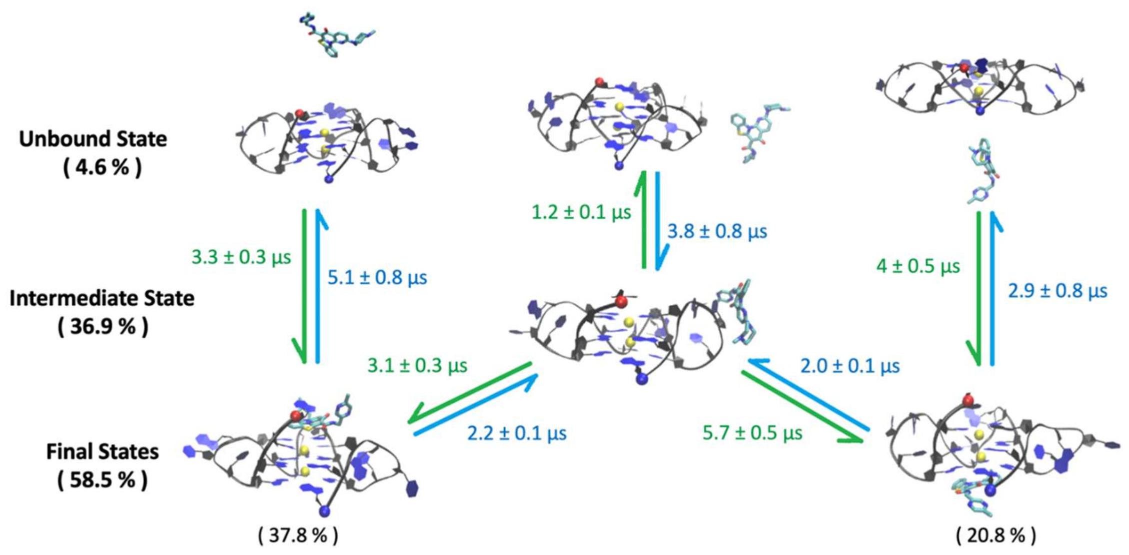


Figure 16. CX-5/human telomeric DNA MSM. The mean first passage times between the four states (unbound, side transition, top, and bottom) of the human telomeric DNA G-quadruplex and CX-5461 complex system.

The next MSM revealed multiple parallel pathways toward the most thermodynamically favorable top binding mode in the c-KIT1 system. The c-KIT1 G4 system shows three major parallel pathways leading to one thermodynamically favorable top binding state: unbound to top binding, and unbound to side binding as an intermediate state before transitioning to a top binding pose, and unbound to bottom binding before transitioning to a side binding pose and finally transitioning to a top binding pose. The mean first passage times between the three states are shown in Figure 17 where green arrows indicate the more likely transition while blue arrows indicate a less likely transition. Each of the pathways led to a thermodynamically favorable top binding state which accounted for 53.5 % of the simulation period and occurred in 1.2 μs. The unbound to side to top pathway was the next fastest totaling 2.7 μs. The slowest pathway was from the unbound state to the side and finally ending in a top binding mode which totaled 6.8 μs. Important to note is that we believe the MSM determined that the bottom binding mode is not a thermodynamically favorable state in this system because of the limited simulation period, however we expect that if the simulations were extended further, a thermodynamically favorable bottom binding site would be seen. The approximate interstate flux for unbound to top binding was 3:4, unbound to side binding was 3:4, unbound to bottom binding was 1:10, side binding to top binding was 1:30, and side binding to bottom binding was 1:50, and top binding to bottom binding was 1:4.

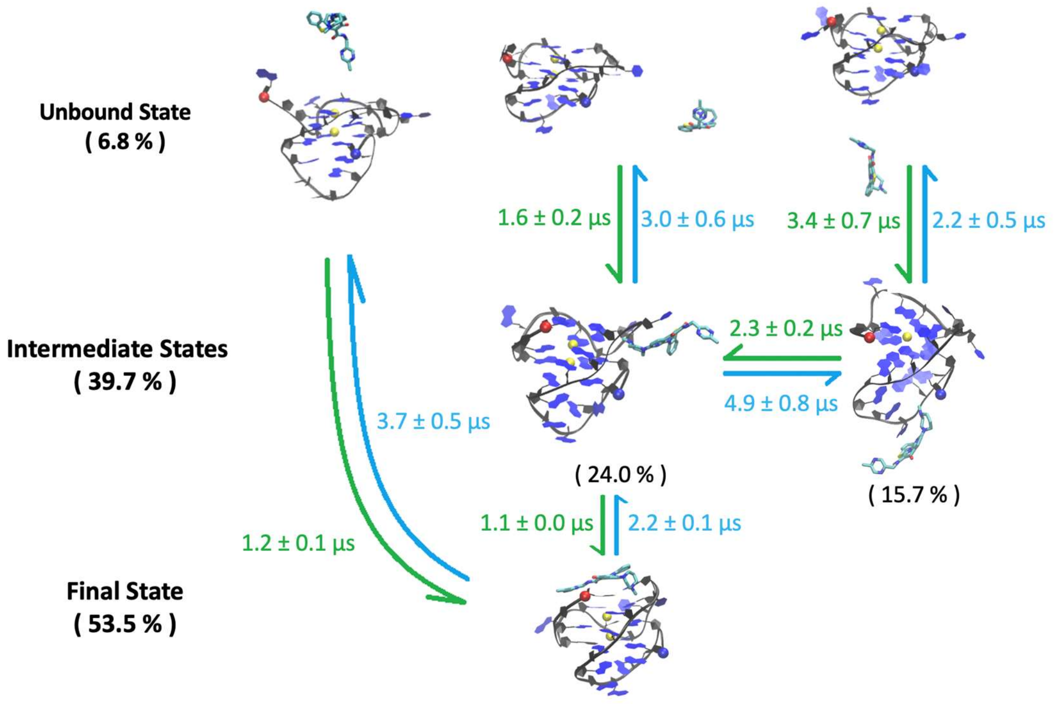


Figure 17. CX-5/c-KIT1 MSM. The mean first passage times between the four states (unbound, side transition, top, and bottom) of the c-KIT1 DNA G-quadruplex and CX-5461 complex system.

The next MSM revealed multiple parallel pathways toward the most thermodynamically stable top binding mode in the MYC system, whereas the bottom stacking mode appears to be an off-pathway intermediate. For the 2MGN system, three major parallel pathways were observed: unbound to top, unbound to bottom, and unbound to side transition and ending in a top binding mode. Unique to this system, the bottom binding pose appears to be highly unstable and likely acts as an off pathway intermediate state where CX-5461 binds to the bottom from an unbound state and once again goes back to the unbound state and follows one of the other pathways leading to the thermodynamically favorable top binding mode. The mean first passage times between these are shown in Figure 18 where green arrows indicate the more likely transition while blue arrows indicate a less likely transition. The top (59.6%) and bottom (9.5%) binding poses collectively make up approximately 69% of the simulation. The transition from unbound directly to the top binding state (1.4 μs) is slightly faster than from unbound to the top binding state through the side transition state (2.4 μs). Transition from the unbound to the bottom binding pose is significantly slower calculated to be 16.7 μs. The approximate interstate flux for unbound to top binding was 1:5, unbound to side binding was 1:1, unbound to bottom binding was 1:4, and side binding to top binding was 1:3.

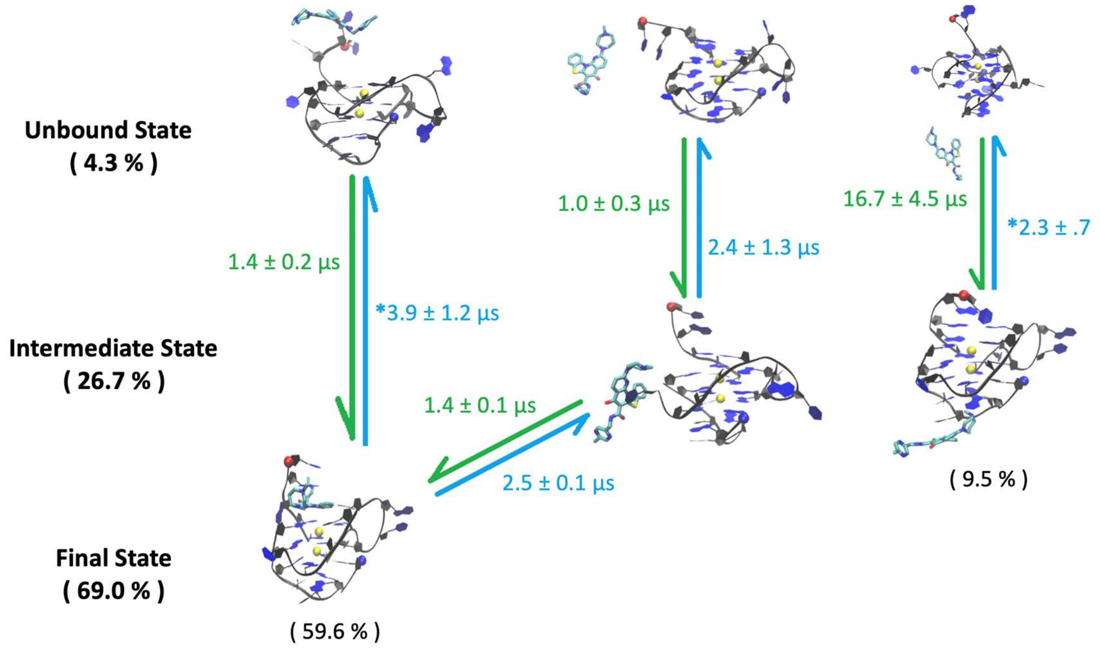


Figure 18. CX-5/MYC MSM. The mean first passage times between the four states (unbound, side transition, top, and bottom) of the MYC DNA G-quadruplex and CX-5461 complex system. Starred (*) transition times indicate mean first passage times that were calculated with relatively minimal transition data and may prove less reliable.

The MSM revealed multiple parallel pathways toward the most thermodynamically stable groove binding mode in the duplex system. The MSM plot of the duplex system and mean first passage times are presented in Figure 19 where green arrows indicate the more likely transition while blue arrows indicate a less likely transition. This plot shows the major pathways for the duplex system include one thermodynamically favorable groove biding state (25.7%) as well top and bottom states that end up in a groove binding mode. Since the groove binding mode is the only one of physiological relevance for long chromosomal DNA, pathways leading to this mode are discussed here. The pathway from unbound directly to the groove binding state is slightly faster (2.0 μs) than from unbound to the transition states (top: 4.7 μs or bottom: 5.0 μs) leading to the groove binding and is significantly more abundant.

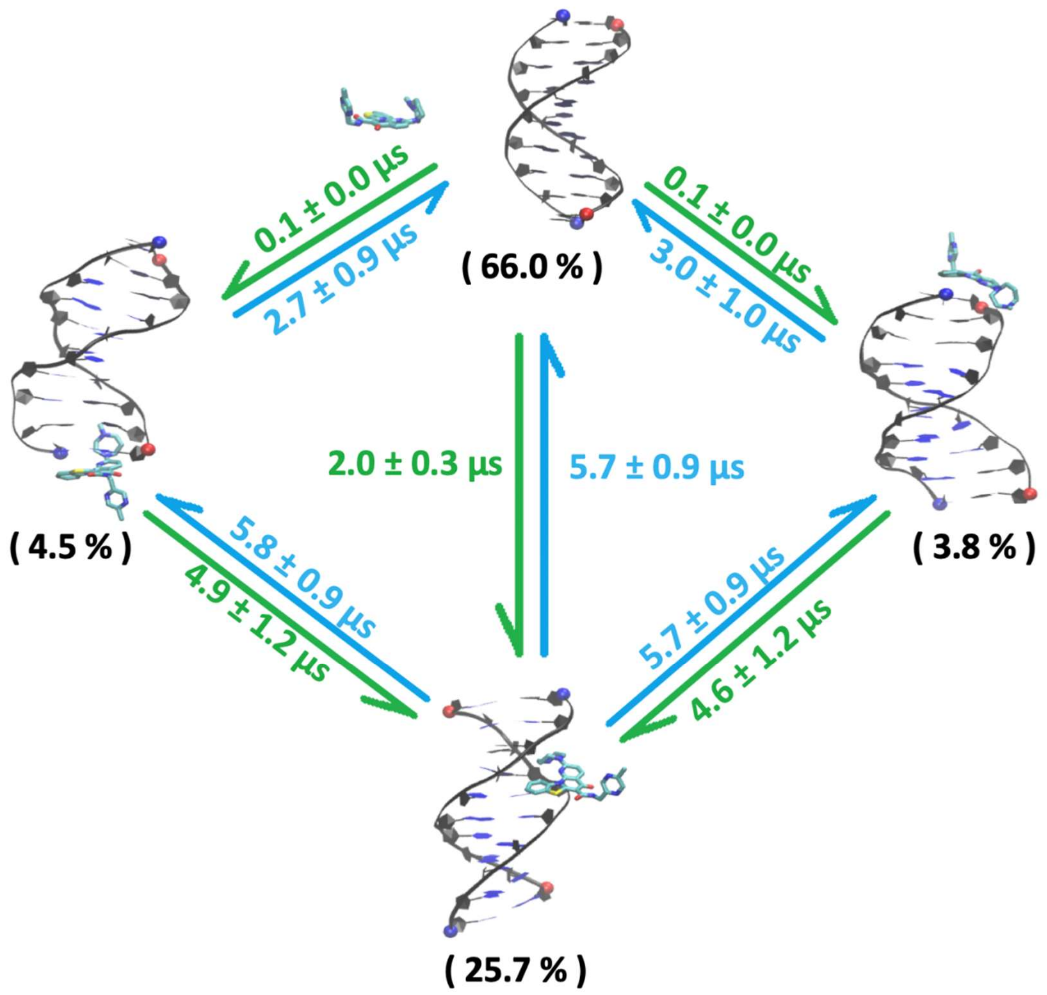


Figure 19. CX-5/DNA duplex MSM. The mean first passage times between the four states (unbound, groove binding, and top and bottom terminal binding) of the DNA duplex and CX-5461 complex system.

Chapter 4

Conclusion and Future Directions

4.1 Conclusion

The coarse-grained MSM analysis is a powerful tool that maintains the Markovian property of a traditional MSM while maintaining experimentally verifiable clusters. Traditional MSM analyses cluster into thousands of experimentally unverifiable "microstates" and are typically used for processes such as protein folding. In my experience, many common implementations such as MSMBuilder or PyEmma provide less than stellar results when attempting to create an MSM for ligand-receptor binding systems. However, the coarse-grained MSM analysis solves the issue of the experimentally unverifiable clusters and is designed specifically to work with ligand-receptor binding systems while still maintaining applicability to other biological processes. It builds upon the high resolution spatial and temporal information provided by MD simulations and enables the user to obtain invaluable kinetic information as well as binding mechanism information. The coarse-grained MSM analysis was built to be used with minimal technical knowledge and is applicable to a wide variety of systems. We have used the coarse-grained MSM analysis in several studies and expect that it can be applied to many more (Chen et al., 2020; Mulholland et al., 2020; Sullivan et al., 2020).

4.2 Future Directions

There are several improvements that I have considered for the future of the coarse-grained MSM analysis. First, the code is primarily performed through the R graphical interface, but a custom designed user interface would make the software even more accessible to users with minimal technical knowledge. Currently, some of the more complex portions of the code require the user to follow fairly complex instructions that could be minimized through the use of a custom graphical interface. Second, portions of the code are currently run through VMD which requires the user to know at least VMD technical knowledge in order to proceed. It is definitely possible to create a method that would bypass the VMD usage, but development of such a method would be fairly complex. In theory, a custom graphical interface could be designed such that the user could simply click on the structures to designate the ligand-receptor system. Third, there are definitely optimization improvements that can be performed on the code itself to decrease the run time or hardware requirements. The current software is already reasonably fast, but further improvements are definitely possible. Last, this coarse-grained MSM analysis was originally designed for analyzing ligand-receptor binding. While this analysis can be applied to other biological processes such as protein folding, I have observed that the data obtained is slightly less ideal as compared to that of our ligand-receptor binding analyses. Additionally, our analyses were typically of simulations that consisted of less than 100,000 frames in total and could be performed in a few hours on a non-specialized computer. When increasing the number of frames by various orders of magnitude, the run time of the analysis also increased by even greater orders of magnitude. Further increasing the robustness of the code could improve the range of application of the coarse-grained MSM analysis.

References

Agarwal, T., Lalwani, M. K., Kumar, S., Roy, S., Chakraborty, T. K., Sivasubbu, S., & Maiti, S. (2011). Morphological Effects of G-Quadruplex Stabilization Using a Small Molecule in Zebrafish. *Biochemistry, 53*(7), 1117-1124. doi:10.1021/bi4009352

Agrawal, P., Hatzakis, E., Guo, K., Carver, M., & Yang, D. (2013). Solution structure of the major G-quadruplex formed in the human VEGF promoter in K+: insights into loop interactions of the parallel G-quadruplexes. *Nucleic Acids Res, 41*(22), 10584-10592. doi:10.1093/nar/gkt784

Ambrus, A., Chen, D., Dai, J., Jones, R. A., & Yang, D. (2005). Solution Structure of the Biologically Relevant G-Quadruplex Element in the Human c-MYC Promoter. Implications for G-Quadruplex Stabilization. *Biochemistry, 44*(6), 2048-2058. doi:10.1021/bi048242p

Barr, L. F., Campbell, S. E., Diette, G. B., Gabrielson, E. W., Kim, S., Shim, H., & Dang, C. V. (2000). c-Myc Suppresses the Tumorigenicity of Lung Cancer Cells and Down-Regulates Vascular Endothelial Growth Factor Expression. *Cancer Research, 60*(1), 143.

Bhat, J., Mondal, S., Sengupta, P., & Chatterjee, S. (2017). In Silico Screening and Binding Characterization of Small Molecules toward a G-Quadruplex Structure Formed in the Promoter Region of c-MYC Oncogene. *ACS Omega, 2*(8), 4382-4397. doi:10.1021/acsomega.6b00531

Boddupally, P. V. L., Hahn, S., Beman, C., De, B., Brooks, T. A., Gokhale, V., & Hurley, L. H. (2012). Anticancer Activity and Cellular Repression of c-MYC by the G-Quadruplex-Stabilizing 11-Piperazinylquindoline Is Not Dependent on Direct Targeting of the G-Quadruplex in the c-MYC Promoter. *J Med Chem, 55*(13), 6076-6086. doi:10.1021/jm300282c

Buket, O., Clement, L., & DanZhou, Y. (2014). DNA G-quadruplex and its potential as anticancer drug target. *Sci China Chem, 57*(12), 1605-1614. doi:10.1007/s11426-014-5235-3

Changeux, J.-P., & Edelstein, S. (2011). Conformational selection or induced fit? 50 years of debate resolved. *F1000 biology reports, 3*, 19-19. doi:10.3410/B3-19

Che, T., Wang, Y. Q., Huang, Z. L., Tan, J. H., Huang, Z. S., & Chen, S. B. (2018). Natural Alkaloids and Heterocycles as G-Quadruplex Ligands and Potential Anticancer Agents. *Molecules, 23*(2). doi:10.3390/molecules23020493

Chen, B., Fountain, G., Sullivan, H.-J., Paradis, N., & Wu, C. (2020). *To probe the binding pathway of a disubstituted benzofuran compound (D089-0563) to c-MYC Pu24 G-quadruplex using free ligand binding simulations and Markov state model analysis*. Manuscript in preparation.

Chung, W. J., Heddi, B., Hamon, F., Teulade-Fichou, M. P., & Phan, A. T. (2014). Solution Structure of a G-quadruplex Bound to the Bisquinolinium Compound Phen-DC3. *Angewandte Chemie-International Edition, 53*(4), 999-1002. doi:10.1002/anie.201308063

Cooney, M., Czernuszewicz, G., Postel, E. H., Flint, S. J., & Hogan, M. E. (1988). Site-specific oligonucleotide binding represses transcription of the human c-myc gene in vitro. *Science, 241*(4864), 456. doi:10.1126/science.3293213

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems, 1695*(5), 1--9.

Dai, J., Carver, M., Hurley, L. H., & Yang, D. (2011). Solution Structure of a 2:1 Quindoline–c-MYC G-Quadruplex: Insights into G-Quadruplex-Interactive Small Molecule Drug Design. *Journal of the American Chemical Society, 133*(44), 17673-17680. doi:10.1021/ja205646q

Davis, J. T. (2004). G-Quartets 40 Years Later: From 5′-GMP to Molecular Biology and Supramolecular Chemistry. *Angewandte Chemie International Edition, 43*(6), 668-698. doi:10.1002/anie.200300589

Davis, T. L., Firulli, A. B., & Kinniburgh, A. J. (1989). Ribonucleoprotein and protein factors bind to an H-DNA-forming c-myc DNA element: possible regulators of the c-myc gene. *Proceedings of the National Academy of Sciences of the United States of America, 86*(24), 9682-9686. doi:10.1073/pnas.86.24.9682

Deng, N., Wickstrom, L., Cieplak, P., Lin, C., & Yang, D. (2017). Resolving the Ligand-Binding Specificity in c-MYC G-Quadruplex DNA: Absolute Binding Free Energy Calculations and SPR Experiment. *The journal of physical chemistry. B, 121*(46), 10484-10497. doi:10.1021/acs.jpcb.7b09406

Dill, K. A., Ozkan, S. B., Shell, M. S., & Weikl, T. R. (2008). The protein folding problem. *Annual review of biophysics, 37*, 289-316. doi:10.1146/annurev.biophys.37.092707.153558

Felsenstein, K. M., Saunders, L. B., Simmons, J. K., Leon, E., Calabrese, D. R., Zhang, S., . . . Schneekloth, J. S. (2011). Small Molecule Microarrays Enable the Identification of a Selective, Quadruplex-Binding Inhibitor of MYC Expression. *ACS Chemical Biology, 11*(1), 139-148. doi:10.1021/acschembio.5b00577

Harikrishna, S., Kotaru, S., & Pradeepkumar, P. I. (2017). Ligand-induced conformational preorganization of loops of c-MYC G-quadruplex DNA and its implications in structure-specific drug design. *Molecular Biosystems, 13*(8), 1458-1468. doi:10.1039/c7mb00175d

Hawksworth, D., Ravindranath, L., Chen, Y., Furusato, B., Sesterhenn, I. A., McLeod, D. G., . . . Petrovics, G. (2010). Overexpression of C-MYC oncogene in prostate cancer predicts biochemical recurrence. *Prostate Cancer And Prostatic Diseases, 13*, 311. doi:10.1038/pcan.2010.31

Hsu, S.-T. D., Varnai, P., Bugaut, A., Reszka, A. P., Neidle, S., & Balasubramanian, S. (2009). A G-Rich Sequence within the c-kit Oncogene Promoter Forms a Parallel G-Quadruplex Having Asymmetric G-Tetrad Dynamics. *Journal of the American Chemical Society, 131*(37), 13399-13409. doi:10.1021/ja904007p

Humphrey, W., Dalke, A., & Schulten, K. (1996). VMD: Visual molecular dynamics. *Journal of Molecular Graphics & Modelling, 14*(1), 33-38. doi:10.1016/0263-7855(96)00018-5

J Wu, H. (1996). The expression of c-myc protein in uterine cervical cancer: A possible prognostic indicator. *Nihon Sanka Fujinka Gakkai zasshi, 48*, 515-521.

