

MSc in Applied Bioinformatics and Data Analysis

MASTER'S THESIS

Automating the multidimensional clustering of data derived from a Principal Component Analysis of molecular dynamics trajectories using HDBSCAN.

ANASTASIOS MELAS, R.N. 682

Supervisor: Nicholas M. Glykos,
Associate Professor, Laboratory of Structural and Computational Biology,
Democritus University of Thrace

Alexandroupolis, October, 2026

Δηλώνω ότι η παρούσα εργασία με τίτλο “Automating the multidimensional clustering of data derived from a Principal Component Analysis of molecular dynamics trajectories using HDBSCAN.” είναι πρωτότυπη και πραγματοποιήθηκε από εμένα προσωπικά, μεταπτυχιακό φοιτητή του Τμήματος Μοριακής Βιολογίας και Γενετικής, με Αρ. Μητρώου 682. Βεβαιώνω ότι κατά την εκπόνηση της εργασίας και τη συγγραφή της τηρήθηκαν τα προβλεπόμενα από το νόμο, καθώς και ότι ακολουθήθηκαν πλήρως οι αρχές της ακαδημαϊκής ηθικής και δεοντολογίας, οι οποίες απαγορεύουν την παραποίηση των αποτελεσμάτων, την κατάχρηση της διανοητικής ιδιοκτησίας άλλων και τη λογοκλοπή.

Αναστάσιος Μελάς

Αλεξανδρούπολη, Οκτώβριος, 2026

Table of Contents

CHAPTER 1: Introduction and Theoretical Background	1
1.1 The Free Energy Landscape and the Limitations of Traditional Molecular Dynamics Analysis	1
1.2 The Limitations of Cartesian Coordinates and the Need for Internal Representations	1
1.3 Mathematical Foundation of dPCA: Removal of Circular Discontinuity.....	1
1.4 The Distorting Nature of Low-Dimensional Projections	2
1.5 Density-Based Clustering in Multidimensional Space.....	2
CHAPTER 2: The Architecture of clusterMD: An Automated Analysis Pipeline.....	4
2.1 Memory Management and Data Ingestion (I/O Module).....	4
2.2 Density-Based Clustering (HDBSCAN Clustering).....	5
2.3 Population-Derived Free-Energy Estimates and Observed Label Changes	5
2.4 Automated Structural Extraction and Characterization Using carma	6
2.5 Visualization and Data Export.....	8
CHAPTER 3: Software Implementation and Computational Optimization	10
3.1 Development Stack and Computational Core.....	10
3.2 Computational Resource Management and Scalability	10
3.3 User Interface, Diagnostic Mode, and Reproducibility	11
3.4 Environment Deployment and Open Source	12
CHAPTER 4: Application Examples and Computational Evaluation	13
4.1 Datasets and Analysis Settings	13
4.2 Multidimensional Clustering and Population Summaries of CLN025	13
4.3 Structural Characterization and Optional RMSD Filtering	14
4.4 Observed Label Changes and Chronological Inspection	15
4.5 Execution on the Larger CDP-1 Dataset	17
4.6 Sensitivity to Temporal Subsampling	17
4.7 Qualitative Comparison of Structural Ensembles	18
5.1 Methodological Limitations and Heuristics	19
5.2 Potential Applications in Computational Structural Biology	20
5.3 Future Work and Pipeline Expansion.....	20
Bibliography.....	22
Appendix: clusterMD Source Code	24

Table of Figures

Figure 2.1: Overview of the clusterMD workflow.....	4
Figure 2.2: Structural characterization workflow	7
Figure 2.3: Structural ensembles before and after RMSD filtering	8
Figure 4.1: PC1–PC2 population-derived free-energy landscape of CLN025.....	13
Figure 4.2: CLN025 projection colored by multidimensional assignments.....	13
Figure 4.3: Structural superposition of CLN025 Cluster 0.....	14
Figure 4.4: CLN025 ensembles before and after RMSD filtering	14
Figure 4.5: Structural comparison of CLN025 Clusters 0, 3, and 19	15
Figure 4.6: PCoA representation and observed cluster connectivity	16
Figure 4.7: CDP-1 ensembles before and after RMSD filtering	17
Figure 4.8: Structural superpositions from the comparator analysis	18
Figure 4.9: Structural superpositions from clusterMD	18