Joung, I. S., & Cheatham, T. E., 3rd. (2008). Determination of alkali and halide monovalent ion parameters for use in explicitly solvated biomolecular simulations. *The journal of physical chemistry. B, 112*(30), 9020-9041. doi:10.1021/jp8001614

Kang, H.-J., & Park, H.-J. (2009). Novel Molecular Mechanism for Actinomycin D Activity as an Oncogenic Promoter G-Quadruplex Binder. *Biochemistry, 48*(31), 7392-7398. doi:10.1021/bi9006836

Kim, B. G., Evans, H. M., Dubins, D. N., & Chalikian, T. V. (2011). Effects of Salt on the Stability of a G-Quadruplex from the Human c-MYC Promoter. *Biochemistry, 54*(22), 3420-3430. doi:10.1021/acs.biochem.5b00097

Ma, D. L., Chan, D. S., Fu, W. C., He, H. Z., Yang, H., Yan, S. C., & Leung, C. H. (2012). Discovery of a natural product-like c-myc G-quadruplex DNA groove-binder by molecular docking. *PLoS One, 7*(8), e43278. doi:10.1371/journal.pone.0043278

Machireddy, B., Sullivan, H.-J., & Wu, C. (2019). Binding of BRACO19 to a Telomeric G-Quadruplex DNA Probed by All-Atom Molecular Dynamics Simulations with Explicit Solvent. *Molecules (Basel, Switzerland), 24*(6), 1010. doi:10.3390/molecules24061010

Magrath, I. (1990). The Pathogenesis of Burkitt's Lymphoma. In G. F. Vande Woude & G. Klein (Eds.), *Advances in Cancer Research* (Vol. 55, pp. 133-270): Academic Press.

Mathad, R. I., Hatzakis, E., Dai, J., & Yang, D. (2011). c-MYC promoter G-quadruplex formed at the 5'-end of NHE III1 element: insights into biological relevance and parallel-stranded G-quadruplex stability. *Nucleic Acids Res, 39*(20), 9023-9033. doi:10.1093/nar/gkr612

McGibbon, Robert T., Beauchamp, Kyle A., Harrigan, Matthew P., Klein, C., Swails, Jason M., Hernández, Carlos X., . . . Pande, Vijay S. (2015). MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. *Biophysical Journal, 109*(8), 1528-1532. doi:https://doi.org/10.1016/j.bpj.2015.08.015

Mulholland, K., Sullivan, H.-J., Garner, J., Cai, J., Chen, B., & Wu, C. (2020). Three-Dimensional Structure of RNA Monomeric G-Quadruplex Containing ALS and FTD Related G4C2 Repeat and Its Binding with TMPyP4 Probed by Homology Modeling based on Experimental Constraints and Molecular Dynamics Simulations. *ACS chemical neuroscience., 11*(1), 57-75. doi:10.1021/acschemneuro.9b00572

Neidle, S. (2016). Quadruplex Nucleic Acids as Novel Therapeutic Targets. *J Med Chem, 59*(13), 5987-6011. doi:10.1021/acs.jmedchem.5b01835

Noé, F., Horenko, I., Schütte, C., & Smith, J. C. (2007). Hierarchical analysis of conformational dynamics in biomolecules: Transition networks of metastable states. *The Journal of Chemical Physics, 126*(15), 155102. doi:10.1063/1.2714539

Pande, V. S., Beauchamp, K., & Bowman, G. R. (2010). Everything you wanted to know about Markov State Models but were afraid to ask. *Methods, 52*(1), 99-105. doi:10.1016/j.ymeth.2010.06.002

Pany, S. P., Bommisetti, P., Diveshkumar, K. V., & Pradeepkumar, P. I. (2016). Benzothiazole hydrazones of furylbenzamides preferentially stabilize c-MYC and c-KIT1 promoter G-quadruplex DNAs. *Org Biomol Chem, 14*(24), 5779-5793. doi:10.1039/c6ob00138f

Pedregosa, F., Ga, #235, Varoquaux, l., Gramfort, A., Michel, V., . . . Duchesnay, d. (2011). Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res., 12*, 2825-2830.

Phan, A. T., Modi, Y. S., & Patel, D. J. (2004). Propeller-Type Parallel-Stranded G-Quadruplexes in the Human c-myc Promoter. *Journal of the American Chemical Society, 126*(28), 8710-8716. doi:10.1021/ja048805k

Prinz, J.-H., Wu, H., Sarich, M., Keller, B., Senne, M., Held, M., . . . Noé, F. (2011). Markov models of molecular kinetics: Generation and validation. *The Journal of Chemical Physics, 134*(17), 174105. doi:10.1063/1.3565032

Rapp, U. R., Korn, C., Ceteci, F., Karreman, C., Luetkenhaus, K., Serafin, V., . . . Potapenko, T. (2009). MYC is a metastasis gene for non-small-cell lung cancer. *PLoS One, 4*(6), e6029. doi:10.1371/journal.pone.0006029

Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics, 20*, 53-65. doi:https://doi.org/10.1016/0377-0427(87)90125-7

Ruggiero, E., & Richter, S. N. (2018). G-quadruplexes and G-quadruplex ligands: targets and tools in antiviral therapy. *Nucleic Acids Res, 46*(7), 3270-3283. doi:10.1093/nar/gky187

Shen, Z., Mulholland, K. A., Zheng, Y., & Wu, C. (2017). Binding of anticancer drug daunomycin to a TGGGGT G-quadruplex DNA probed by all-atom molecular dynamics simulations: additional pure groove binding mode and implications on designing more selective G-quadruplex ligands. *Journal of Molecular Modeling, 23*(9), 256. doi:10.1007/s00894-017-3417-6

Smith, D. R., Myint, T., & Goh, H. S. (1993). Over-expression of the c-myc proto-oncogene in colorectal carcinoma. *British journal of cancer, 68*(2), 407-413. doi:10.1038/bjc.1993.350

Sponer, J., Cang, X., & Cheatham, T. E., 3rd. (2012). Molecular dynamics simulations of G-DNA and perspectives on the simulation of nucleic acid structures. *Methods, 57*(1), 25-39. doi:10.1016/j.ymeth.2012.04.005

Sponer, J., & Spacková, N. a. (2007). Molecular dynamics simulations and their application to four-stranded DNA. *Methods (San Diego, Calif.), 43*(4), 278-290. doi:10.1016/j.ymeth.2007.02.004

Sullivan, H.-J., Chen, B., & Wu, C. (2020). *To probe the binding of CX-5461, an anti-cancer DNA G-quadruplex stabilizer, to human telomeric, cKIT-1, and c-Myc G-quadruplexes and a DNA duplex using molecular dynamics binding simulations*. Manuscript submitted for publication.

Suntharalingam, K., White, A. J. P., & Vilar, R. (2009). Synthesis, Structural Characterization, and Quadruplex DNA Binding Studies of Platinum(II)-Terpyridine Complexes. *Inorganic Chemistry, 48*(19), 9427-9435. doi:10.1021/ic901319n

Tawani, A., Mishra, S. K., & Kumar, A. (2017). Structural insight for the recognition of G-quadruplex structure at human c-myc promoter sequence by flavonoid Quercetin. *Scientific Reports, 7*. doi:10.1038/s41598-017-03906-3

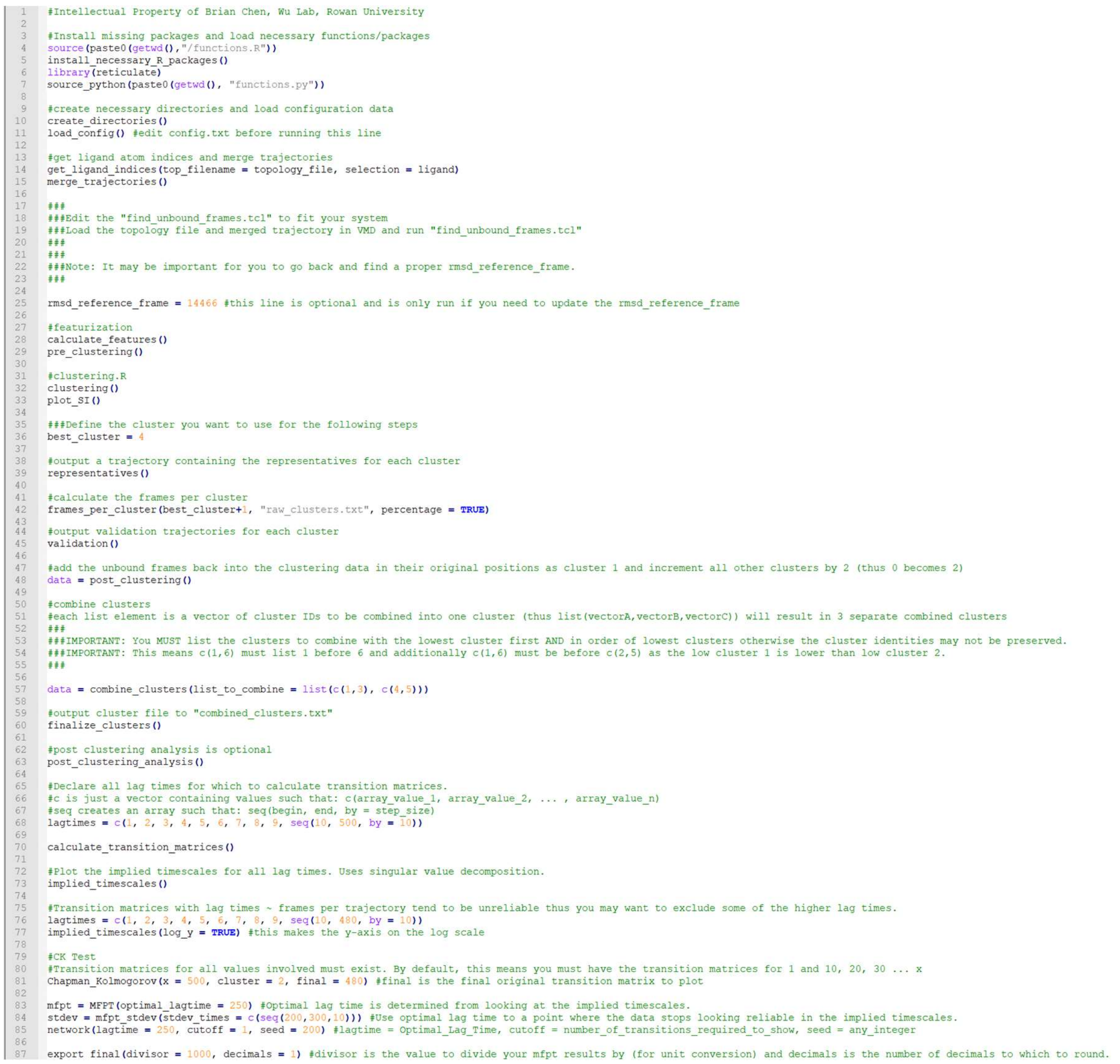
Watson, P. H., Safneck, J. R., Le, K., Dubik, D., & Shiu, R. P. C. (1993). Relationship of c-myc Amplification to Progression of Breast Cancer From In Situ to Invasive Tumor and Lymph Node Metastasis. *JNCI: Journal of the National Cancer Institute, 85*(11), 902-907. doi:10.1093/jnci/85.11.902

Zhu, H., Xiao, S., & Liang, H. (2013). Structural dynamics of human telomeric G-quadruplex loops studied by molecular dynamics simulations. *PLoS One, 8*(8), e71380. doi:10.1371/journal.pone.0071380

Appendix A

main.R

Pertinent instructions and other comments are shown in green after the '#' symbols. These are not part not the code itself and only serve to help the user.



```r
#Intellectual Property of Brian Chen, Wu Lab, Rowan University

#Install missing packages and load necessary functions/packages
source(paste0(getwd(),"/functions.R"))
install_necessary_R_packages()
library(reticulate)
source_python(paste0(getwd(), "functions.py"))

#create necessary directories and load configuration data
create_directories()
load_config() #edit config.txt before running this line

#get ligand atom indices and merge trajectories
get_ligand_indices(top_filename = topology_file, selection = ligand)
merge_trajectories()

###
###Edit the "find_unbound_frames.tcl" to fit your system
###Load the topology file and merged trajectory in VMD and run "find_unbound_frames.tcl"
###
###
###Note: It may be important for you to go back and find a proper rmsd_reference_frame.
###

rmsd_reference_frame = 14466 #this line is optional and is only run if you need to update the rmsd_reference_frame

#featurization
calculate_features()
pre_clustering()

#clustering.R
clustering()
plot_SI()

###Define the cluster you want to use for the following steps
best_cluster = 4

#output a trajectory containing the representatives for each cluster
representatives()

#calculate the frames per cluster
frames_per_cluster(best_cluster+1, "raw_clusters.txt", percentage = TRUE)

#output validation trajectories for each cluster
validation()

#add the unbound frames back into the clustering data in their original positions as cluster 1 and increment all other clusters by 2 (thus 0 becomes 2)
data = post_clustering()

#combine clusters
#each list element is a vector of cluster IDs to be combined into one cluster (thus list(vectorA,vectorB,vectorC)) will result in 3 separate combined clusters
###
###IMPORTANT: You MUST list the clusters to combine with the lowest cluster first AND in order of lowest clusters otherwise the cluster identities may not be preserved.
###IMPORTANT: This means c(1,6) must list 1 before 6 and additionally c(1,6) must be before c(2,5) as the low cluster 1 is lower than low cluster 2.
###

data = combine_clusters(list_to_combine = list(c(1,3), c(4,5)))

#output cluster file to "combined_clusters.txt"
finalize_clusters()

#post clustering analysis is optional
post_clustering_analysis()

#Declare all lag times for which to calculate transition matrices.
#c is just a vector containing values such that: c(array_value_1, array_value_2, ... , array_value_n)
#seq creates an array such that: seq(begin, end, by = step_size)
lagtimes = c(1, 2, 3, 4, 5, 6, 7, 8, 9, seq(10, 500, by = 10))

calculate_transition_matrices()

#Plot the implied timescales for all lag times. Uses singular value decomposition.
implied_timescales()

#Transition matrices with lag times ~ frames per trajectory tend to be unreliable thus you may want to exclude some of the higher lag times.
lagtimes = c(1, 2, 3, 4, 5, 6, 7, 8, 9, seq(10, 480, by = 10))
implied_timescales(log_y = TRUE) #this makes the y-axis on the log scale

#CK Test
#Transition matrices for all values involved must exist. By default, this means you must have the transition matrices for 1 and 10, 20, 30 ... x
Chapman_Kolmogorov(x = 500, cluster = 2, final = 480) #final is the final original transition matrix to plot

mfpt = MFPT(optimal_lagtime = 250) #Optimal lag time is determined from looking at the implied timescales.
stdev = mfpt_stdev(stdev_times = c(seq(200,300,10))) #Use optimal lag time to a point where the data stops looking reliable in the implied timescales.
network(lagtime = 250, cutoff = 1, seed = 200) #lagtime = Optimal_Lag_Time, cutoff = number_of_transitions_required_to_show, seed = any_integer

export_final(divisor = 1000, decimals = 1) #divisor is the value to divide your mfpt results by (for unit conversion) and decimals is the number of decimals to which to round.
```

Figure A1. Code in main.R. This figure shows all of the code present in main.R.

Appendix B

functions.R

Pertinent instructions and other comments are shown in green after the '#' symbols. These are not part not the code itself and only serve to help the user. Table B1 includes a helpful list of all functions present in functions.R.

Table B1

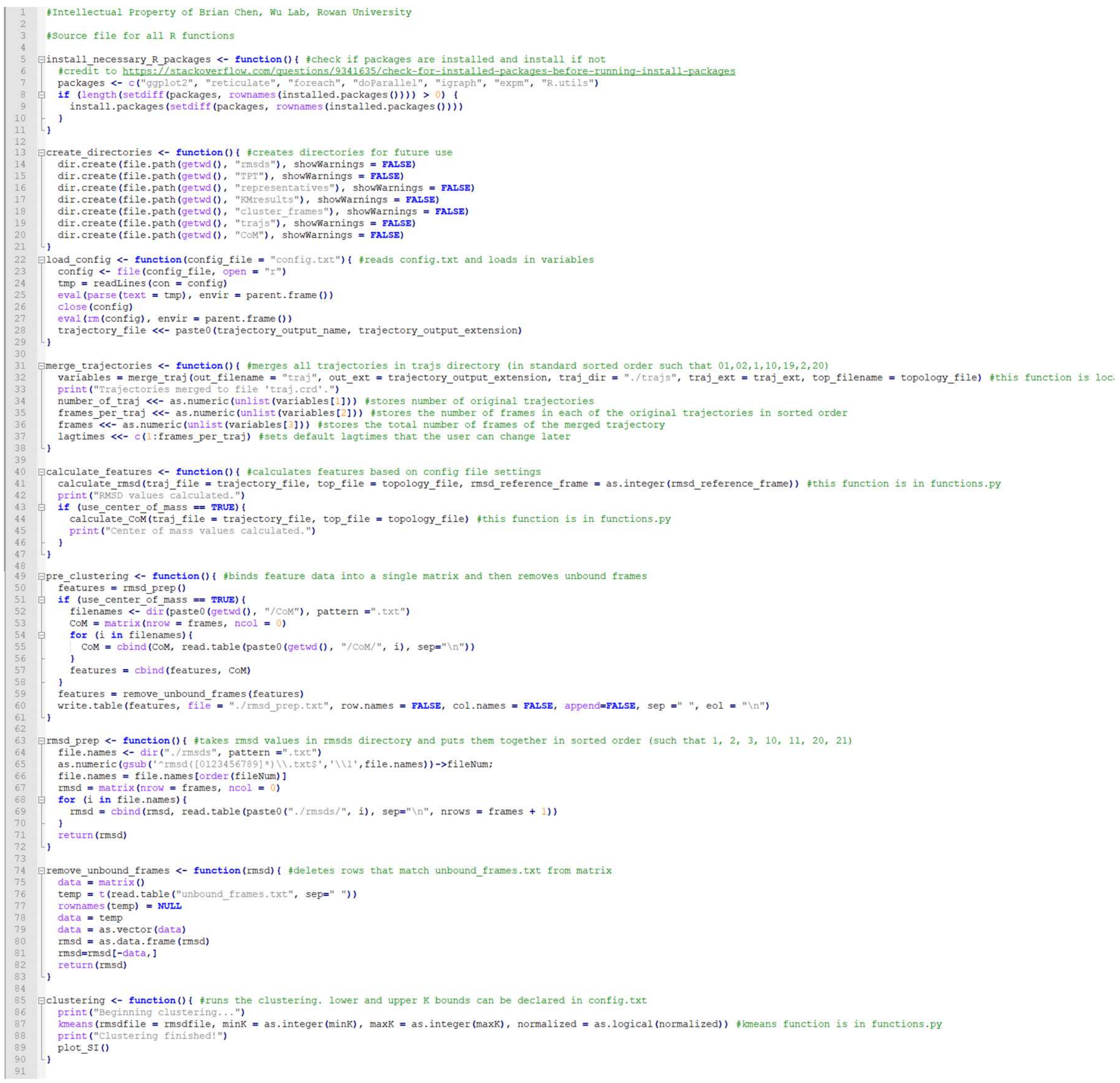
Functions present in functions.R

Functions given sequentially with its primary purpose.

Function Name	Lines	Primary Purpose
install_necessary_R_packages	5-11	Installs necessary R packages, if not installed
create_directories	13-20	Creates subdirectories in working directory
load_config	22-29	Loads configuration settings
merge_trajectories	31-38	Calls merge_traj (Appendix C) and stores results
calculate_features	40-47	Calls calculate_rmsd (Appendix C) and calculate_CoM (Appendix C)
pre_clustering	49-61	Preprocesses feature data into "rmsd_prep.txt"
rmsd_prep	63-72	Subfunction for pre_clustering
remove_unbound_frames	74-83	Removes unbound frames
clustering	85-90	Calls kmeans (Appendix C)
plot_SI	92-109	Plots the silhouette indices from *k*-means results
representatives	111-116	Calls find_representative_frames and get_representatives (Appendix C)
find_representative_frames	118-199	Finds the representative frames of each cluster
frames_per_cluster	201-215	Calculates frames per cluster
validation	217-221	Calls validation_frames and cluster_validation (Appendix C)
validation_frames	223-245	Outputs all frames of each cluster in cluster_frames directory as text files
recombine	247-272	Recombines the unbound frames with cluster data
post_clustering	274-278	Calls recombine
combine_clusters	280-317	Performs cluster_combination
finalize_clusters	319-321	Outputs final clusters to "combined_clusters.txt"
post_cluster_analysis	323-335	Reobtain frames per cluster, representatives, and validation
calculate_transition_matrices	337-343	Calls prep_data, create_tpt, and normalize_tpt

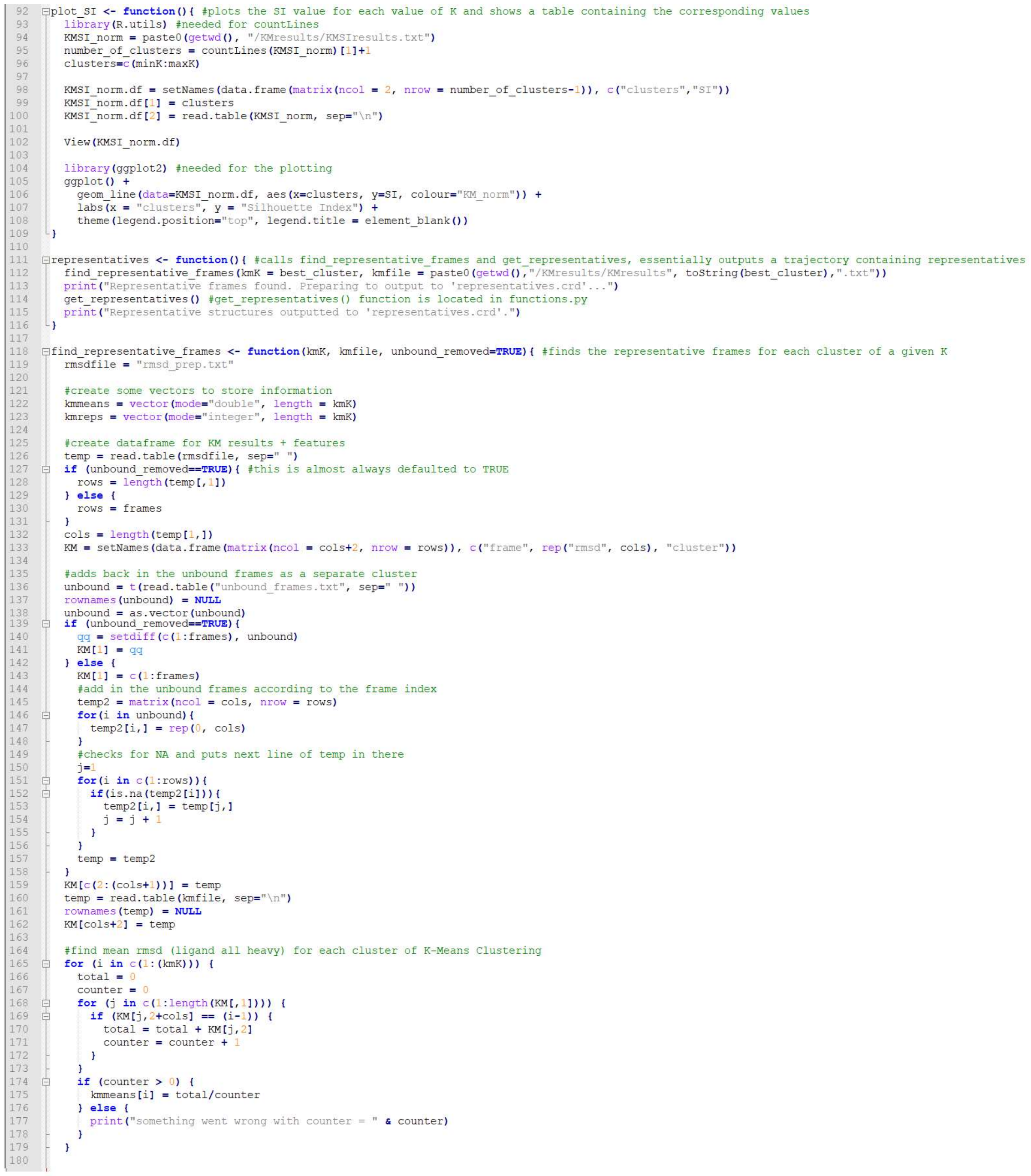
Table B1 (continued)

Function Name	Lines	Primary Purpose
prep_data	345-351	Prepares data for subsequent functions
create_tpt	353-386	Outputs transition matrices into TPT directory as text files
normalize_tpt	388-434	Outputs symmetric matrices and normalized matrices into TPT directory as text files
implied_timescales	436-500	Calculates and plots the implied timescales
Chapman-Kolmogorov	502-627	Performs the Chapman-Kolmogorov test
MFPT	629-700	Builds and solves the MFPT system of equations
hush	702-706	Suppresses unnecessary output
mfpt_stdev	708-729	Calculates standard deviations for MFPT
network	731-774	Builds a network model for transitions
export_final	776-783	Outputs MFPT and standard deviations to “final_mfpt.txt”



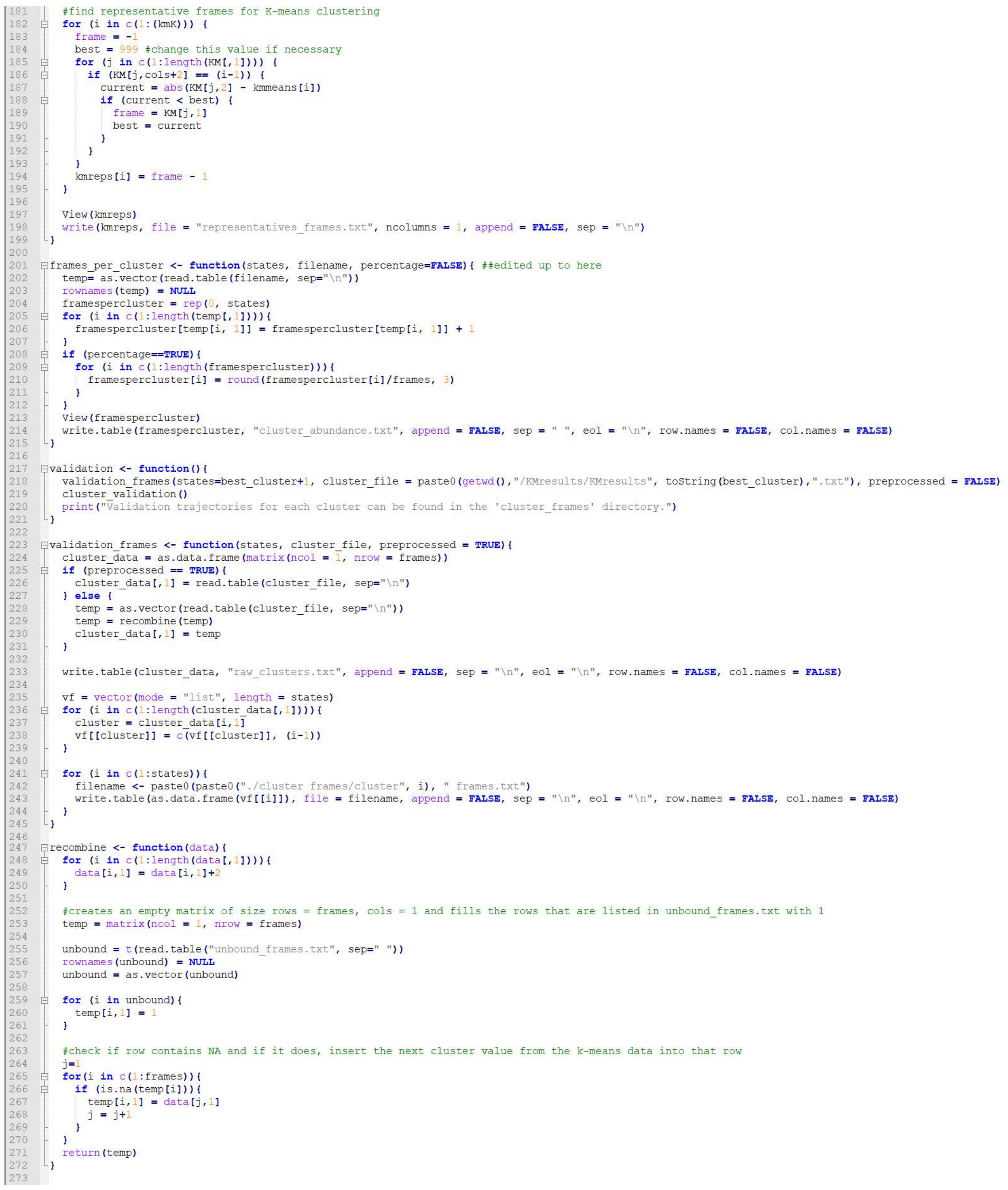
```r
#Intellectual Property of Brian Chen, Wu Lab, Rowan University

#Source file for all R functions

install_necessary_R_packages <- function(){ #check if packages are installed and install if not
  #credit to https://stackoverflow.com/questions/9341635/check-for-installed-packages-before-running-install-packages
  packages <- c("ggplot2", "reticulate", "foreach", "doParallel", "igraph", "expm", "R.utils")
  if (length(setdiff(packages, rownames(installed.packages()))) > 0) {
    install.packages(setdiff(packages, rownames(installed.packages())))
  }
}

create_directories <- function(){ #creates directories for future use
  dir.create(file.path(getwd(), "rmsds"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "TPT"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "representatives"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "KMresults"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "cluster_frames"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "trajs"), showWarnings = FALSE)
  dir.create(file.path(getwd(), "CoM"), showWarnings = FALSE)
}
load_config <- function(config_file = "config.txt"){ #reads config.txt and loads in variables
  config <- file(config_file, open = "r")
  tmp = readLines(con = config)
  eval(parse(text = tmp), envir = parent.frame())
  close(config)
  eval(rm(config), envir = parent.frame())
  trajectory_file <<- paste0(trajectory_output_name, trajectory_output_extension)
}

merge_trajectories <- function(){ #merges all trajectories in trajs directory (in standard sorted order such that 01,02,1,10,19,2,20)
  variables = merge_traj(out_filename = "traj", out_ext = trajectory_output_extension, traj_dir = "./trajs", traj_ext = traj_ext, top_filename = topology_file) #this function is loc
  print("Trajectories merged to file 'traj.crd'.")
  number_of_traj <<- as.numeric(unlist(variables[1])) #stores number of original trajectories
  frames_per_traj <<- as.numeric(unlist(variables[2])) #stores the number of frames in each of the original trajectories in sorted order
  frames <<- as.numeric(unlist(variables[3])) #stores the total number of frames of the merged trajectory
  lagtimes <<- c(1:frames_per_traj) #sets default lagtimes that the user can change later
}

calculate_features <- function(){ #calculates features based on config file settings
  calculate_rmsd(traj_file = trajectory_file, top_file = topology_file, rmsd_reference_frame = as.integer(rmsd_reference_frame)) #this function is in functions.py
  print("RMSD values calculated.")
  if (use_center_of_mass == TRUE){
    calculate_CoM(traj_file = trajectory_file, top_file = topology_file) #this function is in functions.py
    print("Center of mass values calculated.")
  }
}

pre_clustering <- function(){ #binds feature data into a single matrix and then removes unbound frames
  features = rmsd_prep()
  if (use_center_of_mass == TRUE){
    filenames <- dir(paste0(getwd(), "/CoM"), pattern =".txt")
    CoM = matrix(nrow = frames, ncol = 0)
    for (i in filenames){
      CoM = cbind(CoM, read.table(paste0(getwd(), "/CoM/", i), sep="\n"))
    }
    features = cbind(features, CoM)
  }
  features = remove_unbound_frames(features)
  write.table(features, file = "./rmsd_prep.txt", row.names = FALSE, col.names = FALSE, append=FALSE, sep =" ", eol = "\n")
}

rmsd_prep <- function(){ #takes rmsd values in rmsds directory and puts them together in sorted order (such that 1, 2, 3, 10, 11, 20, 21)
  file.names <- dir("./rmsds", pattern =".txt")
  as.numeric(gsub('^rmsd([0123456789]*)\\.txt$','\\1',file.names))->fileNum;
  file.names = file.names[order(fileNum)]
  rmsd = matrix(nrow = frames, ncol = 0)
  for (i in file.names){
    rmsd = cbind(rmsd, read.table(paste0("./rmsds/", i), sep="\n", nrows = frames + 1))
  }
  return(rmsd)
}

remove_unbound_frames <- function(rmsd){ #deletes rows that match unbound_frames.txt from matrix
  data = matrix()
  temp = t(read.table("unbound_frames.txt", sep=" "))
  rownames(temp) = NULL
  data = temp
  data = as.vector(data)
  rmsd = as.data.frame(rmsd)
  rmsd=rmsd[-data,]
  return(rmsd)
}

clustering <- function(){ #runs the clustering. lower and upper K bounds can be declared in config.txt
  print("Beginning clustering...")
  kmeans(rmsdfile = rmsdfile, minK = as.integer(minK), maxK = as.integer(maxK), normalized = as.logical(normalized)) #kmeans function is in functions.py
  print("Clustering finished!")
  plot_SI()
}

```

Figure B1. Code in functions.R. This figure shows all of the code present in functions.R.



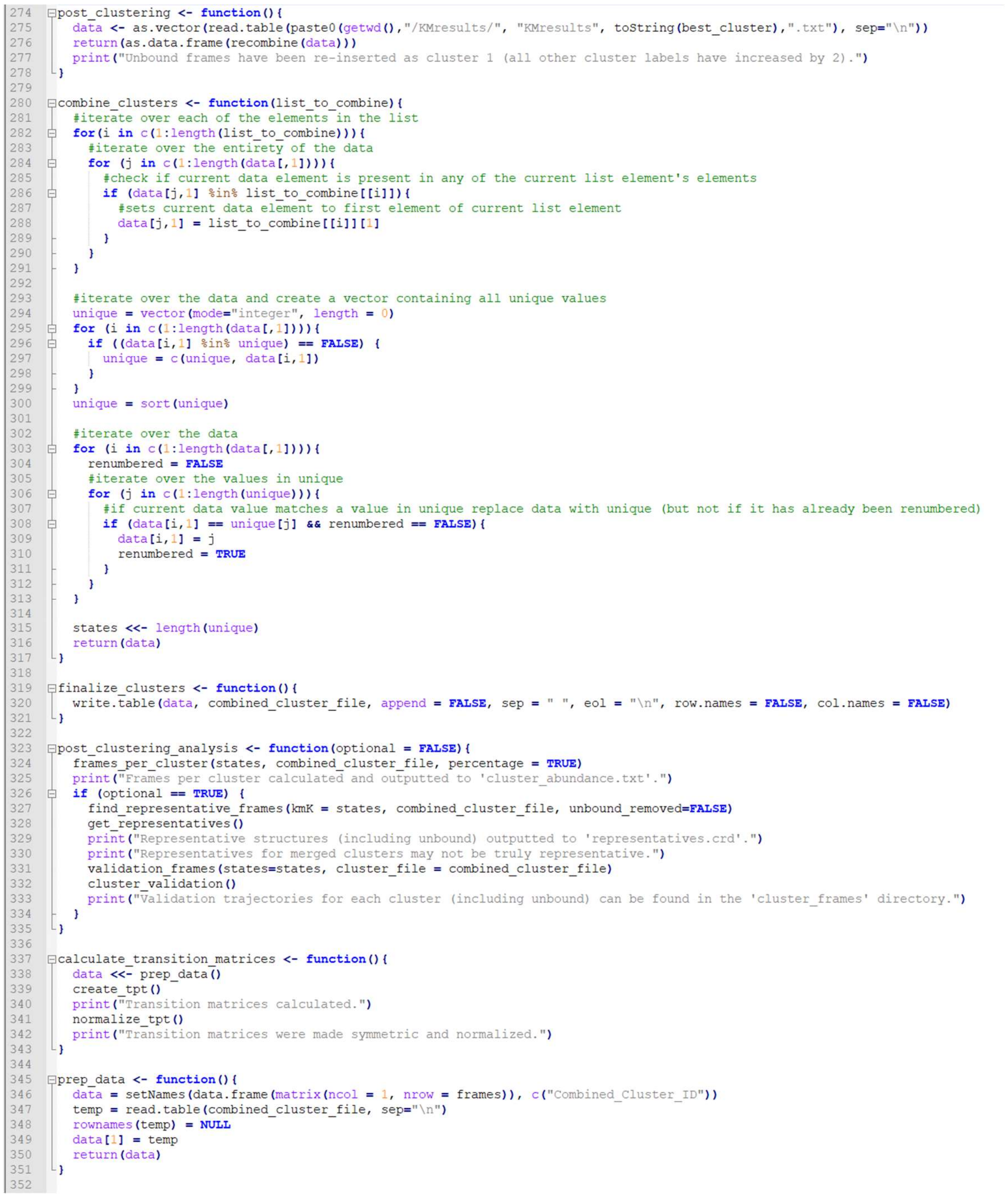
```r
plot_SI <- function(){ #plots the SI value for each value of K and shows a table containing the corresponding values
  library(R.utils) #needed for countLines
  KMSI_norm = paste0(getwd(), "/KMresults/KMSIresults.txt")
  number_of_clusters = countLines(KMSI_norm)[1]+1
  clusters=c(minK:maxK)

  KMSI_norm.df = setNames(data.frame(matrix(ncol = 2, nrow = number_of_clusters-1)), c("clusters","SI"))
  KMSI_norm.df[1] = clusters
  KMSI_norm.df[2] = read.table(KMSI_norm, sep="\n")

  View(KMSI_norm.df)

  library(ggplot2) #needed for the plotting
  ggplot() +
    geom_line(data=KMSI_norm.df, aes(x=clusters, y=SI, colour="KM_norm")) +
    labs(x = "clusters", y = "Silhouette Index") +
    theme(legend.position="top", legend.title = element_blank())
}

representatives <- function(){ #calls find_representative_frames and get_representatives, essentially outputs a trajectory containing representatives
  find_representative_frames(kmK = best_cluster, kmfile = paste0(getwd(),"/KMresults/KMresults", toString(best_cluster),".txt"))
  print("Representative frames found. Preparing to output to 'representatives.crd'...")
  get_representatives() #get_representatives() function is located in functions.py
  print("Representative structures outputted to 'representatives.crd'.")
}

find_representative_frames <- function(kmK, kmfile, unbound_removed=TRUE){ #finds the representative frames for each cluster of a given K
  rmsdfile = "rmsd_prep.txt"

  #create some vectors to store information
  kmmeans = vector(mode="double", length = kmK)
  kmreps = vector(mode="integer", length = kmK)

  #create dataframe for KM results + features
  temp = read.table(rmsdfile, sep=" ")
  if (unbound_removed==TRUE){ #this is almost always defaulted to TRUE
    rows = length(temp[,1])
  } else {
    rows = frames
  }
  cols = length(temp[1,])
  KM = setNames(data.frame(matrix(ncol = cols+2, nrow = rows)), c("frame", rep("rmsd", cols), "cluster"))

  #adds back in the unbound frames as a separate cluster
  unbound = t(read.table("unbound_frames.txt", sep=" "))
  rownames(unbound) = NULL
  unbound = as.vector(unbound)
  if (unbound_removed==TRUE){
    qq = setdiff(c(1:frames), unbound)
    KM[1] = qq
  } else {
    KM[1] = c(1:frames)
    #add in the unbound frames according to the frame index
    temp2 = matrix(ncol = cols, nrow = rows)
    for(i in unbound){
      temp2[i,] = rep(0, cols)
    }
    #checks for NA and puts next line of temp in there
    j=1
    for(i in c(1:rows)){
      if(is.na(temp2[i])){
        temp2[i,] = temp[j,]
        j = j + 1
      }
    }
    temp = temp2
  }
  KM[c(2:(cols+1))] = temp
  temp = read.table(kmfile, sep="\n")
  rownames(temp) = NULL
  KM[cols+2] = temp

  #find mean rmsd (ligand all heavy) for each cluster of K-Means Clustering
  for (i in c(1:(kmK))) {
    total = 0
    counter = 0
    for (j in c(1:length(KM[,1]))) {
      if (KM[j,2+cols] == (i-1)) {
        total = total + KM[j,2]
        counter = counter + 1
      }
    }
    if (counter > 0) {
      kmmeans[i] = total/counter
    } else {
      print("something went wrong with counter = " & counter)
    }
  }
```

Figure B1 (continued)



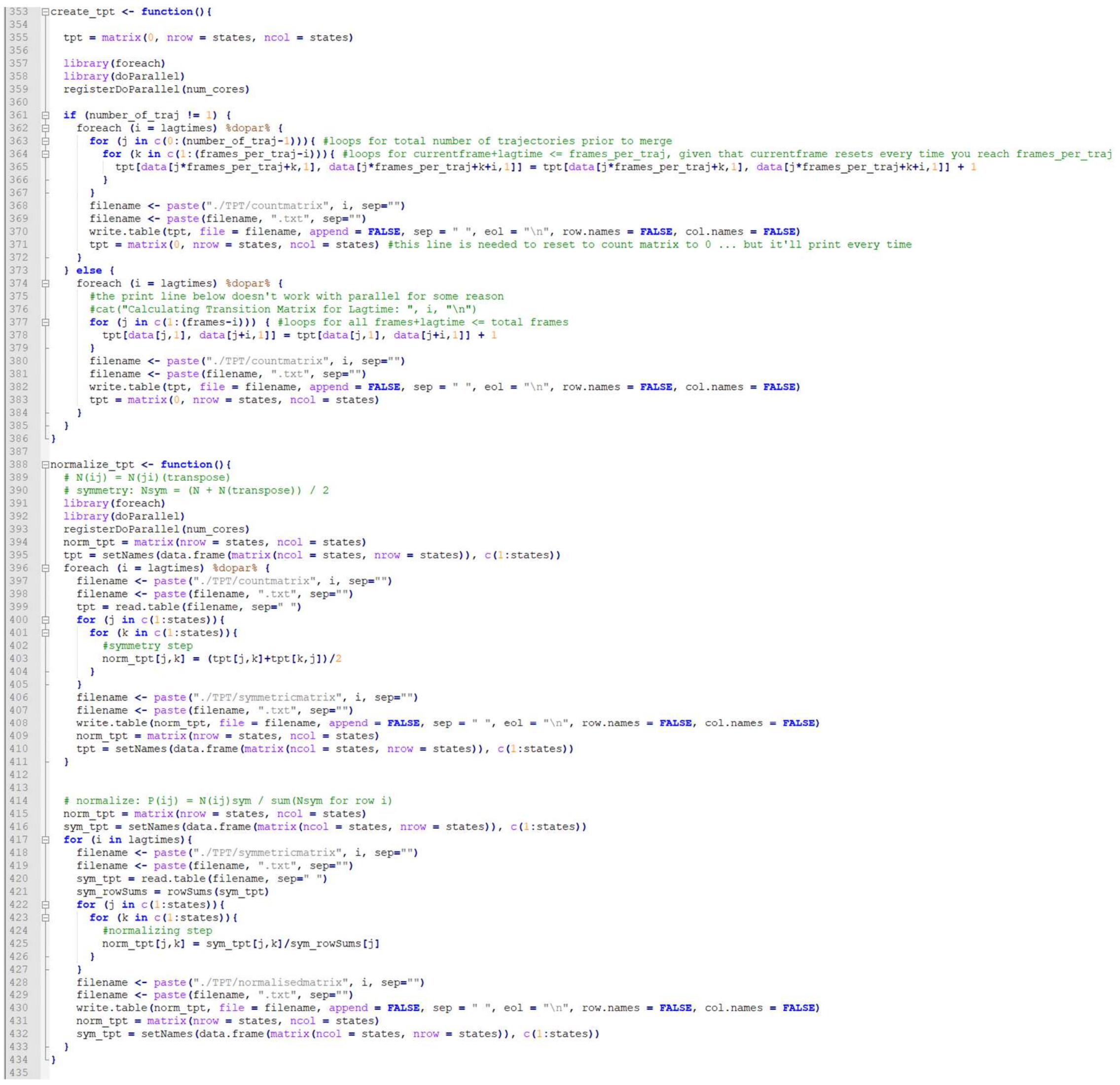
```r
  #find representative frames for K-means clustering
  for (i in c(1:(kmK))) {
    frame = -1
    best = 999 #change this value if necessary
    for (j in c(1:length(KM[,1]))) {
      if (KM[j,cols+2] == (i-1)) {
        current = abs(KM[j,2] - kmmeans[i])
        if (current < best) {
          frame = KM[j,1]
          best = current
        }
      }
    }
    kmreps[i] = frame - 1
  }

  View(kmreps)
  write(kmreps, file = "representatives_frames.txt", ncolumns = 1, append = FALSE, sep = "\n")
}

frames_per_cluster <- function(states, filename, percentage=FALSE){ ##edited up to here
  temp= as.vector(read.table(filename, sep="\n"))
  rownames(temp) = NULL
  framespercluster = rep(0, states)
  for (i in c(1:length(temp[,1]))){
    framespercluster[temp[i, 1]] = framespercluster[temp[i, 1]] + 1
  }
  if (percentage==TRUE){
    for (i in c(1:length(framespercluster))){
      framespercluster[i] = round(framespercluster[i]/frames, 3)
    }
  }
  View(framespercluster)
  write.table(framespercluster, "cluster_abundance.txt", append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
}

validation <- function(){
  validation_frames(states=best_cluster+1, cluster_file = paste0(getwd(),"/KMresults/KMresults", toString(best_cluster),".txt"), preprocessed = FALSE)
  cluster_validation()
  print("Validation trajectories for each cluster can be found in the 'cluster_frames' directory.")
}

validation_frames <- function(states, cluster_file, preprocessed = TRUE){
  cluster_data = as.data.frame(matrix(ncol = 1, nrow = frames))
  if (preprocessed == TRUE){
    cluster_data[,1] = read.table(cluster_file, sep="\n")
  } else {
    temp = as.vector(read.table(cluster_file, sep="\n"))
    temp = recombine(temp)
    cluster_data[,1] = temp
  }

  write.table(cluster_data, "raw_clusters.txt", append = FALSE, sep = "\n", eol = "\n", row.names = FALSE, col.names = FALSE)

  vf = vector(mode = "list", length = states)
  for (i in c(1:length(cluster_data[,1]))){
    cluster = cluster_data[i,1]
    vf[[cluster]] = c(vf[[cluster]], (i-1))
  }

  for (i in c(1:states)){
    filename <- paste0(paste0("./cluster_frames/cluster", i), "_frames.txt")
    write.table(as.data.frame(vf[[i]]), file = filename, append = FALSE, sep = "\n", eol = "\n", row.names = FALSE, col.names = FALSE)
  }
}

recombine <- function(data){
  for (i in c(1:length(data[,1]))){
    data[i,1] = data[i,1]+2
  }

  #creates an empty matrix of size rows = frames, cols = 1 and fills the rows that are listed in unbound_frames.txt with 1
  temp = matrix(ncol = 1, nrow = frames)

  unbound = t(read.table("unbound_frames.txt", sep=" "))
  rownames(unbound) = NULL
  unbound = as.vector(unbound)

  for (i in unbound){
    temp[i,1] = 1
  }

  #check if row contains NA and if it does, insert the next cluster value from the k-means data into that row
  j=1
  for(i in c(1:frames)){
    if (is.na(temp[i])){
      temp[i,1] = data[j,1]
      j = j+1
    }
  }
  return(temp)
}
```

Figure B1 (continued)