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Associate Professor Nicholas M. Glykos, for providing the foundational idea for this project. His continuous guidance, unwavering support, and structural insights were instrumental throughout the entire development of clusterMD. I also wish to extend my heartfelt thanks to my family, friends, and everyone who stood by me, offering their patience and encouragement over the past months.

During the preparation of this thesis and the optimization of the clusterMD codebase, generative AI tools were utilized to assist with code debugging, the refinement of technical English prose, and the generation of graphical abstract concepts. All architectural logic, multidimensional data analysis, and final scientific interpretations remain the sole intellectual property and responsibility of the author.

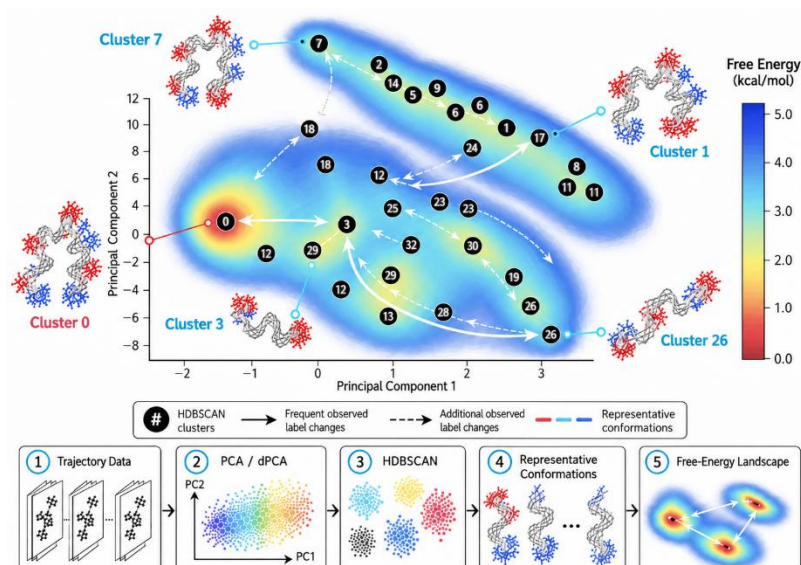
Abstract

Molecular dynamics (MD) simulations generate large conformational datasets whose analysis requires methods for organizing sampled configurations and connecting statistical patterns to molecular structures. This thesis presents *clusterMD*, an automated workflow for density-based clustering and structural characterization of multidimensional principal component analysis (PCA) and dihedral principal component analysis (dPCA) projections. The workflow applies HDBSCAN in the retained projection space, preserves original trajectory-frame identifiers, and reports cluster populations while retaining unassigned configurations. For each cluster, an observed frame nearest the arithmetic centroid is selected as a representative. Structural extraction, fitting, averaging, and optional downstream RMSD filtering are performed using the external *carma* program without changing the original clustering assignments. Population-derived free-energy visualizations and chronological summaries of observed label changes support exploratory interpretation. Applications to CLN025 and CDP-1 illustrate the integration of clustering, structural inspection, and computational reporting for large peptide datasets. A temporal-subsampling comparison examines sensitivity under the tested settings. The resulting clusters describe density structure in the selected coordinate representation; thermodynamic and kinetic interpretations require additional sampling and validation evidence. *clusterMD* provides a reproducible framework for organizing molecular-dynamics projection data and examining their associated structural ensembles. The source code and installation documentation are available at <https://github.com/tasosmelas/md-hdbscan-clustering>.

Keywords

Molecular dynamics, principal component analysis, dihedral principal component analysis, HDBSCAN, conformational clustering, structural characterization, population-derived free energy.