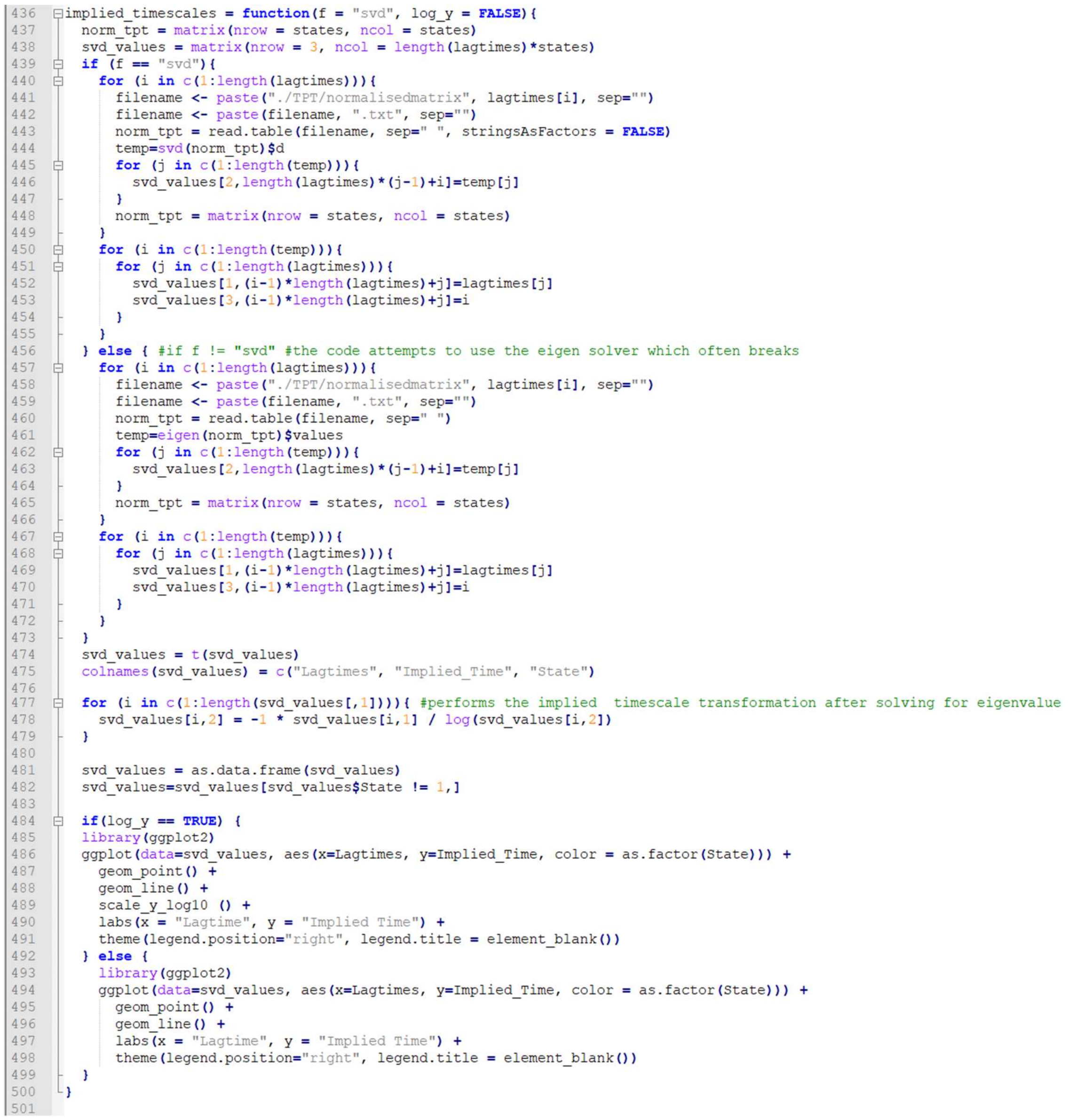
```
post_clustering <- function(){
  data <- as.vector(read.table(paste0(getwd(),"/KMresults/", "KMresults", toString(best_cluster),".txt"), sep="\n"))
  return(as.data.frame(recombine(data)))
  print("Unbound frames have been re-inserted as cluster 1 (all other cluster labels have increased by 2).")
}

combine_clusters <- function(list_to_combine){
  #iterate over each of the elements in the list
  for(i in c(1:length(list_to_combine))){
    #iterate over the entirety of the data
    for (j in c(1:length(data[,1]))){
      #check if current data element is present in any of the current list element's elements
      if (data[j,1] %in% list_to_combine[[i]]){
        #sets current data element to first element of current list element
        data[j,1] = list_to_combine[[i]][1]
      }
    }
  }

  #iterate over the data and create a vector containing all unique values
  unique = vector(mode="integer", length = 0)
  for (i in c(1:length(data[,1]))){
    if ((data[i,1] %in% unique) == FALSE) {
      unique = c(unique, data[i,1])
    }
  }
  unique = sort(unique)

  #iterate over the data
  for (i in c(1:length(data[,1]))){
    renumbered = FALSE
    #iterate over the values in unique
    for (j in c(1:length(unique))){
      #if current data value matches a value in unique replace data with unique (but not if it has already been renumbered)
      if (data[i,1] == unique[j] && renumbered == FALSE){
        data[i,1] = j
        renumbered = TRUE
      }
    }
  }

  states <<- length(unique)
  return(data)
}

finalize_clusters <- function(){
  write.table(data, combined_cluster_file, append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
}

post_clustering_analysis <- function(optional = FALSE){
  frames_per_cluster(states, combined_cluster_file, percentage = TRUE)
  print("Frames per cluster calculated and outputted to 'cluster_abundance.txt'.")
  if (optional == TRUE) {
    find_representative_frames(kmK = states, combined_cluster_file, unbound_removed=FALSE)
    get_representatives()
    print("Representative structures (including unbound) outputted to 'representatives.crd'.")
    print("Representatives for merged clusters may not be truly representative.")
    validation_frames(states=states, cluster_file = combined_cluster_file)
    cluster_validation()
    print("Validation trajectories for each cluster (including unbound) can be found in the 'cluster_frames' directory.")
  }
}

calculate_transition_matrices <- function(){
  data <<- prep_data()
  create_tpt()
  print("Transition matrices calculated.")
  normalize_tpt()
  print("Transition matrices were made symmetric and normalized.")
}

prep_data <- function(){
  data = setNames(data.frame(matrix(ncol = 1, nrow = frames)), c("Combined_Cluster_ID"))
  temp = read.table(combined_cluster_file, sep="\n")
  rownames(temp) = NULL
  data[1] = temp
  return(data)
}
```

Figure B1 (continued)



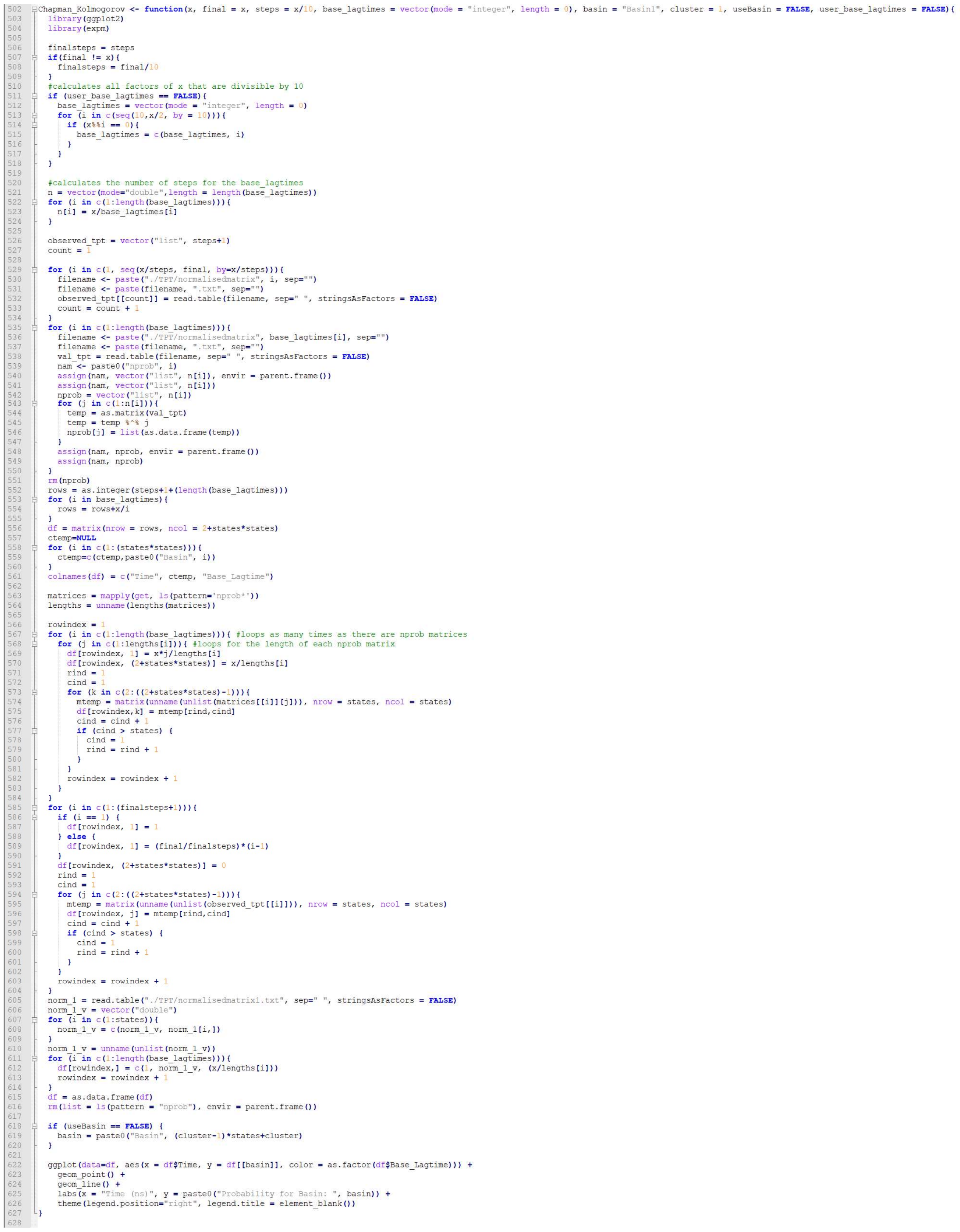
```r
create_tpt <- function(){

  tpt = matrix(0, nrow = states, ncol = states)

  library(foreach)
  library(doParallel)
  registerDoParallel(num_cores)

  if (number_of_traj != 1) {
    foreach (i = lagtimes) %dopar% {
      for (j in c(0:(number_of_traj-1))){ #loops for total number of trajectories prior to merge
        for (k in c(1:(frames_per_traj-i))){ #loops for currentframe+lagtime <= frames_per_traj, given that currentframe resets every time you reach frames_per_traj
          tpt[data[j*frames_per_traj+k,1], data[j*frames_per_traj+k+i,1]] = tpt[data[j*frames_per_traj+k,1], data[j*frames_per_traj+k+i,1]] + 1
        }
      }
      filename <- paste("./TPT/countmatrix", i, sep="")
      filename <- paste(filename, ".txt", sep="")
      write.table(tpt, file = filename, append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
      tpt = matrix(0, nrow = states, ncol = states) #this line is needed to reset to count matrix to 0 ... but it'll print every time
    }
  } else {
    foreach (i = lagtimes) %dopar% {
      #the print line below doesn't work with parallel for some reason
      #cat("Calculating Transition Matrix for Lagtime: ", i, "\n")
      for (j in c(1:(frames-i))) { #loops for all frames+lagtime <= total frames
        tpt[data[j,1], data[j+i,1]] = tpt[data[j,1], data[j+i,1]] + 1
      }
      filename <- paste("./TPT/countmatrix", i, sep="")
      filename <- paste(filename, ".txt", sep="")
      write.table(tpt, file = filename, append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
      tpt = matrix(0, nrow = states, ncol = states)
    }
  }
}

normalize_tpt <- function(){
  # N(ij) = N(ji)(transpose)
  # symmetry: Nsym = (N + N(transpose)) / 2
  library(foreach)
  library(doParallel)
  registerDoParallel(num_cores)
  norm_tpt = matrix(nrow = states, ncol = states)
  tpt = setNames(data.frame(matrix(ncol = states, nrow = states)), c(1:states))
  foreach (i = lagtimes) %dopar% {
    filename <- paste("./TPT/countmatrix", i, sep="")
    filename <- paste(filename, ".txt", sep="")
    tpt = read.table(filename, sep=" ")
    for (j in c(1:states)){
      for (k in c(1:states)){
        #symmetry step
        norm_tpt[j,k] = (tpt[j,k]+tpt[k,j])/2
      }
    }
    filename <- paste("./TPT/symmetricmatrix", i, sep="")
    filename <- paste(filename, ".txt", sep="")
    write.table(norm_tpt, file = filename, append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
    norm_tpt = matrix(nrow = states, ncol = states)
    tpt = setNames(data.frame(matrix(ncol = states, nrow = states)), c(1:states))
  }


  # normalize: P(ij) = N(ij)sym / sum(Nsym for row i)
  norm_tpt = matrix(nrow = states, ncol = states)
  sym_tpt = setNames(data.frame(matrix(ncol = states, nrow = states)), c(1:states))
  for (i in lagtimes){
    filename <- paste("./TPT/symmetricmatrix", i, sep="")
    filename <- paste(filename, ".txt", sep="")
    sym_tpt = read.table(filename, sep=" ")
    sym_rowSums = rowSums(sym_tpt)
    for (j in c(1:states)){
      for (k in c(1:states)){
        #normalizing step
        norm_tpt[j,k] = sym_tpt[j,k]/sym_rowSums[j]
      }
    }
    filename <- paste("./TPT/normalisedmatrix", i, sep="")
    filename <- paste(filename, ".txt", sep="")
    write.table(norm_tpt, file = filename, append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
    norm_tpt = matrix(nrow = states, ncol = states)
    sym_tpt = setNames(data.frame(matrix(ncol = states, nrow = states)), c(1:states))
  }
}
```

Figure B1 (continued). Function create_tpt depicted here does not support trajectories of different lengths. This was changed in a more recent version of the software but the updated raw code is currently inaccessible due to the COVID-19 outbreak at the time of this writing.



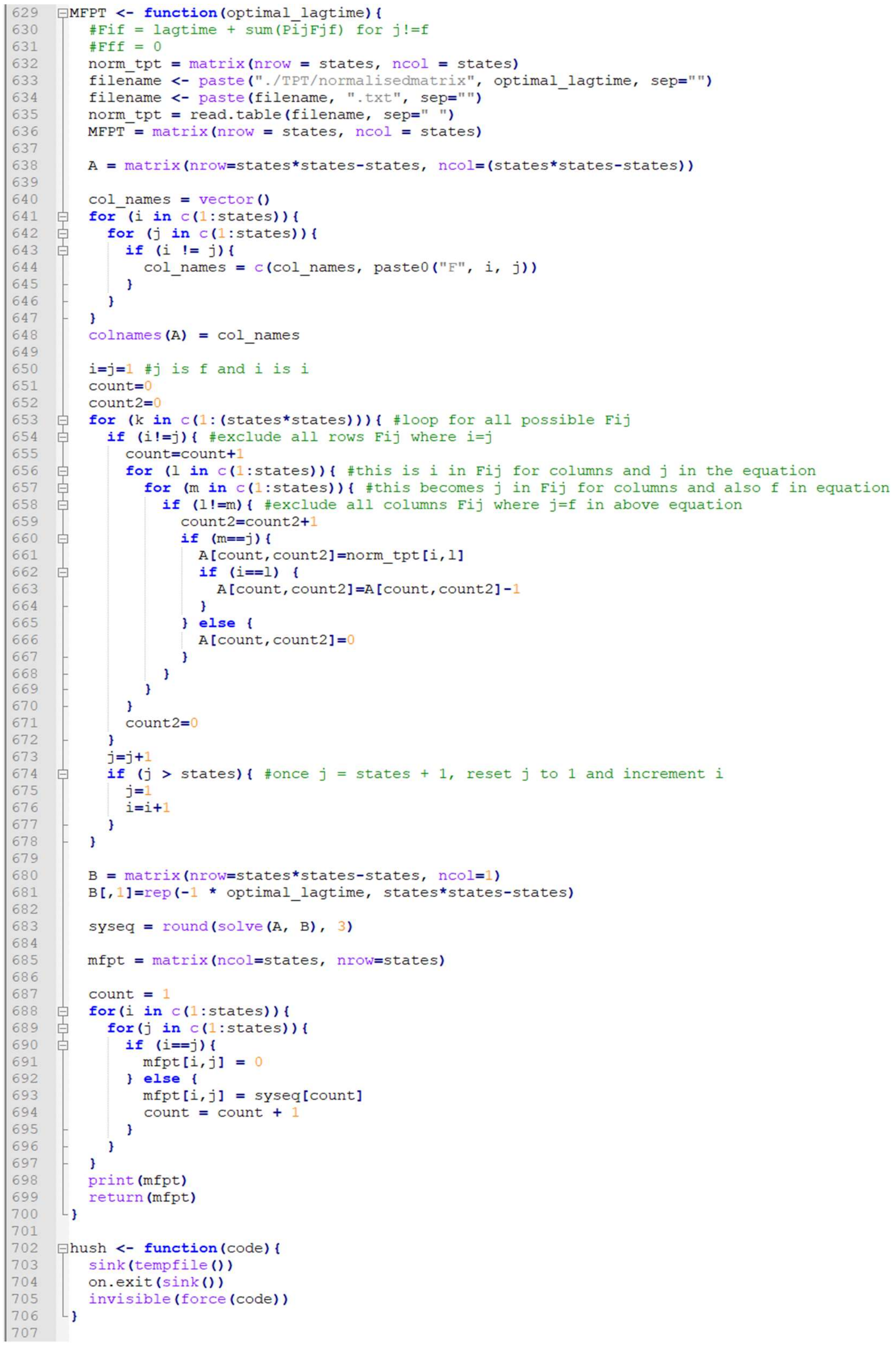
```r
implied_timescales = function(f = "svd", log_y = FALSE){
  norm_tpt = matrix(nrow = states, ncol = states)
  svd_values = matrix(nrow = 3, ncol = length(lagtimes)*states)
  if (f == "svd"){
    for (i in c(1:length(lagtimes))){
      filename <- paste("./TPT/normalisedmatrix", lagtimes[i], sep="")
      filename <- paste(filename, ".txt", sep="")
      norm_tpt = read.table(filename, sep=" ", stringsAsFactors = FALSE)
      temp=svd(norm_tpt)$d
      for (j in c(1:length(temp))){
        svd_values[2,length(lagtimes)*(j-1)+i]=temp[j]
      }
      norm_tpt = matrix(nrow = states, ncol = states)
    }
    for (i in c(1:length(temp))){
      for (j in c(1:length(lagtimes))){
        svd_values[1,(i-1)*length(lagtimes)+j]=lagtimes[j]
        svd_values[3,(i-1)*length(lagtimes)+j]=i
      }
    }
  } else { #if f != "svd" #the code attempts to use the eigen solver which often breaks
    for (i in c(1:length(lagtimes))){
      filename <- paste("./TPT/normalisedmatrix", lagtimes[i], sep="")
      filename <- paste(filename, ".txt", sep="")
      norm_tpt = read.table(filename, sep=" ")
      temp=eigen(norm_tpt)$values
      for (j in c(1:length(temp))){
        svd_values[2,length(lagtimes)*(j-1)+i]=temp[j]
      }
      norm_tpt = matrix(nrow = states, ncol = states)
    }
    for (i in c(1:length(temp))){
      for (j in c(1:length(lagtimes))){
        svd_values[1,(i-1)*length(lagtimes)+j]=lagtimes[j]
        svd_values[3,(i-1)*length(lagtimes)+j]=i
      }
    }
  }
  svd_values = t(svd_values)
  colnames(svd_values) = c("Lagtimes", "Implied_Time", "State")

  for (i in c(1:length(svd_values[,1]))){ #performs the implied  timescale transformation after solving for eigenvalue
    svd_values[i,2] = -1 * svd_values[i,1] / log(svd_values[i,2])
  }

  svd_values = as.data.frame(svd_values)
  svd_values=svd_values[svd_values$State != 1,]

  if(log_y == TRUE) {
  library(ggplot2)
  ggplot(data=svd_values, aes(x=Lagtimes, y=Implied_Time, color = as.factor(State))) +
    geom_point() +
    geom_line() +
    scale_y_log10 () +
    labs(x = "Lagtime", y = "Implied Time") +
    theme(legend.position="right", legend.title = element_blank())
  } else {
    library(ggplot2)
    ggplot(data=svd_values, aes(x=Lagtimes, y=Implied_Time, color = as.factor(State))) +
      geom_point() +
      geom_line() +
      labs(x = "Lagtime", y = "Implied Time") +
      theme(legend.position="right", legend.title = element_blank())
  }
}
```

Figure B1 (continued)