Graphical Abstract



Graphical abstract - Overview of *clusterMD*: upstream PCA/dPCA projections, HDBSCAN assignments, population-derived visualizations, observed label changes, and structural extraction. The illustration is schematic and does not establish kinetic pathways.

CHAPTER 1: Introduction and Theoretical Background

1.1 The Free Energy Landscape and the Limitations of Traditional Molecular Dynamics Analysis

The biological functionality of macromolecules is not a static property of a single structure, but is linked to the thermodynamic equilibrium and kinetics of their conformational transitions. During their biological activity, molecular systems explore a multidimensional free-energy landscape (Dinner et al., 2000). Free-energy basins describe regions of relatively high equilibrium probability, while barriers separating them influence conformational transition rates and folding pathways. Identifying a basin as a metastable state additionally requires evidence that the system remains within it for a sufficiently long time relative to its internal relaxation.

The temporal evolution of these systems is tracked using Molecular Dynamics (MD) simulations (Lindorff-Larsen et al., 2011). However, a trajectory containing T saved frames of N atoms can be represented as a Cartesian coordinate matrix in $\mathbb{R}^{3N \times T}$, the direct processing of which can entail substantial computational costs. Software such as *carma* (Glykos, 2006) effectively handles the computational aspect through frame-by-frame analysis algorithms, allowing the application of dimension reduction methods, such as Principal Component Analysis (PCA), to isolate the dominant collective molecular motions. PCA identifies linear combinations of coordinates that maximize the represented variance. However, the interpretation of these components depends on the coordinate representation and structural alignment, and retaining only a few components can obscure distinctions between conformational regions.

1.2 The Limitations of Cartesian Coordinates and the Need for Internal Representations

For Cartesian PCA of internal molecular motions, global translations and rotations are usually removed by least-squares structural alignment, for example using the Kabsch algorithm, before analyzing the Cartesian coordinates (x , y , z).

For systems undergoing large-amplitude conformational changes, the separation of overall rotation from internal motion can depend on the structural alignment. This can introduce alignment-dependent contributions to the covariance matrix and complicate the interpretation of the resulting principal components (Altis et al., 2008).

An alternative is to use internal coordinates, such as the backbone dihedral angles ϕ and ψ , which are invariant under global translation and rotation. Angular periodicity must nevertheless be handled explicitly: -180° and $+180^\circ$ describe the same orientation, although treating them as ordinary real-valued coordinates assigns them a separation of 360° . Direct covariance analysis of untransformed angles can therefore introduce artificial discontinuities and distort the representation of conformational variation.

1.3 Mathematical Foundation of dPCA: Removal of Circular Discontinuity

To take advantage of the invariance of the internal coordinates without introducing metric discontinuities at the boundaries of $[-\pi, \pi]$, our approach employs the method of Dihedral

Principal Component Analysis (*dPCA*). Each circular variable φ_n is projected nonlinearly onto the unit circle via the trigonometric transformation:

$$\begin{aligned} q_{2n-1} &= \cos(\varphi_n) \\ q_{2n} &= \sin(\varphi_n) \end{aligned}$$

This transformation embeds N angular variables in a $2N$ -dimensional Euclidean space. The sine and cosine coordinates vary continuously across the angular boundary, allowing PCA to be applied without the discontinuity introduced by treating angles as ordinary real-valued variables (Altis et al., 2008). The covariance matrix $C \in \mathbb{R}^{2N \times 2N}$ is now computed without artificial constraints.

An equivalent complex representation is:

$$z_n = e^{i\varphi_n} = q_{2n-1} + iq_{2n}$$

The complete sine–cosine representation preserves each angular variable modulo 2π . This property does not guarantee that a truncated principal-component representation uniquely identifies every conformation or preserves every free-energy basin. Distinct configurations can overlap after dimensionality reduction, and backbone dihedral angles do not specify all atomic degrees of freedom (Altis et al., 2008).