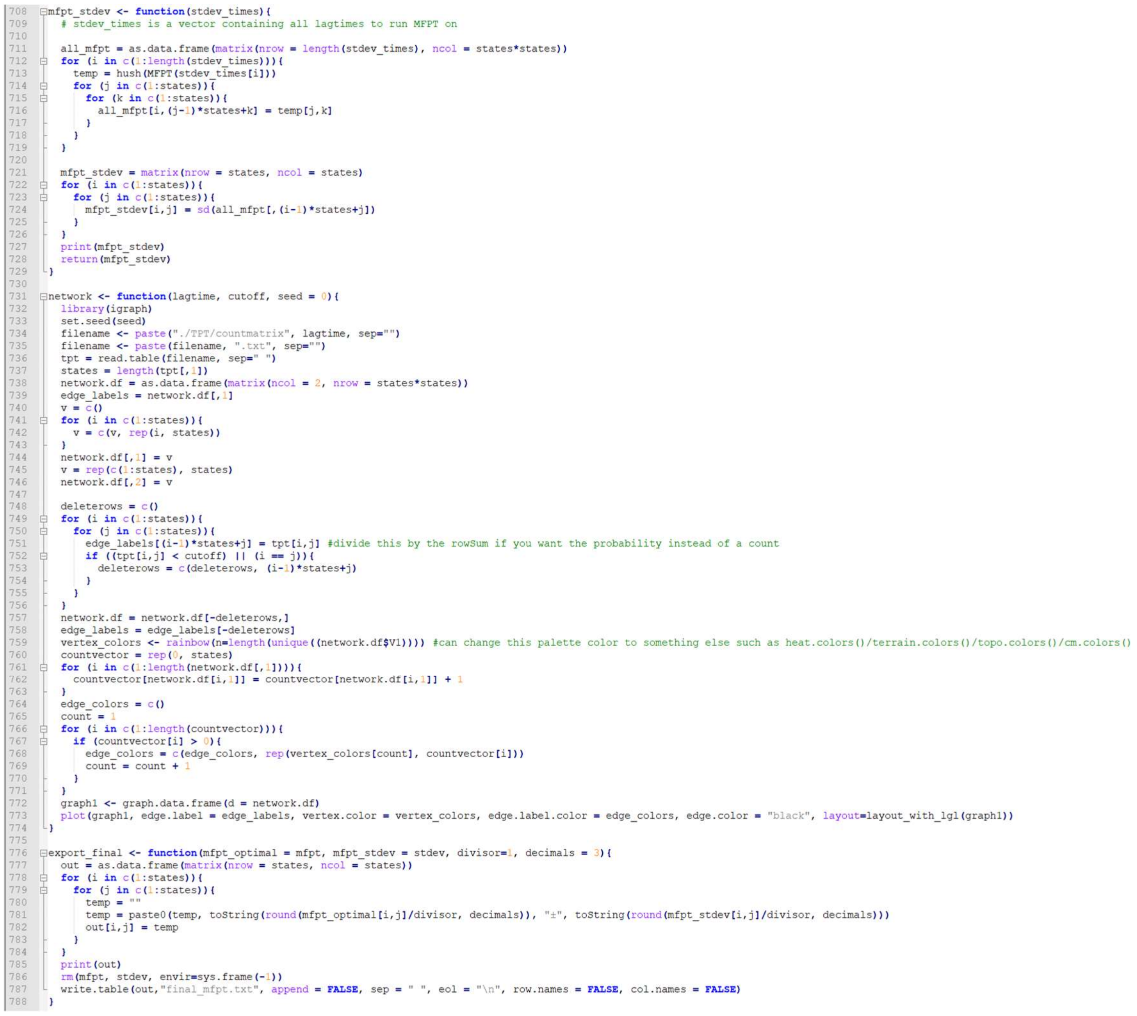
```r
Chapman_Kolmogorov <- function(x, final = x, steps = x/10, base_lagtimes = vector(mode = "integer", length = 0), basin = "Basin1", cluster = 1, useBasin = FALSE, user_base_lagtimes = FALSE){
  library(ggplot2)
  library(expm)

  finalsteps = steps
  if(final != x){
    finalsteps = final/10
  }
  #calculates all factors of x that are divisible by 10
  if (user_base_lagtimes == FALSE){
    base_lagtimes = vector(mode = "integer", length = 0)
    for (i in c(seq(10,x/2, by = 10))){
      if (x%%i == 0){
        base_lagtimes = c(base_lagtimes, i)
      }
    }
  }

  #calculates the number of steps for the base_lagtimes
  n = vector(mode="double",length = length(base_lagtimes))
  for (i in c(1:length(base_lagtimes))){
    n[i] = x/base_lagtimes[i]
  }

  observed_tpt = vector("list", steps+1)
  count = 1

  for (i in c(1, seq(x/steps, final, by=x/steps))){
    filename <- paste("./TPT/normalisedmatrix", i, sep="")
    filename <- paste(filename, ".txt", sep="")
    observed_tpt[[count]] = read.table(filename, sep=" ", stringsAsFactors = FALSE)
    count = count + 1
  }
  for (i in c(1:length(base_lagtimes))){
    filename <- paste("./TPT/normalisedmatrix", base_lagtimes[i], sep="")
    filename <- paste(filename, ".txt", sep="")
    val_tpt = read.table(filename, sep=" ", stringsAsFactors = FALSE)
    nam <- paste0("nprob", i)
    assign(nam, vector("list", n[i]), envir = parent.frame())
    assign(nam, vector("list", n[i]))
    nprob = vector("list", n[i])
    for (j in c(1:n[i])){
      temp = as.matrix(val_tpt)
      temp = temp %^% j
      nprob[j] = list(as.data.frame(temp))
    }
    assign(nam, nprob, envir = parent.frame())
    assign(nam, nprob)
  }
  rm(nprob)
  rows = as.integer(steps+1+(length(base_lagtimes)))
  for (i in base_lagtimes){
    rows = rows+x/i
  }
  df = matrix(nrow = rows, ncol = 2+states*states)
  ctemp=NULL
  for (i in c(1:(states*states))){
    ctemp=c(ctemp,paste0("Basin", i))
  }
  colnames(df) = c("Time", ctemp, "Base_Lagtime")

  matrices = mapply(get, ls(pattern='nprob*'))
  lengths = unname(lengths(matrices))

  rowindex = 1
  for (i in c(1:length(base_lagtimes))){ #loops as many times as there are nprob matrices
    for (j in c(1:lengths[i])){ #loops for the length of each nprob matrix
      df[rowindex, 1] = x*j/lengths[i]
      df[rowindex, (2+states*states)] = x/lengths[i]
      rind = 1
      cind = 1
      for (k in c(2:((2+states*states)-1))){
        mtemp = matrix(unname(unlist(matrices[[i]][j])), nrow = states, ncol = states)
        df[rowindex,k] = mtemp[rind,cind]
        cind = cind + 1
        if (cind > states) {
          cind = 1
          rind = rind + 1
        }
      }
      rowindex = rowindex + 1
    }
  }
  for (i in c(1:(finalsteps+1))){
    if (i == 1) {
      df[rowindex, 1] = 1
    } else {
      df[rowindex, 1] = (final/finalsteps)*(i-1)
    }
    df[rowindex, (2+states*states)] = 0
    rind = 1
    cind = 1
    for (j in c(2:((2+states*states)-1))){
      mtemp = matrix(unname(unlist(observed_tpt[[i]])), nrow = states, ncol = states)
      df[rowindex, j] = mtemp[rind,cind]
      cind = cind + 1
      if (cind > states) {
        cind = 1
        rind = rind + 1
      }
    }
    rowindex = rowindex + 1
  }
  norm_1 = read.table("./TPT/normalisedmatrix1.txt", sep=" ", stringsAsFactors = FALSE)
  norm_1_v = vector("double")
  for (i in c(1:states)){
    norm_1_v = c(norm_1_v, norm_1[i,])
  }
  norm_1_v = unname(unlist(norm_1_v))
  for (i in c(1:length(base_lagtimes))){
    df[rowindex,] = c(1, norm_1_v, (x/lengths[i]))
    rowindex = rowindex + 1
  }
  df = as.data.frame(df)
  rm(list = ls(pattern = "nprob"), envir = parent.frame())

  if (useBasin == FALSE) {
    basin = paste0("Basin", (cluster-1)*states+cluster)
  }

  ggplot(data=df, aes(x = df$Time, y = df[[basin]], color = as.factor(df$Base_Lagtime))) +
    geom_point() +
    geom_line() +
    labs(x = "Time (ns)", y = paste0("Probability for Basin: ", basin)) +
    theme(legend.position="right", legend.title = element_blank())
}
```

Figure B1 (continued)



```r
MFPT <- function(optimal_lagtime){
  #Fif = lagtime + sum(PijFjf) for j!=f
  #Fff = 0
  norm_tpt = matrix(nrow = states, ncol = states)
  filename <- paste("./TPT/normalisedmatrix", optimal_lagtime, sep="")
  filename <- paste(filename, ".txt", sep="")
  norm_tpt = read.table(filename, sep=" ")
  MFPT = matrix(nrow = states, ncol = states)

  A = matrix(nrow=states*states-states, ncol=(states*states-states))

  col_names = vector()
  for (i in c(1:states)){
    for (j in c(1:states)){
      if (i != j){
        col_names = c(col_names, paste0("F", i, j))
      }
    }
  }
  colnames(A) = col_names

  i=j=1 #j is f and i is i
  count=0
  count2=0
  for (k in c(1:(states*states))){ #loop for all possible Fij
    if (i!=j){ #exclude all rows Fij where i=j
      count=count+1
      for (l in c(1:states)){ #this is i in Fij for columns and j in the equation
        for (m in c(1:states)){ #this becomes j in Fij for columns and also f in equation
          if (l!=m){ #exclude all columns Fij where j=f in above equation
            count2=count2+1
            if (m==j){
              A[count,count2]=norm_tpt[i,l]
              if (i==l) {
                A[count,count2]=A[count,count2]-1
              }
            } else {
              A[count,count2]=0
            }
          }
        }
      }
      count2=0
    }
    j=j+1
    if (j > states){ #once j = states + 1, reset j to 1 and increment i
      j=1
      i=i+1
    }
  }

  B = matrix(nrow=states*states-states, ncol=1)
  B[,1]=rep(-1 * optimal_lagtime, states*states-states)

  syseq = round(solve(A, B), 3)

  mfpt = matrix(ncol=states, nrow=states)

  count = 1
  for(i in c(1:states)){
    for(j in c(1:states)){
      if (i==j){
        mfpt[i,j] = 0
      } else {
        mfpt[i,j] = syseq[count]
        count = count + 1
      }
    }
  }
  print(mfpt)
  return(mfpt)
}

hush <- function(code){
  sink(tempfile())
  on.exit(sink())
  invisible(force(code))
}
```

Figure B1 (continued)



```r
mfpt_stdev <- function(stdev_times){
  # stdev_times is a vector containing all lagtimes to run MFPT on

  all_mfpt = as.data.frame(matrix(nrow = length(stdev_times), ncol = states*states))
  for (i in c(1:length(stdev_times))){
    temp = hush(MFPT(stdev_times[i]))
    for (j in c(1:states)){
      for (k in c(1:states)){
        all_mfpt[i,(j-1)*states+k] = temp[j,k]
      }
    }
  }

  mfpt_stdev = matrix(nrow = states, ncol = states)
  for (i in c(1:states)){
    for (j in c(1:states)){
      mfpt_stdev[i,j] = sd(all_mfpt[,(i-1)*states+j])
    }
  }
  print(mfpt_stdev)
  return(mfpt_stdev)
}

network <- function(lagtime, cutoff, seed = 0){
  library(igraph)
  set.seed(seed)
  filename <- paste("./TPT/countmatrix", lagtime, sep="")
  filename <- paste(filename, ".txt", sep="")
  tpt = read.table(filename, sep=" ")
  states = length(tpt[,1])
  network.df = as.data.frame(matrix(ncol = 2, nrow = states*states))
  edge_labels = network.df[,1]
  v = c()
  for (i in c(1:states)){
    v = c(v, rep(i, states))
  }
  network.df[,1] = v
  v = rep(c(1:states), states)
  network.df[,2] = v

  deleterows = c()
  for (i in c(1:states)){
    for (j in c(1:states)){
      edge_labels[(i-1)*states+j] = tpt[i,j] #divide this by the rowSum if you want the probability instead of a count
      if ((tpt[i,j] < cutoff) || (i == j)){
        deleterows = c(deleterows, (i-1)*states+j)
      }
    }
  }
  network.df = network.df[-deleterows,]
  edge_labels = edge_labels[-deleterows]
  vertex_colors <- rainbow(n=length(unique((network.df$V1)))) #can change this palette color to something else such as heat.colors()/terrain.colors()/topo.colors()/cm.colors()
  countvector = rep(0, states)
  for (i in c(1:length(network.df[,1]))){
    countvector[network.df[i,1]] = countvector[network.df[i,1]] + 1
  }
  edge_colors = c()
  count = 1
  for (i in c(1:length(countvector))){
    if (countvector[i] > 0){
      edge_colors = c(edge_colors, rep(vertex_colors[count], countvector[i]))
      count = count + 1
    }
  }
  graph1 <- graph.data.frame(d = network.df)
  plot(graph1, edge.label = edge_labels, vertex.color = vertex_colors, edge.label.color = edge_colors, edge.color = "black", layout=layout_with_lgl(graph1))
}

export_final <- function(mfpt_optimal = mfpt, mfpt_stdev = stdev, divisor=1, decimals = 3){
  out = as.data.frame(matrix(nrow = states, ncol = states))
  for (i in c(1:states)){
    for (j in c(1:states)){
      temp = ""
      temp = paste0(temp, toString(round(mfpt_optimal[i,j]/divisor, decimals)), "±", toString(round(mfpt_stdev[i,j]/divisor, decimals)))
      out[i,j] = temp
    }
  }
  print(out)
  rm(mfpt, stdev, envir=sys.frame(-1))
  write.table(out,"final_mfpt.txt", append = FALSE, sep = " ", eol = "\n", row.names = FALSE, col.names = FALSE)
}
```

Figure B1 (continued)

Appendix C

functions.py

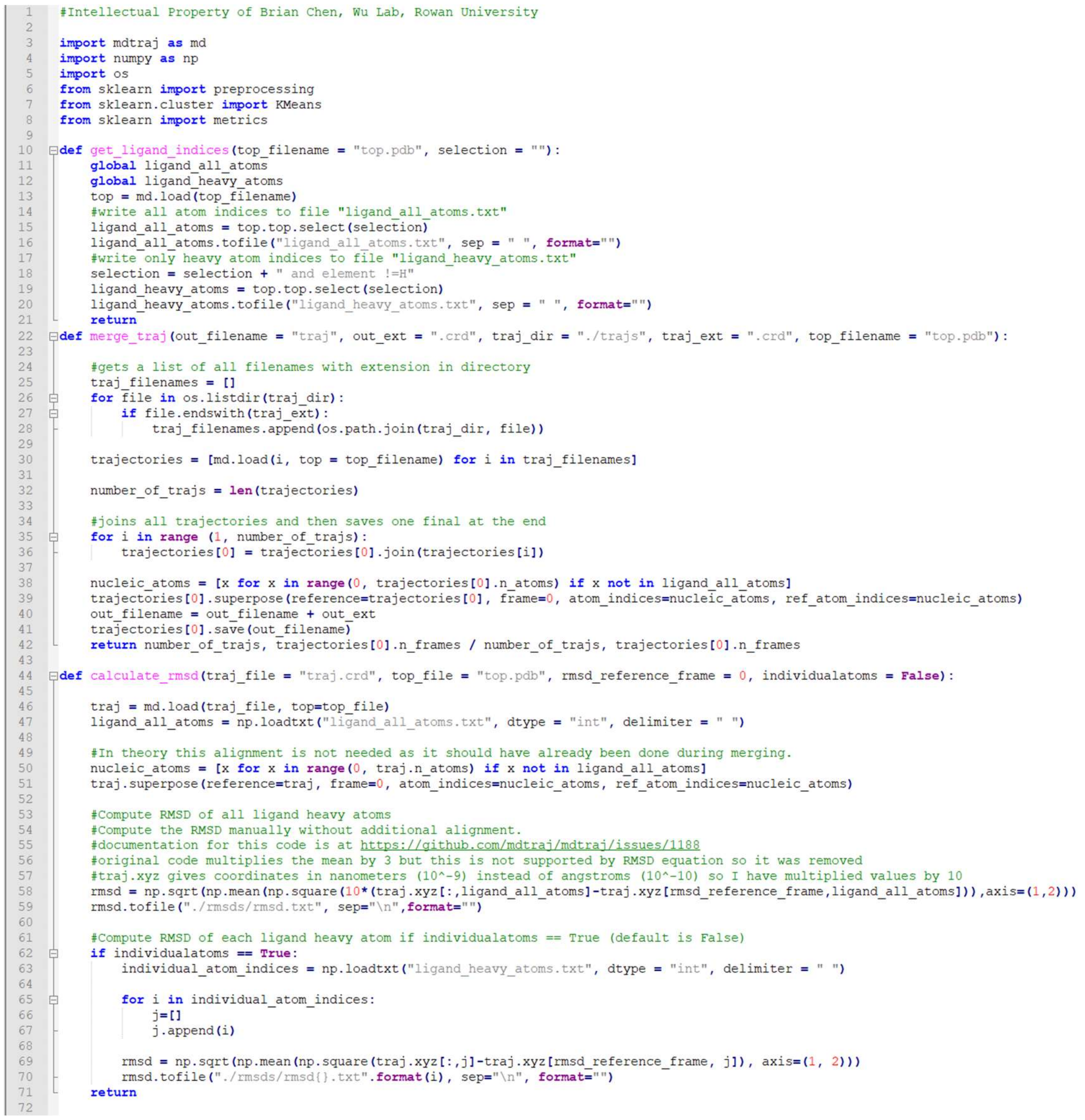
Pertinent instructions and other comments are shown in green after the '#' symbols. These are not part not the code itself and only serve to help the user. Table C1 includes a helpful list of all functions present in functions.R.

Table C1

Functions present in functions.py

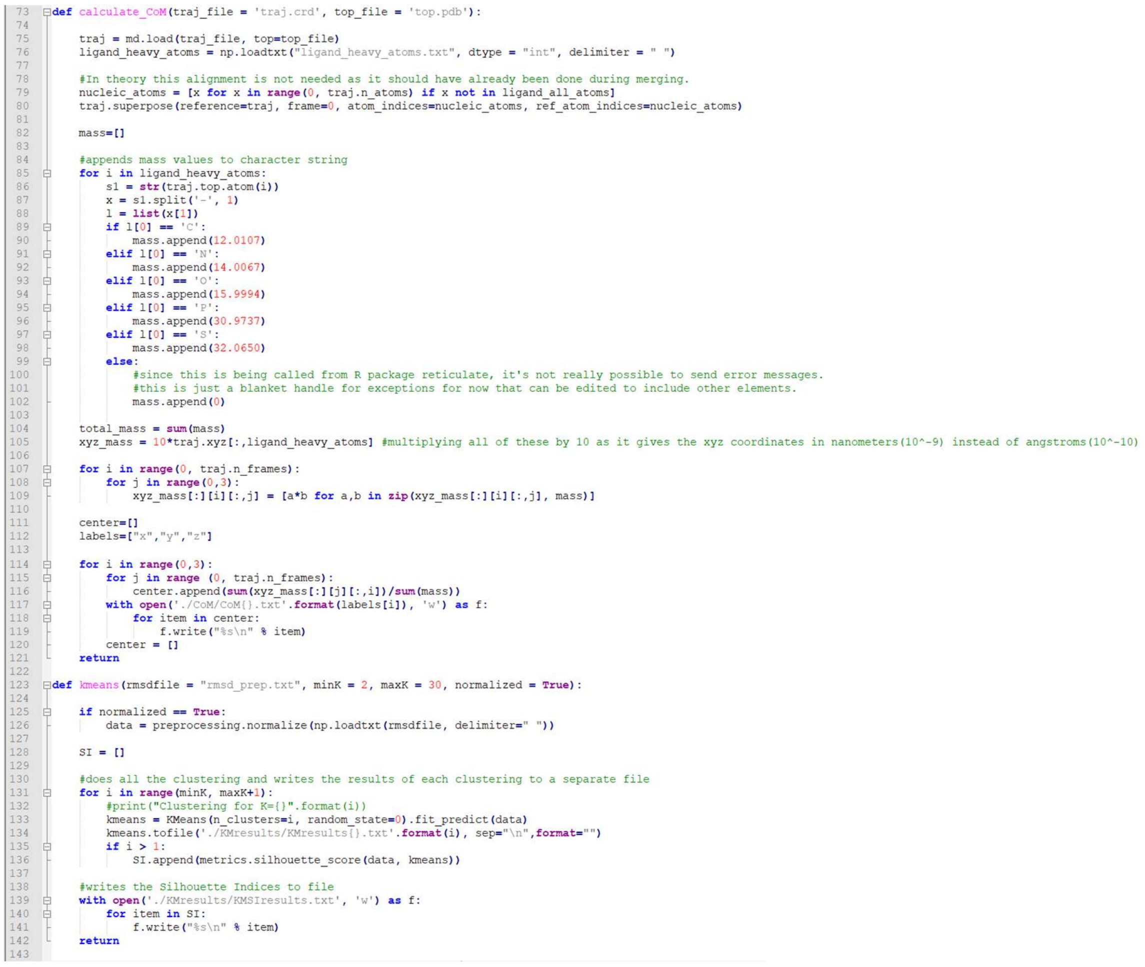
Functions are listed in sequential order with their primary purpose.

Function	Lines	Primary Purpose
get_ligand_indices	10-20	Obtains atom indices of the ligand
merge_traj	22-42	Merge trajectories and output to "traj.crd"
calculate_rmsd	44-71	Calculates RMSD values
calculate_CoM	73-121	Calculates center of mass (X, Y, Z)
kmeans	123-142	Performs *k*-means clustering
get_representatives	144-161	Outputs representatives to "representatives.crd"
cluster_validation	163-186	Outputs validation trajectories to cluster_frames directory



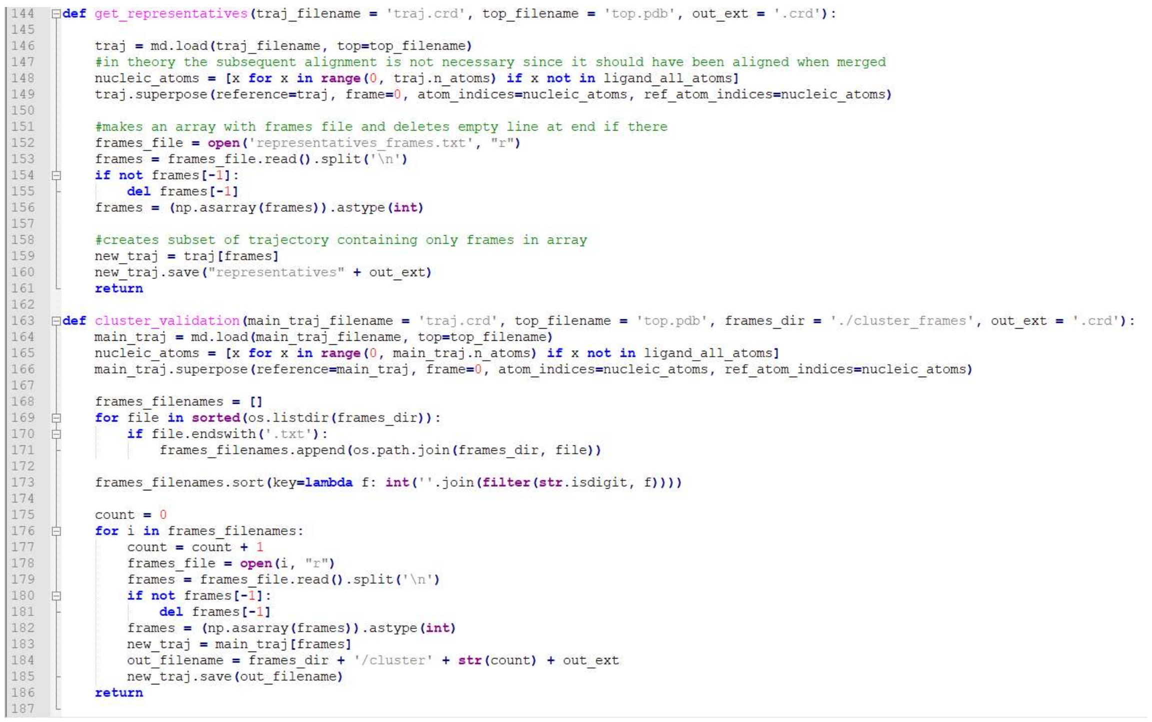
```python
#Intellectual Property of Brian Chen, Wu Lab, Rowan University

import mdtraj as md
import numpy as np
import os
from sklearn import preprocessing
from sklearn.cluster import KMeans
from sklearn import metrics

def get_ligand_indices(top_filename = "top.pdb", selection = ""):
    global ligand_all_atoms
    global ligand_heavy_atoms
    top = md.load(top_filename)
    #write all atom indices to file "ligand_all_atoms.txt"
    ligand_all_atoms = top.top.select(selection)
    ligand_all_atoms.tofile("ligand_all_atoms.txt", sep = " ", format="")
    #write only heavy atom indices to file "ligand_heavy_atoms.txt"
    selection = selection + " and element !=H"
    ligand_heavy_atoms = top.top.select(selection)
    ligand_heavy_atoms.tofile("ligand_heavy_atoms.txt", sep = " ", format="")
    return
def merge_traj(out_filename = "traj", out_ext = ".crd", traj_dir = "./trajs", traj_ext = ".crd", top_filename = "top.pdb"):

    #gets a list of all filenames with extension in directory
    traj_filenames = []
    for file in os.listdir(traj_dir):
        if file.endswith(traj_ext):
            traj_filenames.append(os.path.join(traj_dir, file))

    trajectories = [md.load(i, top = top_filename) for i in traj_filenames]

    number_of_trajs = len(trajectories)

    #joins all trajectories and then saves one final at the end
    for i in range (1, number_of_trajs):
        trajectories[0] = trajectories[0].join(trajectories[i])

    nucleic_atoms = [x for x in range(0, trajectories[0].n_atoms) if x not in ligand_all_atoms]
    trajectories[0].superpose(reference=trajectories[0], frame=0, atom_indices=nucleic_atoms, ref_atom_indices=nucleic_atoms)
    out_filename = out_filename + out_ext
    trajectories[0].save(out_filename)
    return number_of_trajs, trajectories[0].n_frames / number_of_trajs, trajectories[0].n_frames

def calculate_rmsd(traj_file = "traj.crd", top_file = "top.pdb", rmsd_reference_frame = 0, individualatoms = False):

    traj = md.load(traj_file, top=top_file)
    ligand_all_atoms = np.loadtxt("ligand_all_atoms.txt", dtype = "int", delimiter = " ")

    #In theory this alignment is not needed as it should have already been done during merging.
    nucleic_atoms = [x for x in range(0, traj.n_atoms) if x not in ligand_all_atoms]
    traj.superpose(reference=traj, frame=0, atom_indices=nucleic_atoms, ref_atom_indices=nucleic_atoms)

    #Compute RMSD of all ligand heavy atoms
    #Compute the RMSD manually without additional alignment.
    #documentation for this code is at https://github.com/mdtraj/mdtraj/issues/1188
    #original code multiplies the mean by 3 but this is not supported by RMSD equation so it was removed
    #traj.xyz gives coordinates in nanometers (10^-9) instead of angstroms (10^-10) so I have multiplied values by 10
    rmsd = np.sqrt(np.mean(np.square(10*(traj.xyz[:,ligand_all_atoms]-traj.xyz[rmsd_reference_frame,ligand_all_atoms])),axis=(1,2)))
    rmsd.tofile("./rmsds/rmsd.txt", sep="\n",format="")

    #Compute RMSD of each ligand heavy atom if individualatoms == True (default is False)
    if individualatoms == True:
        individual_atom_indices = np.loadtxt("ligand_heavy_atoms.txt", dtype = "int", delimiter = " ")

        for i in individual_atom_indices:
            j=[]
            j.append(i)

        rmsd = np.sqrt(np.mean(np.square(traj.xyz[:,j]-traj.xyz[rmsd_reference_frame, j]), axis=(1, 2)))
        rmsd.tofile("./rmsds/rmsd{}.txt".format(i), sep="\n", format="")
    return

```

Figure C1. Code in functions.py. This figure shows all of the code present in functions.py. Function merge_traj depicted here does not return the number of frames for each individual trajectory. This has been fixed in a more recent version of the software but the updated raw code is currently inaccessible due to the COVID-19 outbreak at the time of this writing.



```python
def calculate_CoM(traj_file = 'traj.crd', top_file = 'top.pdb'):

    traj = md.load(traj_file, top=top_file)
    ligand_heavy_atoms = np.loadtxt("ligand_heavy_atoms.txt", dtype = "int", delimiter = " ")

    #In theory this alignment is not needed as it should have already been done during merging.
    nucleic_atoms = [x for x in range(0, traj.n_atoms) if x not in ligand_all_atoms]
    traj.superpose(reference=traj, frame=0, atom_indices=nucleic_atoms, ref_atom_indices=nucleic_atoms)

    mass=[]

    #appends mass values to character string
    for i in ligand_heavy_atoms:
        s1 = str(traj.top.atom(i))
        x = s1.split('-', 1)
        l = list(x[1])
        if l[0] == 'C':
            mass.append(12.0107)
        elif l[0] == 'N':
            mass.append(14.0067)
        elif l[0] == 'O':
            mass.append(15.9994)
        elif l[0] == 'P':
            mass.append(30.9737)
        elif l[0] == 'S':
            mass.append(32.0650)
        else:
            #since this is being called from R package reticulate, it's not really possible to send error messages.
            #this is just a blanket handle for exceptions for now that can be edited to include other elements.
            mass.append(0)

    total_mass = sum(mass)
    xyz_mass = 10*traj.xyz[:,ligand_heavy_atoms] #multiplying all of these by 10 as it gives the xyz coordinates in nanometers(10^-9) instead of angstroms(10^-10)

    for i in range(0, traj.n_frames):
        for j in range(0,3):
            xyz_mass[:][i][:,j] = [a*b for a,b in zip(xyz_mass[:][i][:,j], mass)]

    center=[]
    labels=["x","y","z"]

    for i in range(0,3):
        for j in range (0, traj.n_frames):
            center.append(sum(xyz_mass[:][j][:,i])/sum(mass))
        with open('./CoM/CoM{}.txt'.format(labels[i]), 'w') as f:
            for item in center:
                f.write("%s\n" % item)
        center = []
    return

def kmeans(rmsdfile = "rmsd_prep.txt", minK = 2, maxK = 30, normalized = True):

    if normalized == True:
        data = preprocessing.normalize(np.loadtxt(rmsdfile, delimiter=" "))

    SI = []

    #does all the clustering and writes the results of each clustering to a separate file
    for i in range(minK, maxK+1):
        #print("Clustering for K={}".format(i))
        kmeans = KMeans(n_clusters=i, random_state=0).fit_predict(data)
        kmeans.tofile('./KMresults/KMresults{}.txt'.format(i), sep="\n",format="")
        if i > 1:
            SI.append(metrics.silhouette_score(data, kmeans))

    #writes the Silhouette Indices to file
    with open('./KMresults/KMSIresults.txt', 'w') as f:
        for item in SI:
            f.write("%s\n" % item)
    return

```

Figure C1 (continued)



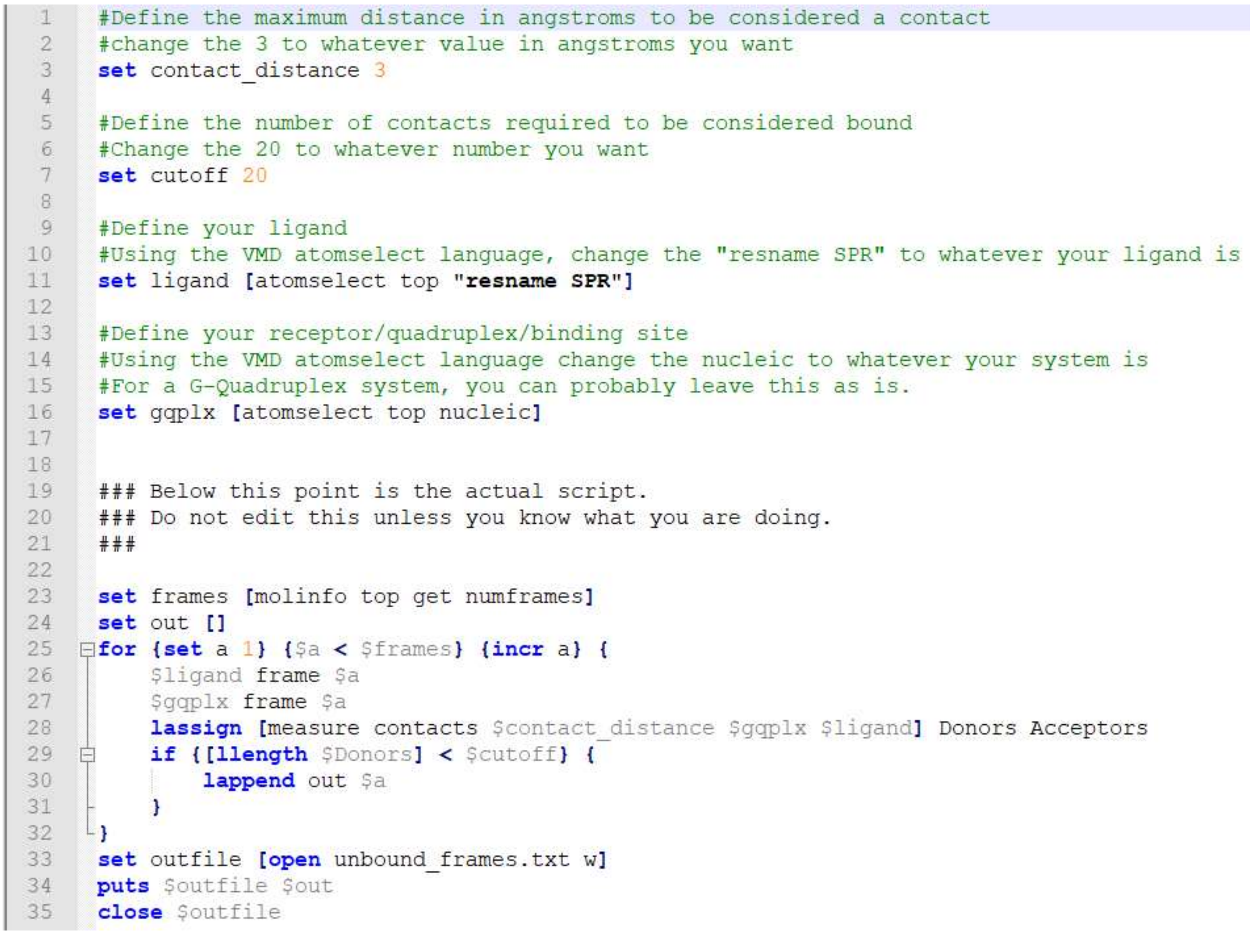
```python
def get_representatives(traj_filename = 'traj.crd', top_filename = 'top.pdb', out_ext = '.crd'):

    traj = md.load(traj_filename, top=top_filename)
    #in theory the subsequent alignment is not necessary since it should have been aligned when merged
    nucleic_atoms = [x for x in range(0, traj.n_atoms) if x not in ligand_all_atoms]
    traj.superpose(reference=traj, frame=0, atom_indices=nucleic_atoms, ref_atom_indices=nucleic_atoms)

    #makes an array with frames file and deletes empty line at end if there
    frames_file = open('representatives_frames.txt', "r")
    frames = frames_file.read().split('\n')
    if not frames[-1]:
        del frames[-1]
    frames = (np.asarray(frames)).astype(int)

    #creates subset of trajectory containing only frames in array
    new_traj = traj[frames]
    new_traj.save("representatives" + out_ext)
    return

def cluster_validation(main_traj_filename = 'traj.crd', top_filename = 'top.pdb', frames_dir = './cluster_frames', out_ext = '.crd'):
    main_traj = md.load(main_traj_filename, top=top_filename)
    nucleic_atoms = [x for x in range(0, main_traj.n_atoms) if x not in ligand_all_atoms]
    main_traj.superpose(reference=main_traj, frame=0, atom_indices=nucleic_atoms, ref_atom_indices=nucleic_atoms)

    frames_filenames = []
    for file in sorted(os.listdir(frames_dir)):
        if file.endswith('.txt'):
            frames_filenames.append(os.path.join(frames_dir, file))

    frames_filenames.sort(key=lambda f: int(''.join(filter(str.isdigit, f))))

    count = 0
    for i in frames_filenames:
        count = count + 1
        frames_file = open(i, "r")
        frames = frames_file.read().split('\n')
        if not frames[-1]:
            del frames[-1]
        frames = (np.asarray(frames)).astype(int)
        new_traj = main_traj[frames]
        out_filename = frames_dir + '/cluster' + str(count) + out_ext
        new_traj.save(out_filename)
    return
```

Figure C1 (continued)

Appendix D

find_unbound_frames.tcl

Pertinent instructions and other comments are shown in green after the '#' symbols. These are not part not the code itself and only serve to help the user.

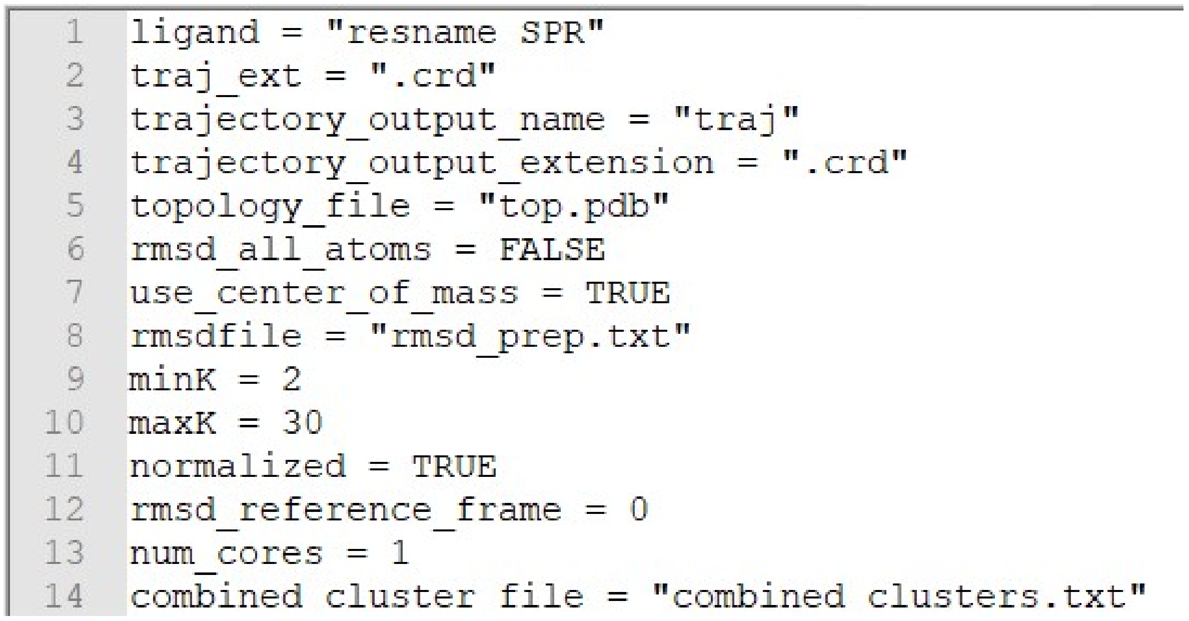


```
#Define the maximum distance in angstroms to be considered a contact
#change the 3 to whatever value in angstroms you want
set contact_distance 3

#Define the number of contacts required to be considered bound
#Change the 20 to whatever number you want
set cutoff 20

#Define your ligand
#Using the VMD atomselect language, change the "resname SPR" to whatever your ligand is
set ligand [atomselect top "resname SPR"]

#Define your receptor/quadruplex/binding site
#Using the VMD atomselect language change the nucleic to whatever your system is
#For a G-Quadruplex system, you can probably leave this as is.
set gqplx [atomselect top nucleic]


### Below this point is the actual script.
### Do not edit this unless you know what you are doing.
###

set frames [molinfo top get numframes]
set out []
for {set a 1} {$a < $frames} {incr a} {
    $ligand frame $a
    $gqplx frame $a
    lassign [measure contacts $contact_distance $gqplx $ligand] Donors Acceptors
    if {[llength $Donors] < $cutoff} {
        lappend out $a
    }
}
set outfile [open unbound_frames.txt w]
puts $outfile $out
close $outfile
```

Figure D1. Code in find_unbound_frames.tcl. This figure shows all of the code present in find_unbound_frames.tcl.

Appendix E

config.txt



```
ligand = "resname SPR"
traj_ext = ".crd"
trajectory_output_name = "traj"
trajectory_output_extension = ".crd"
topology_file = "top.pdb"
rmsd_all_atoms = FALSE
use_center_of_mass = TRUE
rmsdfile = "rmsd_prep.txt"
minK = 2
maxK = 30
normalized = TRUE
rmsd_reference_frame = 0
num_cores = 1
combined cluster file = "combined clusters.txt"
```

Figure E1. Configuration file. This figure shows all of the variables in the configuration file (config.txt).

Appendix F

RMSD, Atom Contacts, and Last Snapshots

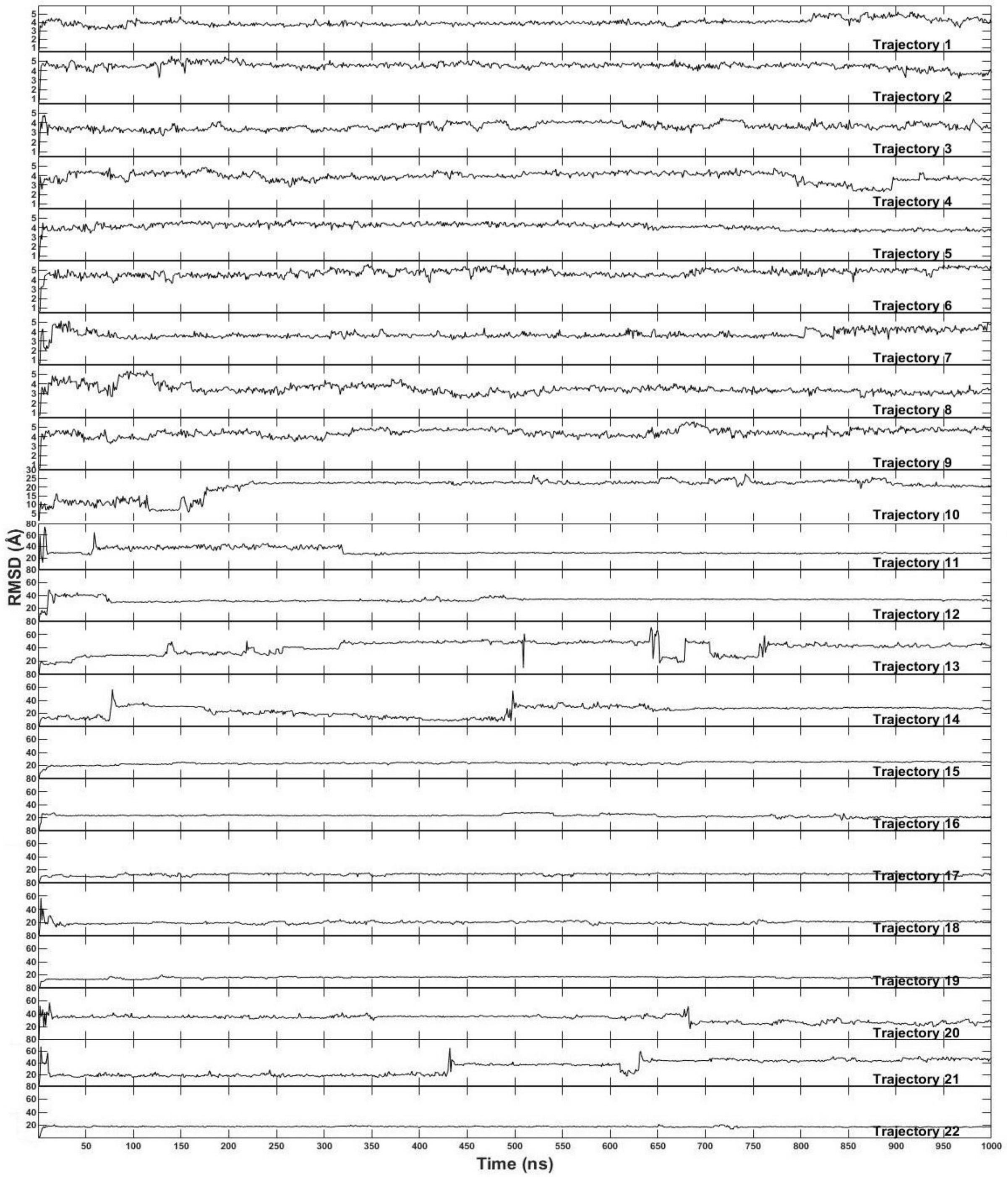


Figure F1. Ligand RMSD. Ligand RMSD in the 33 free ligand binding simulations of DBD1 to the Pu24 G-quadruplex.

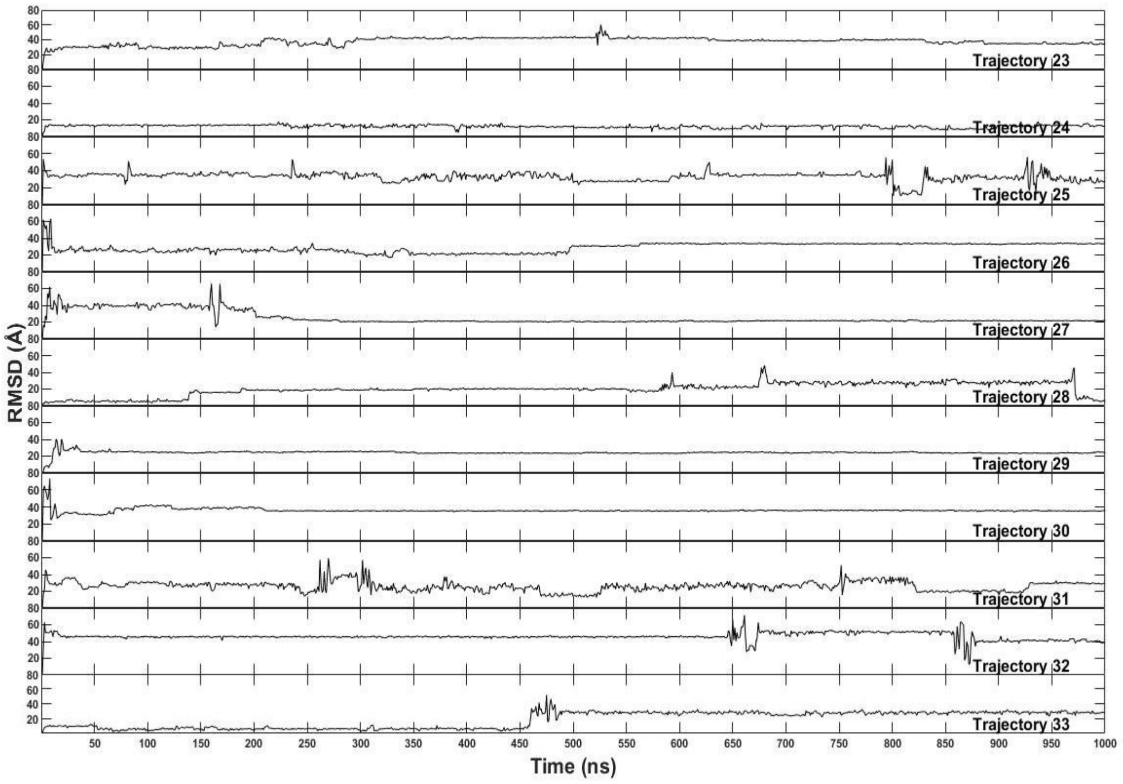


Figure F1 (continued)

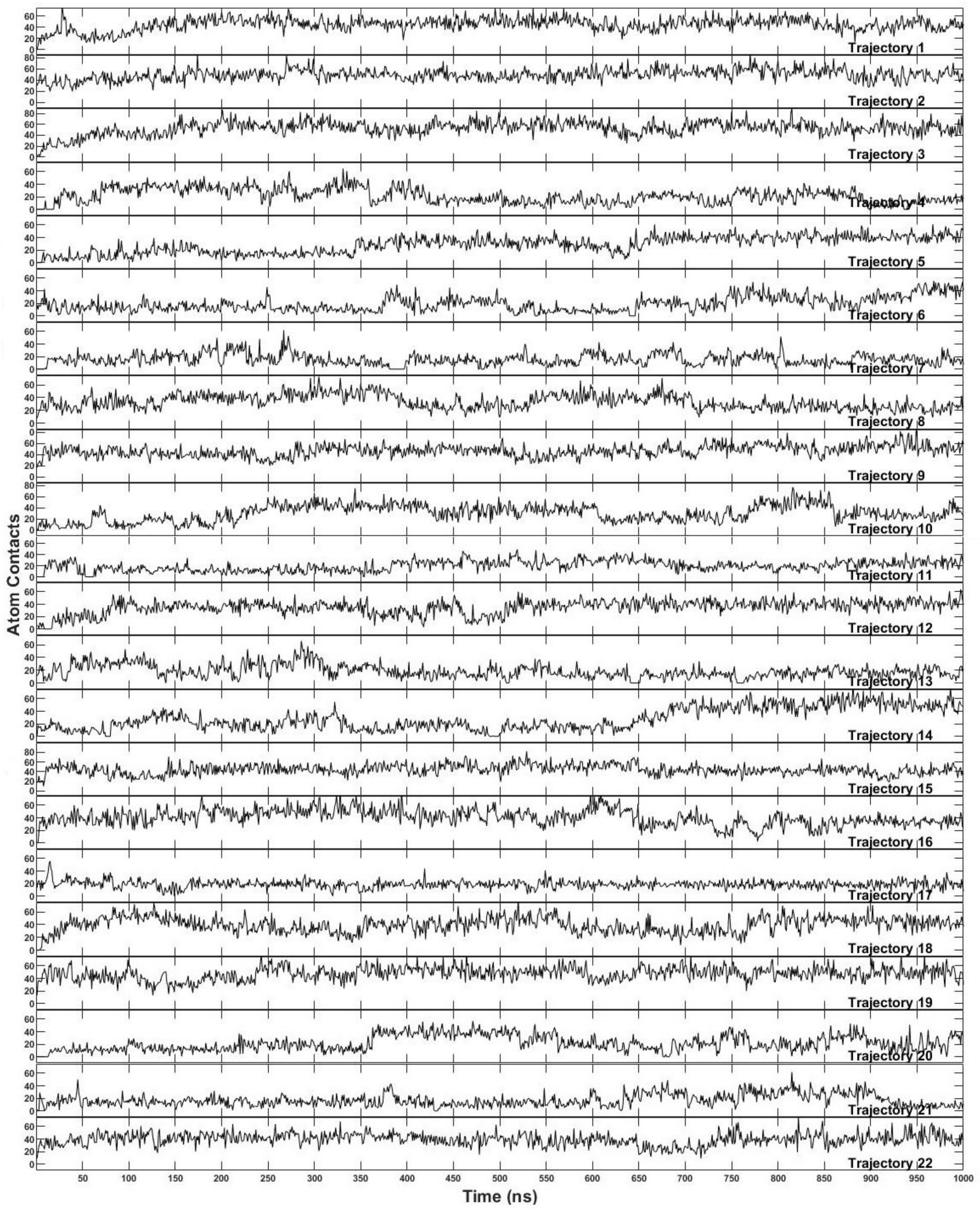


Figure F2. Atom contacts. Atom contacts between DBD1 and the Pu24 G-quadruplex in the 33 free ligand binding simulations.