1.4 The Distorting Nature of Low-Dimensional Projections

A common practice in the literature is to project the free-energy landscape onto the first two principal components (2D-PCA landscape) and then perform clustering on this plane.

Projection onto only two principal components can cause distinct conformational regions to overlap and can obscure features of the underlying free-energy landscape (Altis et al., 2008). To reduce dependence on a two-dimensional display, *clusterMD* performs clustering in the retained multidimensional PCA/dPCA coordinate space. The number of retained components is specified by the user through the `--dim` option, with all available components used when `--dim` is set to 0. Two-dimensional plots are used for visualization rather than defining the clustering space, unless the analysis itself is restricted to two components. The retained dimensionality remains a methodological choice whose influence on clustering should be assessed.

1.5 Density-Based Clustering in Multidimensional Space

Conformational distributions in PCA/dPCA space can be anisotropic and non-convex, making clustering methods that favor compact groups around centroids, such as k -means, less suitable for some molecular-dynamics datasets. Density-based methods provide an alternative that can identify clusters with irregular shapes while leaving sparsely sampled configurations unassigned (Melvin et al., 2016).

A physical motivation for examining conformational density follows from the relationship between equilibrium probability and free energy. For a selected coordinate representation q , the corresponding free-energy profile can be expressed as:

$$F(q) = -k_B T \ln P(q) + C,$$

Here, k_B is the Boltzmann constant, T is the absolute temperature, and $P(q)$ is the equilibrium probability density in the chosen coordinates. Under representative equilibrium sampling, regions of high probability density correspond to low values of this free-energy profile. Estimates obtained from a finite trajectory nevertheless depend on sampling quality and the coordinate representation.

Motivated by this relationship, *clusterMD* applies HDBSCAN to the retained PCA/dPCA coordinates. **HDBSCAN** constructs a hierarchy of density-based clusters and selects clusters according to the chosen selection method (Campello et al., 2013; McInnes et al., 2017). Configurations that are not assigned to a selected cluster retain the label -1. The resulting groups are density-defined conformational clusters; establishing their thermodynamic or kinetic significance requires additional structural and sampling evidence. Likewise, unassigned configurations should not automatically be interpreted as thermal noise or physical transition states.

The objective of this thesis is to automate the analysis of multidimensional PCA/dPCA projections through density-based clustering, cluster population summaries, extraction of representative observed structures, and inspection of changes between assigned labels along trajectory order. *clusterMD* integrates these operations into a reproducible workflow to support structural interpretation of molecular-dynamics ensembles.

CHAPTER 2: The Architecture of *clusterMD*: An Automated Analysis Pipeline

The concepts introduced in Chapter 1 are implemented in *clusterMD* as an automated workflow comprising five analytical stages: projection ingestion, density-based clustering, population and observed-label-change analysis, structural characterization, and visualization and reporting. The current implementation organizes these stages through dedicated functions within a single executable Python script. PCA/dPCA coordinates are generated upstream, for example using *carma* or *grcarma*, and supplied to *clusterMD* for clustering and downstream analysis. The following sections describe how the workflow connects the projected coordinates to their original trajectory frames and associated molecular structures.