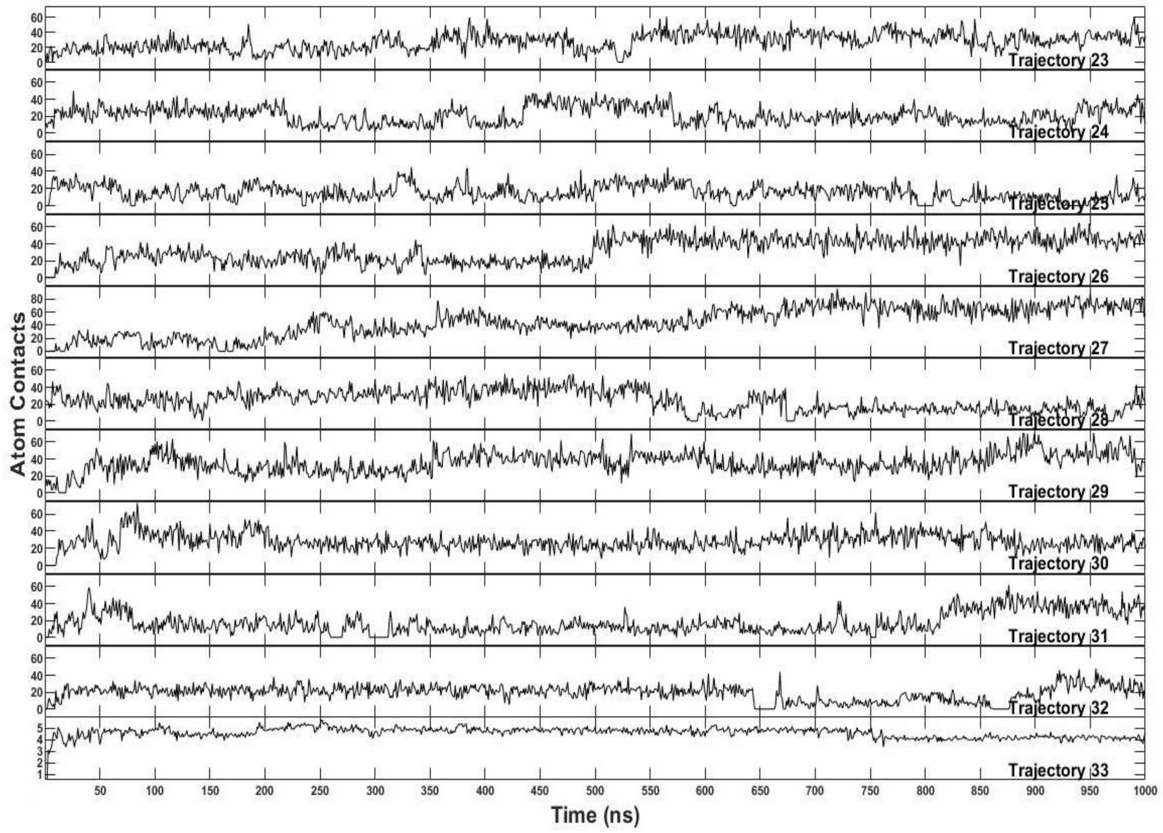


Figure F2 (continued)

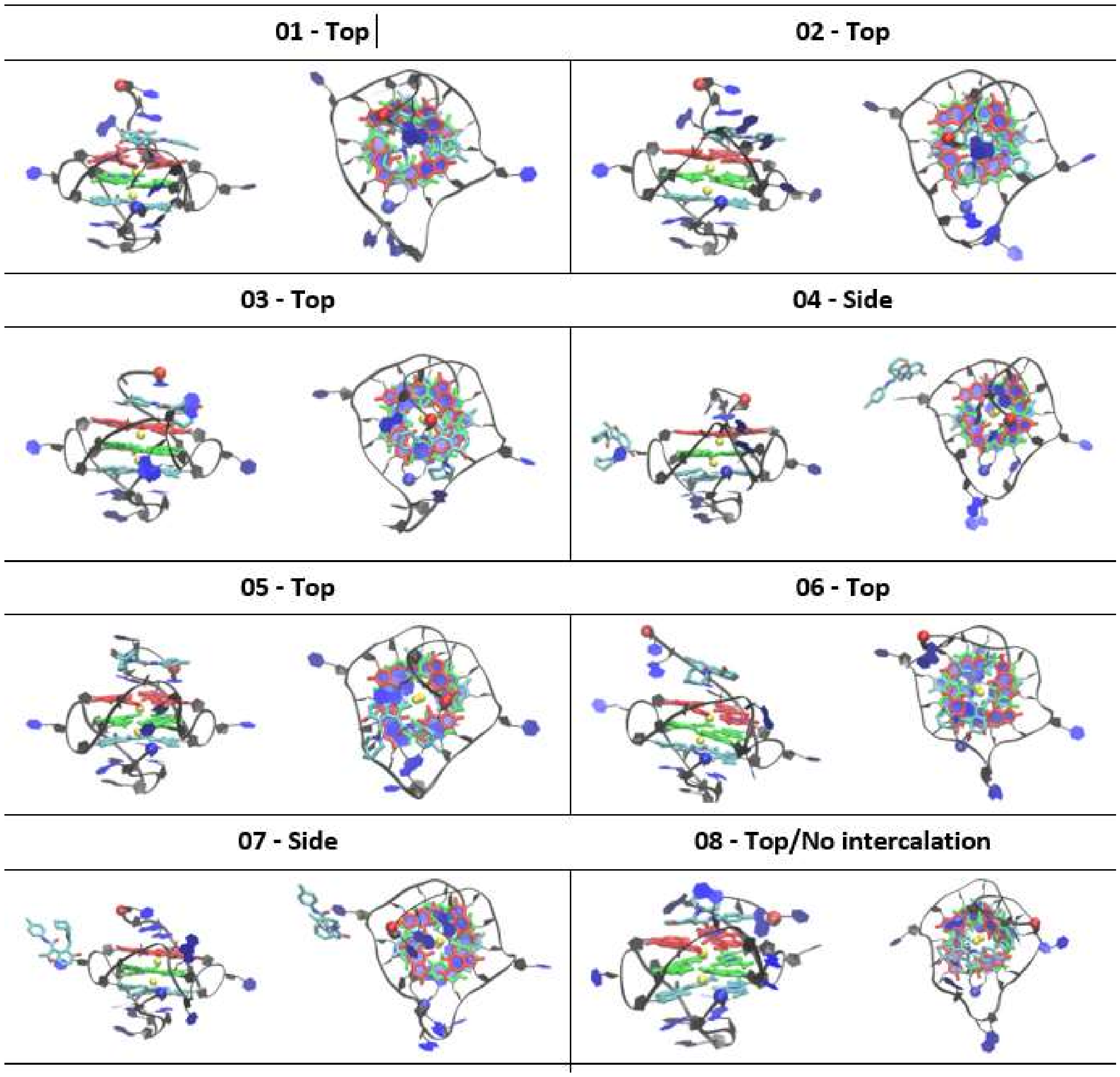


Figure F3. Last snapshots. Last snapshot of the 33 free ligand binding simulations of DBD1 to the Pu24 G-quadruplex.

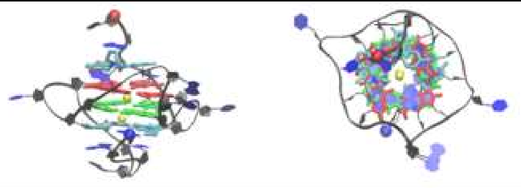
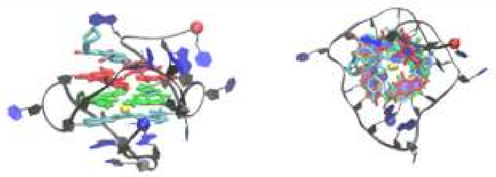
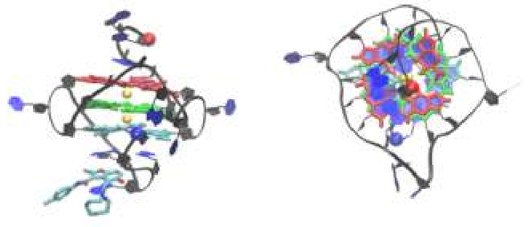
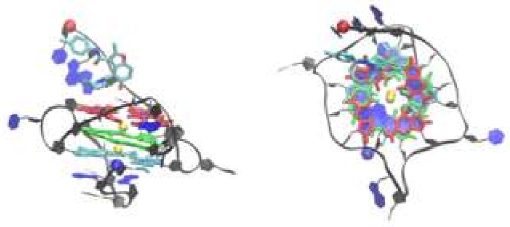
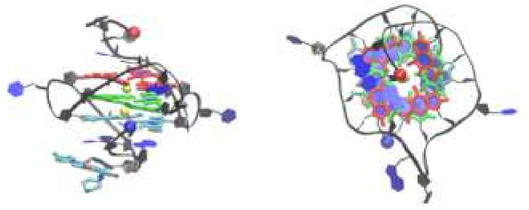
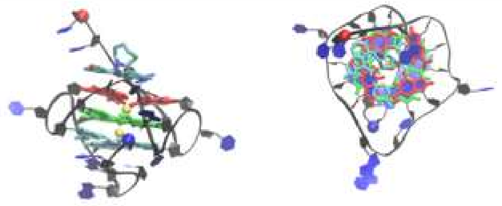
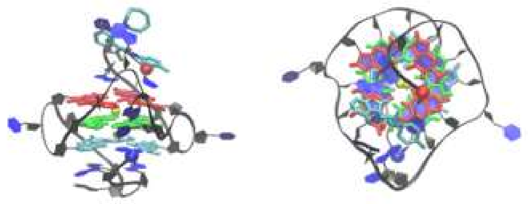
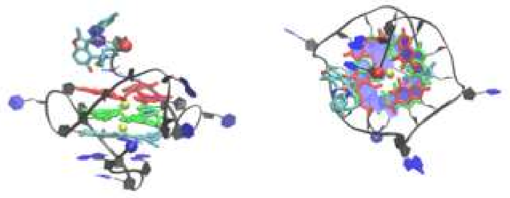
09 – Top	10 - Top/No intercalation
	
11 - Bottom	**12 - Top/No intercalation**
	
13 - Bottom	**14 - Top/No intercalation**
	
15 - Top	**16 - Top**
	

Figure F3 (continued)

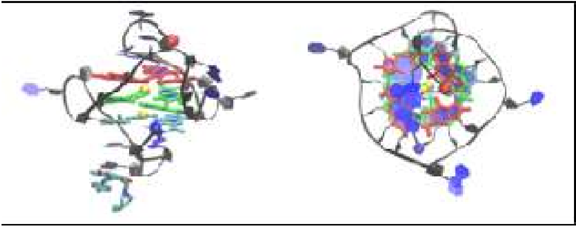
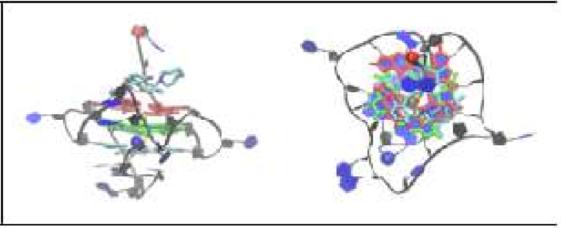
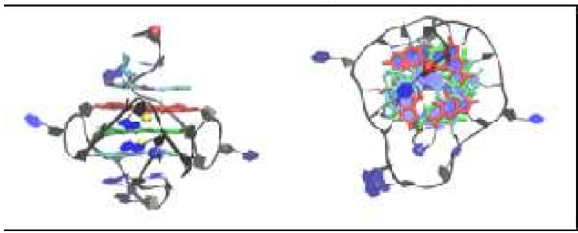
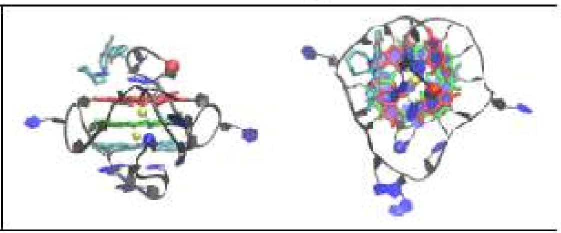
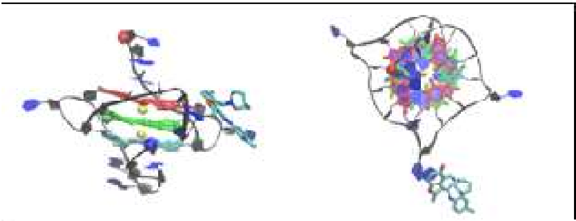
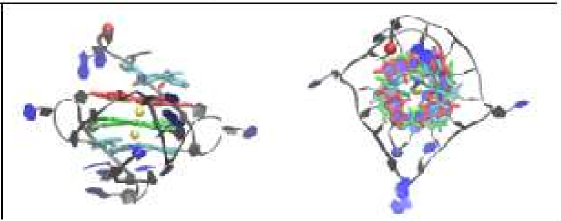
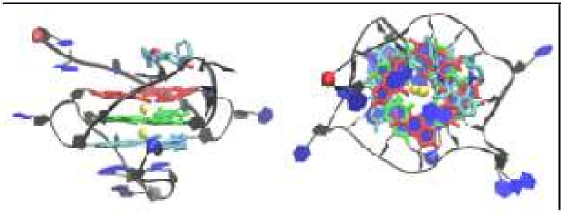
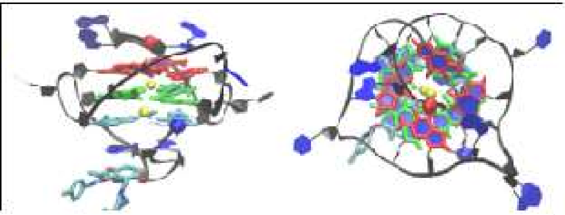
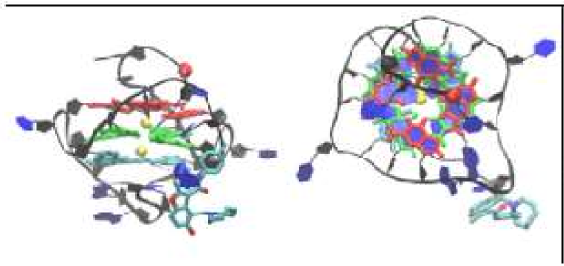
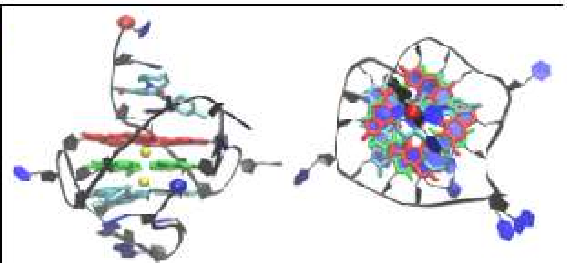
17 - Bottom	18 - Top
	
19 – Top	**20 – Top**
	
21 - Side	**22 - Top/No intercalation**
	
23 - Top/No intercalation	**24 - Bottom**
	
25 - Bottom/Side	**26 - Top**
	

Figure F3 (continued)

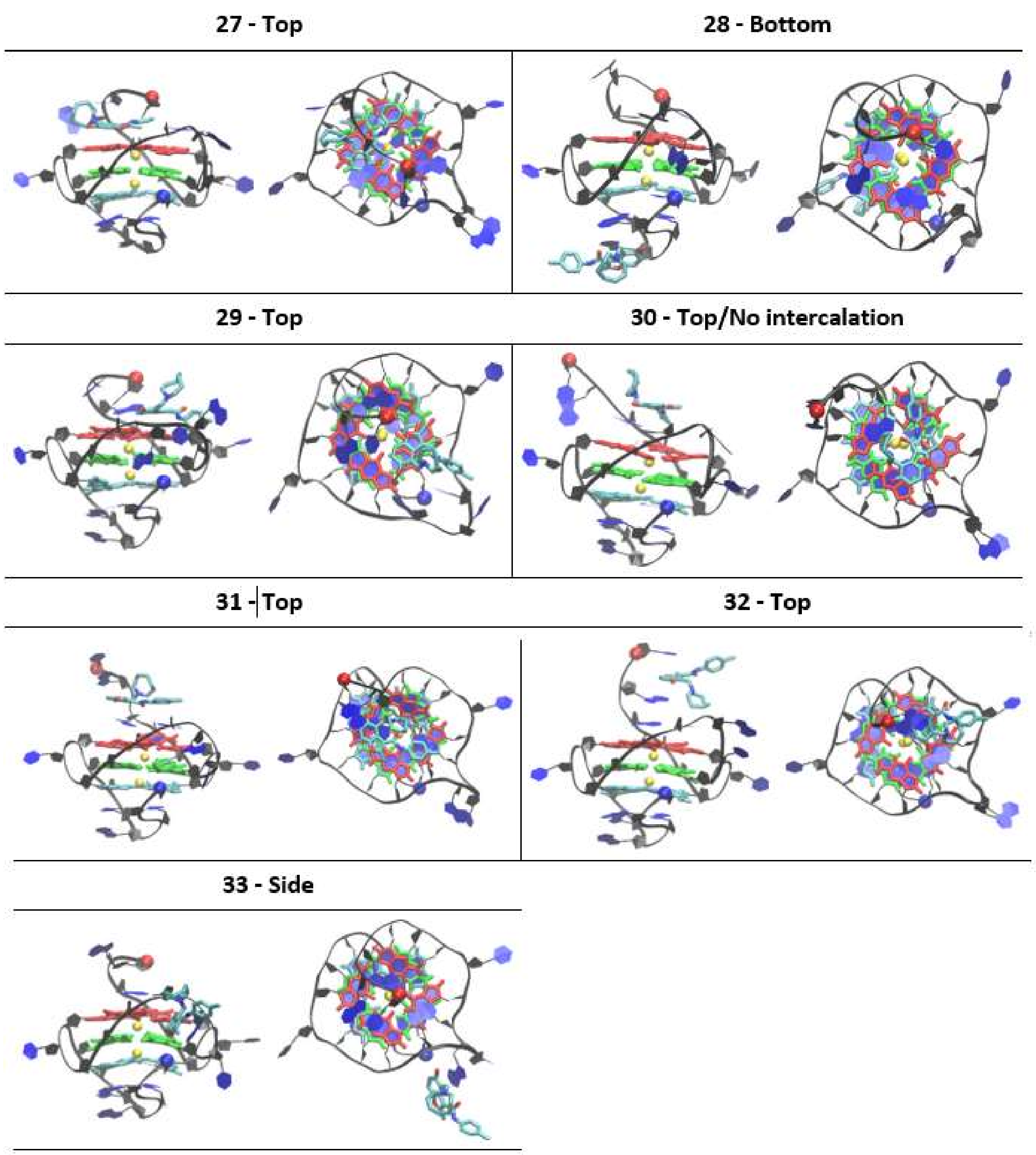


Figure F3 (continued)

Appendix G

Trajectory Snapshots

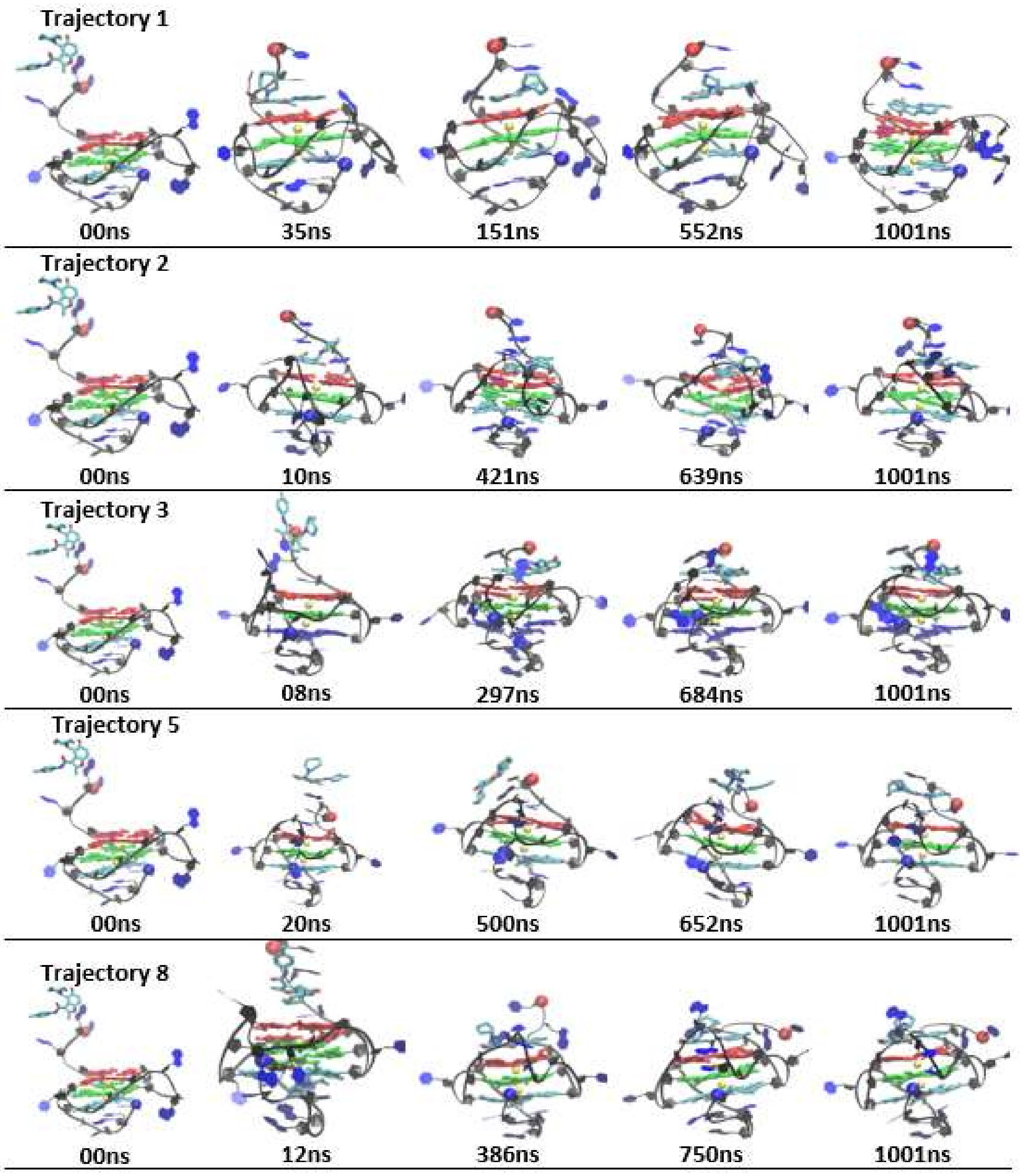


Figure G1. Snapshots of trajectories with top final binding poses. Trajectories of the primary binding pathway of DBD1 to the top site of the Pu24 G-quadruplex.

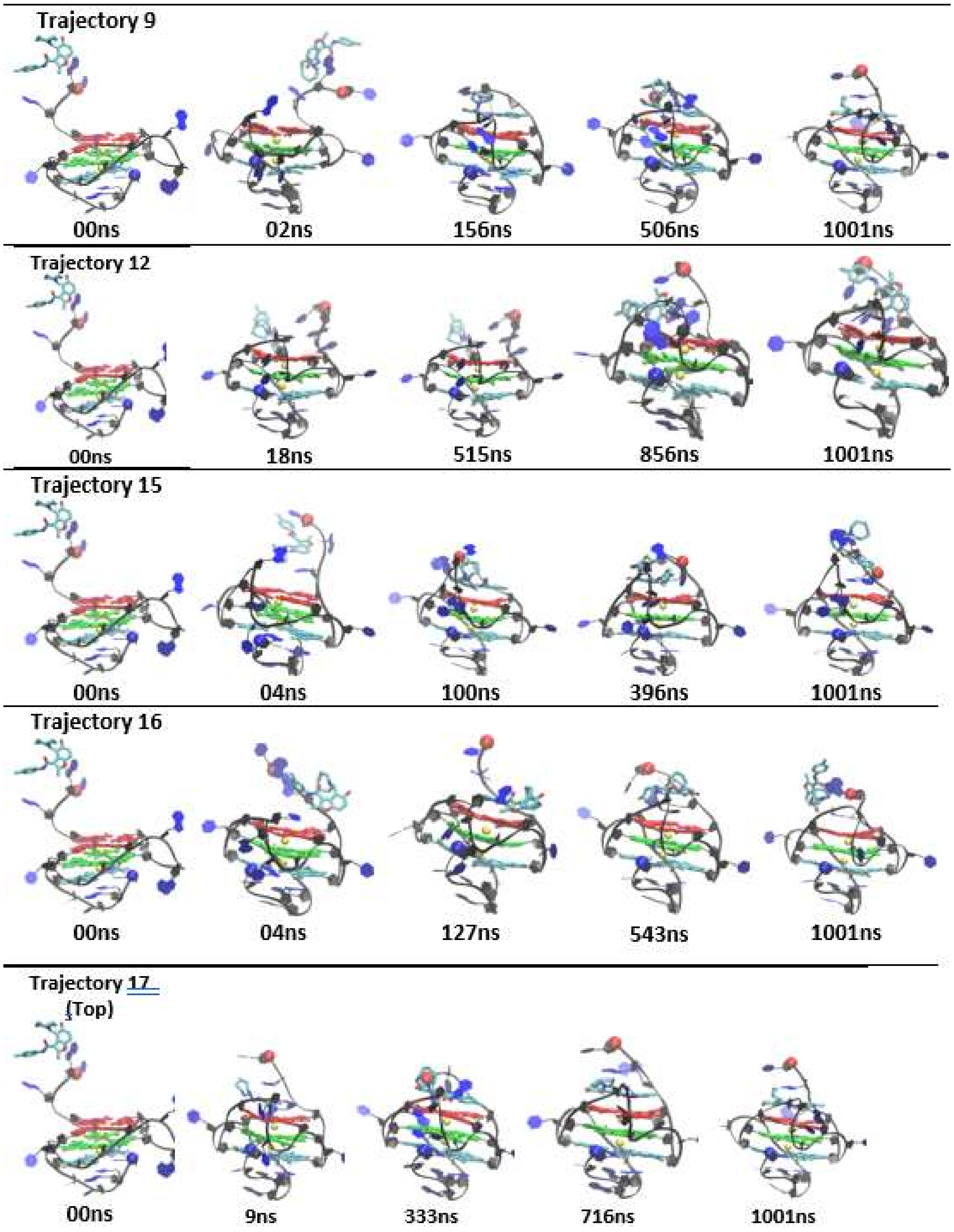


Figure G1 (continued)

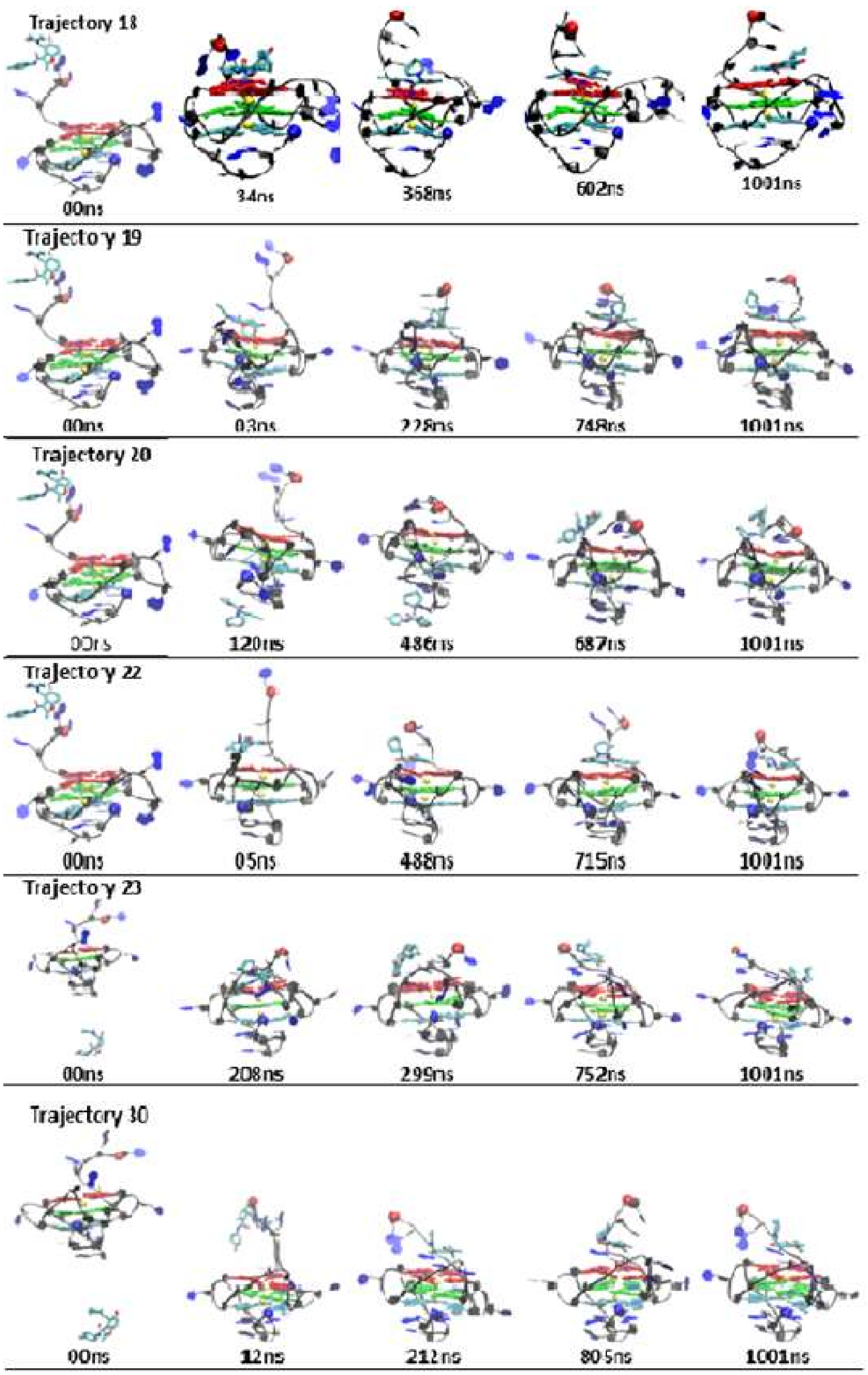


Figure G1 (continued)

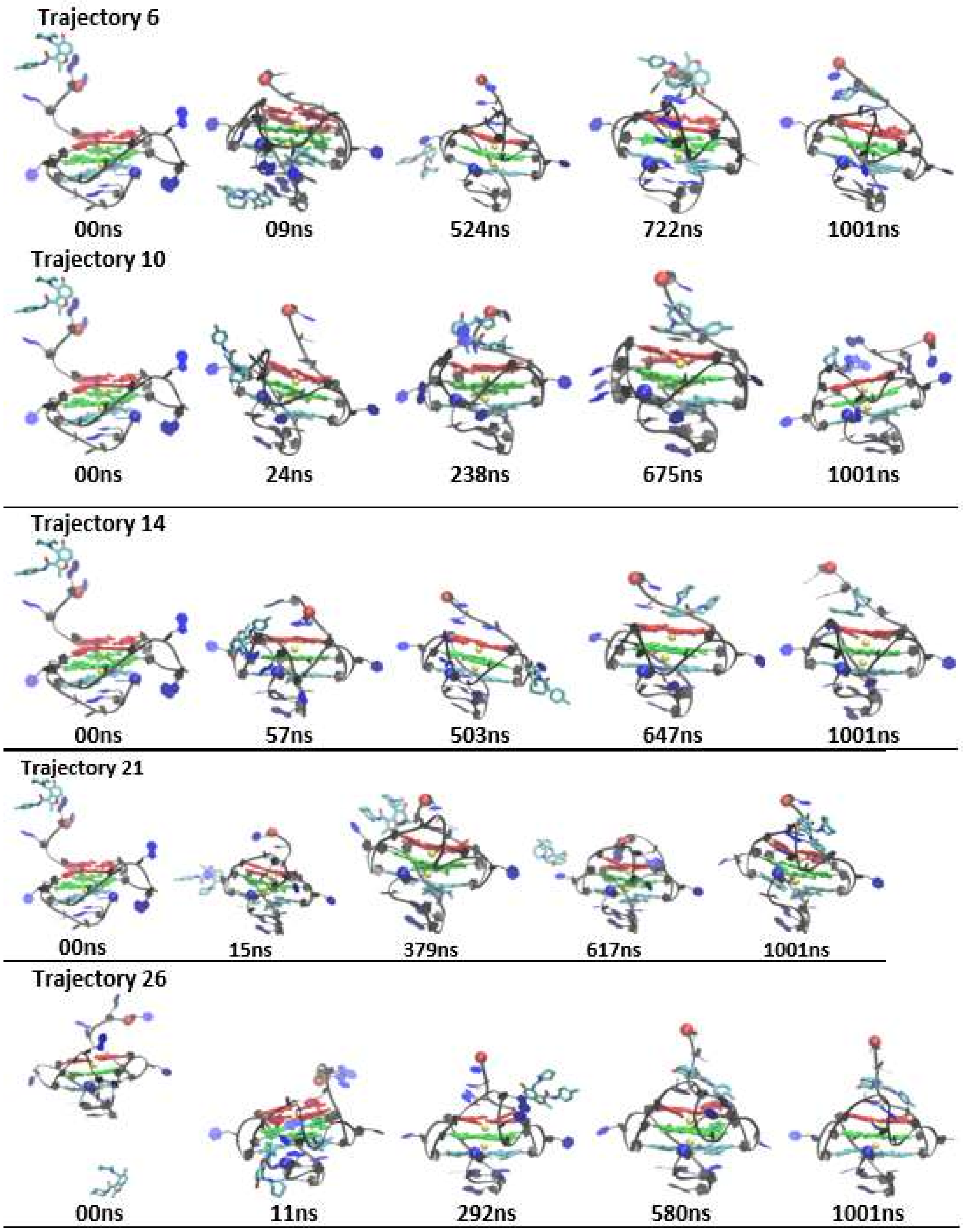


Figure G2. Snapshots of trajectories with top final binding poses with side transition. Trajectories of the secondary binding pathway of DBD1 to the top site of the Pu24 G-quadruplex via a side binding.

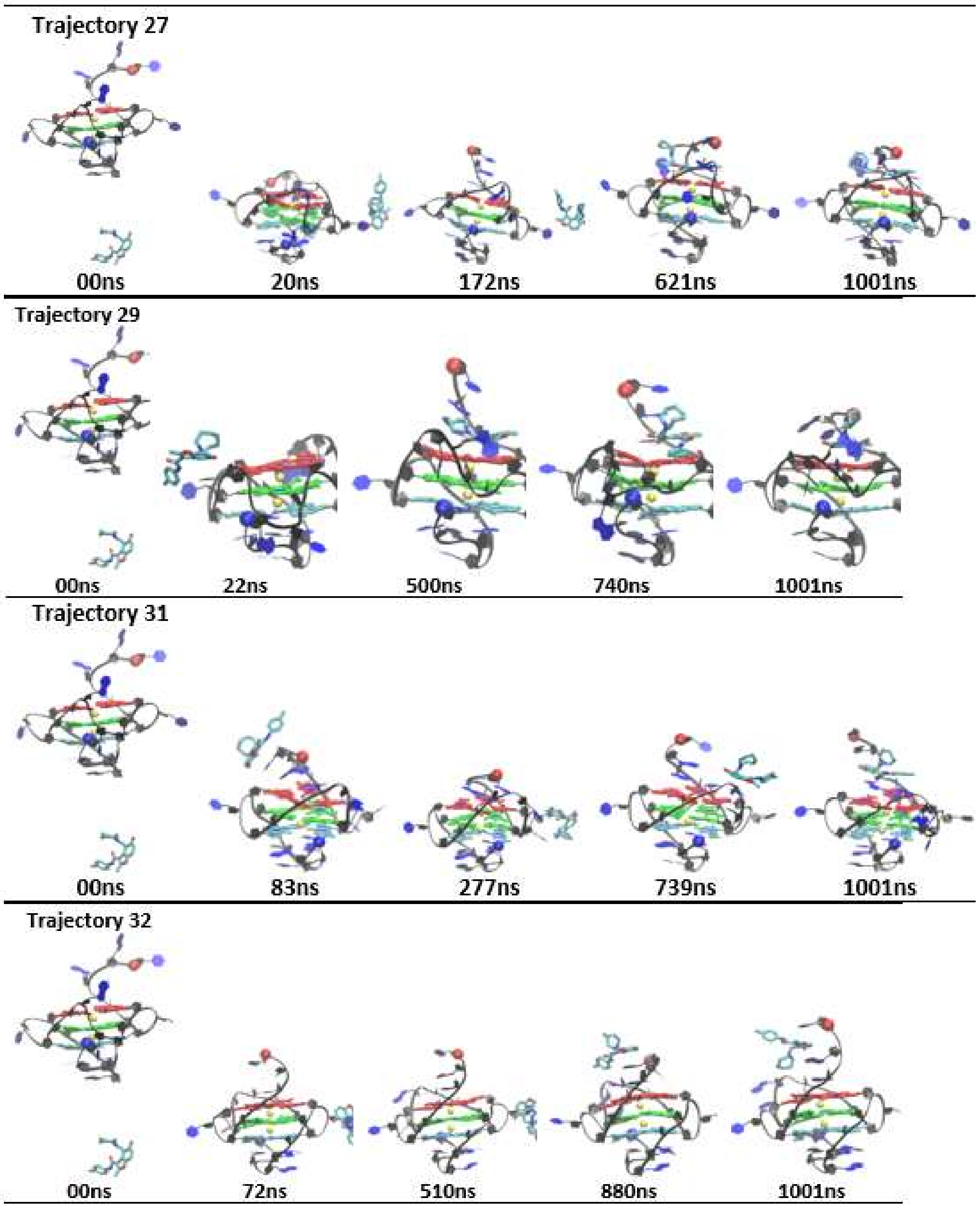


Figure G2 (continued)

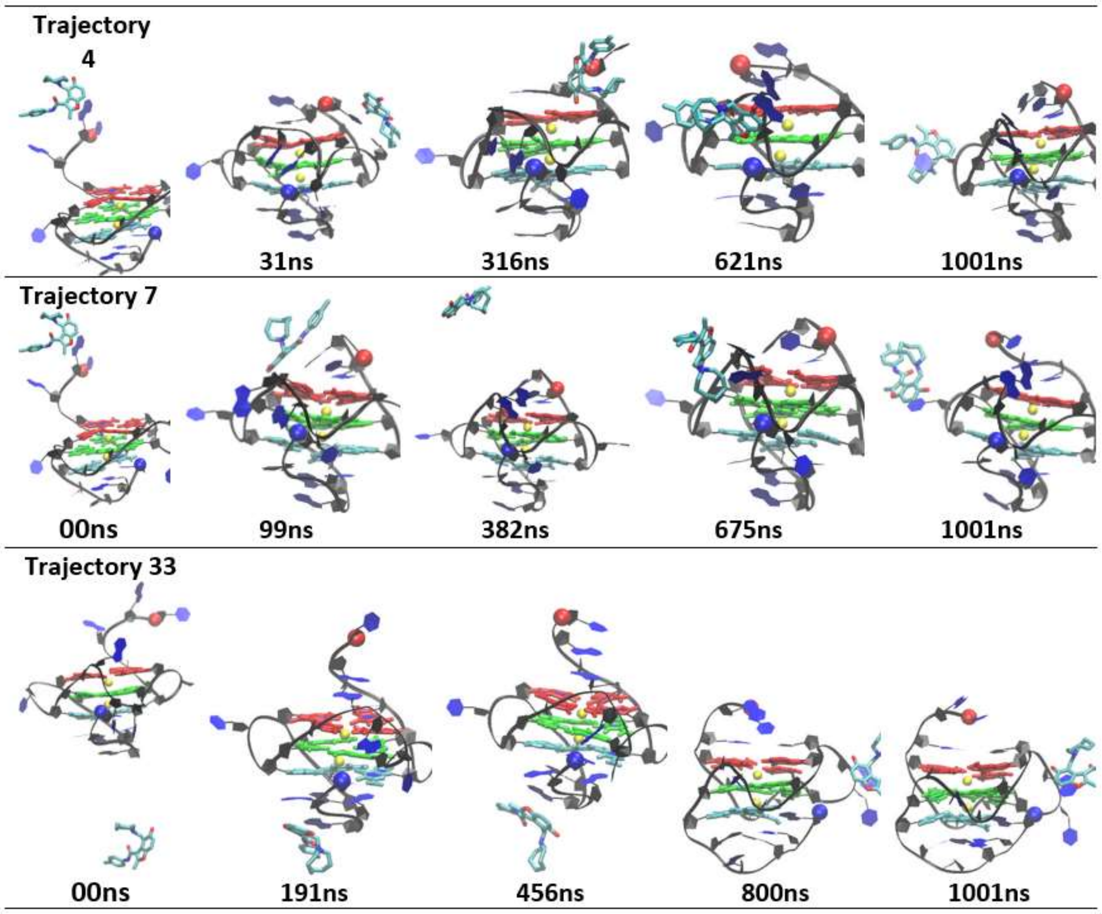


Figure G3. Snapshots of trajectories to side final binding poses. Trajectories of the binding pathway of DBD1 to the side site of the Pu24 G-quadruplex via a bottom/top binding.

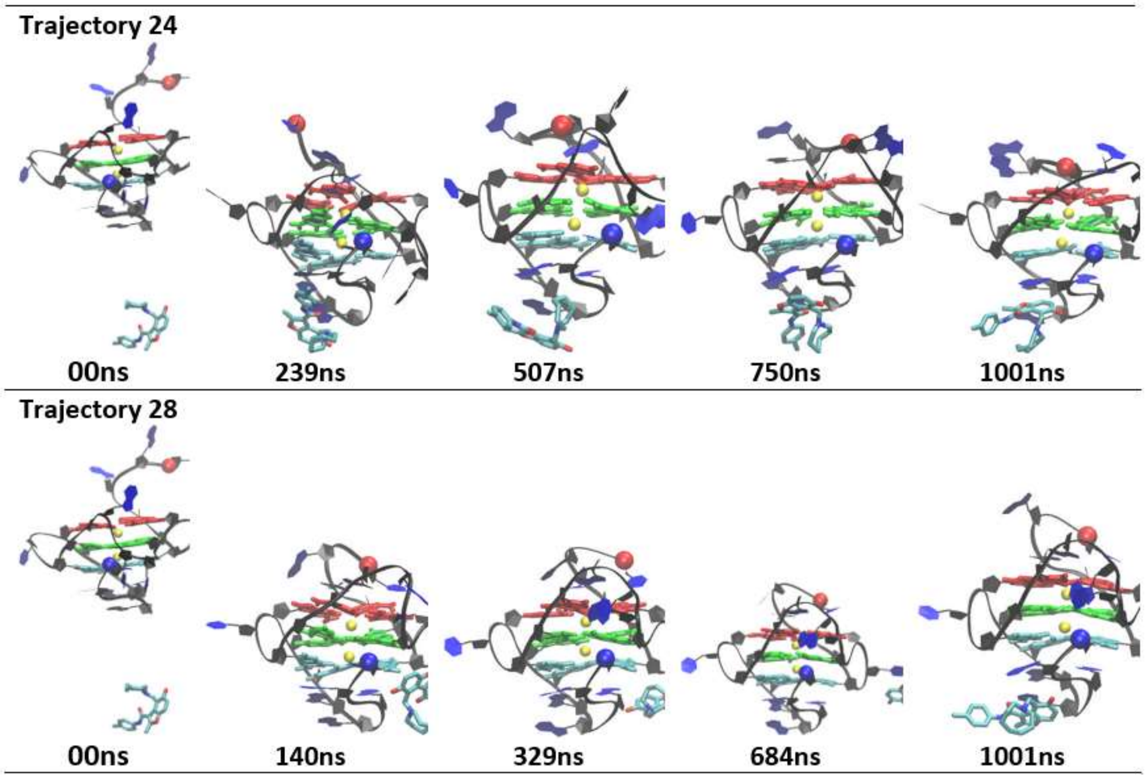


Figure G4. Snapshots of trajectories with bottom final binding poses. Trajectories of the binding pathway of DBD1 to the bottom site of the Pu24 G-quadruplex.

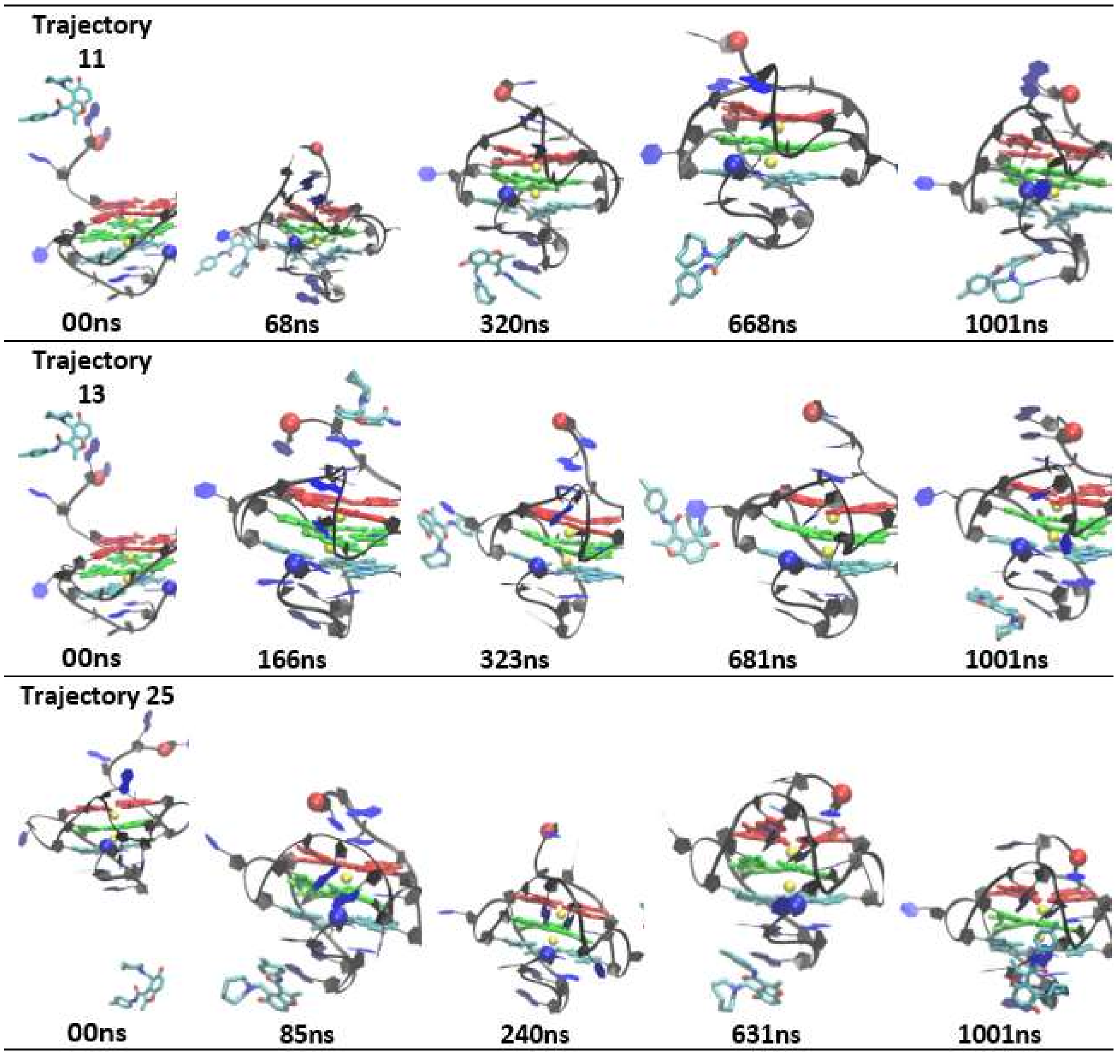


Figure G5. Snapshots of trajectories with bottom final binding poses with side transition. Trajectories of the binding pathway of DBD1 to the bottom site of the Pu24 G-quadruplex via side binding.