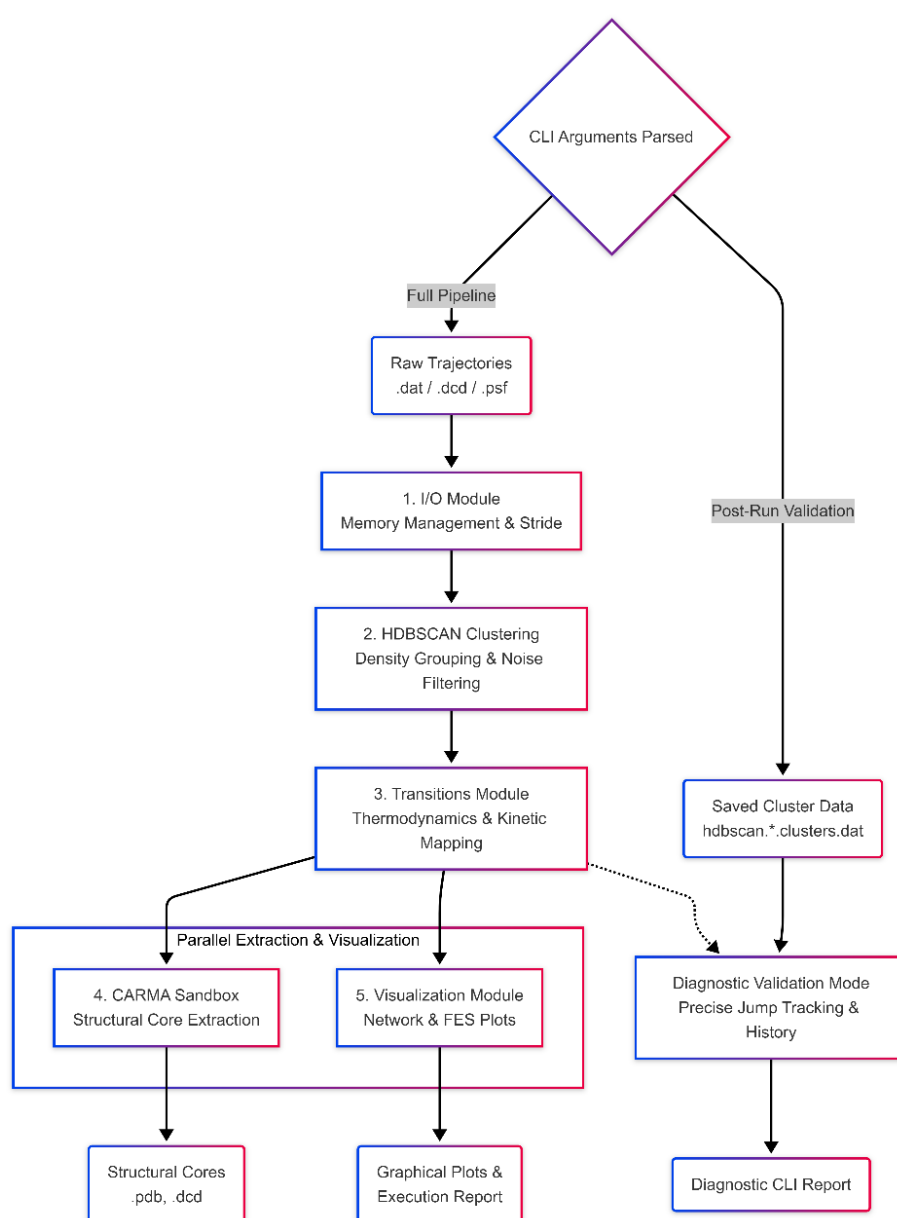


Figure 1.1 - Overview of the *clusterMD* workflow, from ingestion of upstream PCA/dPCA projections to density-based clustering, population and observed-label-change analysis, structural characterization, and visualization and reporting. A separate diagnostic mode inspects saved assignments without repeating clustering or structural extraction.

2.1 Memory Management and Data Ingestion (I/O Module)

Large projection datasets can require substantial memory during ingestion and clustering. The `ingest_trajectory_data` routine estimates the required array size from the input file and preallocates a NumPy array before loading the selected rows. This reduces repeated array allocation during ingestion, although the selected dataset and clustering calculations still require sufficient available memory.

The input projection file contains trajectory-frame identifiers in its first column and PCA/dPCA coordinates in the remaining columns. The `--step` option subsamples rows during ingestion while preserving the original frame identifiers. It therefore applies an additional sampling stride to the projection file rather than replacing any stride used during upstream projection generation. These identifiers are subsequently used to map cluster assignments back to the original trajectory. Subsampling reduces the number of analyzed configurations, but its effect on cluster populations and assignments must be assessed rather than assumed negligible.

The ingestion routine checks sampled records for valid numerical values and positive integer frame identifiers, and records malformed rows that are skipped. Duplicate or non-increasing sampled frame identifiers are flagged because they can compromise structural mapping or chronological interpretation.

2.2 Density-Based Clustering (HDBSCAN Clustering)

Following ingestion, the `execute_clustering_algorithm` routine applies HDBSCAN through the `fast_hdbscan`¹ Python package to the retained multidimensional PCA/dPCA coordinates. The resulting assignments identify density-defined conformational clusters in the selected projection space.

Configurations not assigned to a selected cluster retain the label `-1` and are reported as unassigned/noise. They are not forcibly assigned to a neighboring cluster or discarded from the assignment output. Assigned clusters are subsequently relabeled in descending order of population, so that Cluster 0 denotes the most populated cluster. This relabeling changes the identifiers without changing cluster membership.

The `--method` option exposes the Excess of Mass (eom) and leaf cluster-selection methods. Excess of Mass generally selects broader clusters according to stability within the density hierarchy, whereas leaf selection provides a finer partition by selecting terminal clusters. This algorithmic stability should not be equated with physical or kinetic metastability.

The minimum cluster size and minimum samples parameters can be specified manually through `--size` and `--samples`. When `--auto` is enabled, a frame-count heuristic sets these parameters using the number of analyzed projection rows. This provides a practical starting point rather than a universally optimal parameterization; sensitivity to these settings remains relevant to scientific interpretation.

¹ The `fast_hdbscan` package is available via the Python Package Index (PyPI) and GitHub.

2.3 Population-Derived Free-Energy Estimates and Observed Label Changes

The `calculate_populations_and_energies` routine counts the configurations assigned to each cluster and calculates relative population-derived free-energy estimates at the selected temperature. The most populated assigned cluster defines the zero reference. These estimates describe differences in sampled cluster populations, with thermodynamic interpretation depending on representative equilibrium sampling.

$$\Delta F_i = -RT \ln \left(\frac{n_i}{n_{\max}} \right),$$

Here, n_i is the number of analyzed configurations assigned to cluster i , $n_{\max}$ is the population of the most populated assigned cluster, R is the molar gas constant, and T is the selected temperature. These population-derived quantities are distinct from free-energy barriers between clusters.

Chronological information is summarized by the `map_observed_label_changes` routine. Unassigned labels are omitted from the counting sequence, and a change is recorded whenever two consecutive remaining assigned labels differ. Counts are aggregated as undirected cluster pairs, combining changes in both directions. Consequently, a sequence such as A, -1, -1, B contributes one observed change for the pair A-B, whereas A, -1, A contributes none. These summaries describe observed label changes and do not provide kinetic rates, Markov transition probabilities, or proof of folding pathways.

A diagnostic mode allows an existing assignment file to be inspected without repeating clustering or structural extraction. The user can display the most frequent cluster-pair changes or inspect a specified pair. For individual events, the output reports the preceding and following assigned configurations, their original frame identifiers and retained coordinates, and any intervening unassigned configurations.

2.4 Automated Structural Extraction and Characterization Using *carma*

Structural interpretation requires mapping the clusters identified in projection space back to observed molecular configurations. *clusterMD* performs this mapping using the preserved trajectory-frame identifiers and invokes the external *carma* executable for trajectory extraction, structural fitting, averaging, and RMSD analysis. Each cluster is processed in a separate temporary working directory to reduce interference between intermediate files.

The original DCD trajectory and PSF topology are referenced through absolute paths, avoiding duplication of the full input files within each temporary directory. Extracted and fitted trajectories still require additional storage. Before processing the clusters, the workflow checks structural-input compatibility, the requested atom and segment selections, and trajectory-frame bounds where these can be determined.

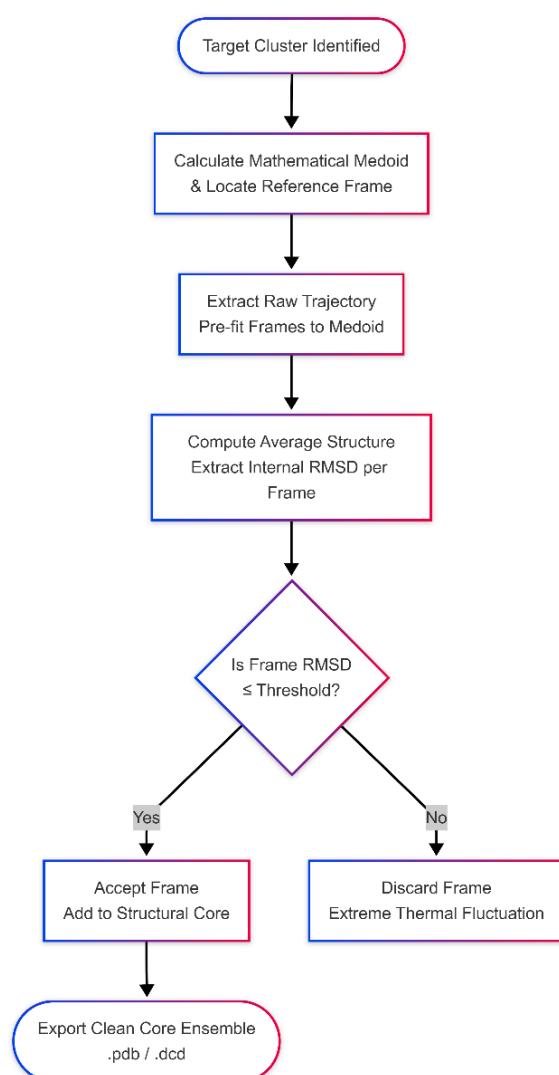


Figure 2.2 - Structural characterization of an assigned cluster. A nearest-centroid frame provides the fitting reference. Optional RMSD filtering uses the initial fitted average; the retained ensemble is then reconstructed. HDBSCAN labels and populations remain unchanged.

Structural characterization follows six main steps, with additional processing when RMSD filtering is enabled:

1. **Atom selection and trajectory extraction:** The workflow attempts to recover the structural atom selection from `last_carma_run.log`. If this information is unavailable or cannot be parsed, it uses the default backbone atom selection comprising N, CA, C, and O. An optional `--segid` argument restricts structural processing to the specified segment. The original frame identifiers of cluster members are supplied to *carma* to extract the corresponding trajectory configurations.
2. **Representative selection and structural fitting:** The arithmetic centroid of each cluster is calculated in the retained PCA/dPCA coordinate space. The observed cluster frame nearest this centroid is selected as the representative, and the initial cluster ensemble is fitted to that frame using *carma*. This representative is a nearest-centroid observed frame rather than a mathematical medoid.
3. **Average structure and RMSD evaluation:** The fitted ensemble is used to calculate an average structure and a structural superposition. RMSD values relative to the fitted ensemble average are obtained for the structural evaluation and optional filtering step.

4. **Optional RMSD filtering:** When `--rmsd` is greater than zero, frames satisfying the selected RMSD threshold relative to the initial fitted average are retained. These frames are extracted again from the original trajectory, and the filtered ensemble is fitted and its average and superposition recalculated. When filtering is disabled, the full cluster ensemble is retained. If no frames satisfy the threshold, the workflow reports a warning and retains the complete ensemble.
5. **Structural reporting:** The workflow reports the maximum RMSD to the final ensemble average and distinguishes the original HDBSCAN population from the number of retained structural frames. Because the average is recalculated after filtering, the final maximum RMSD can differ from the threshold applied during the initial filtering step.
6. **Structure export:** The workflow exports the representative observed frame as a PDB file, together with the generated average structure, superposition, and associated trajectory outputs. The representative is selected from the original cluster before filtering and is not recalculated from the retained structural subset.

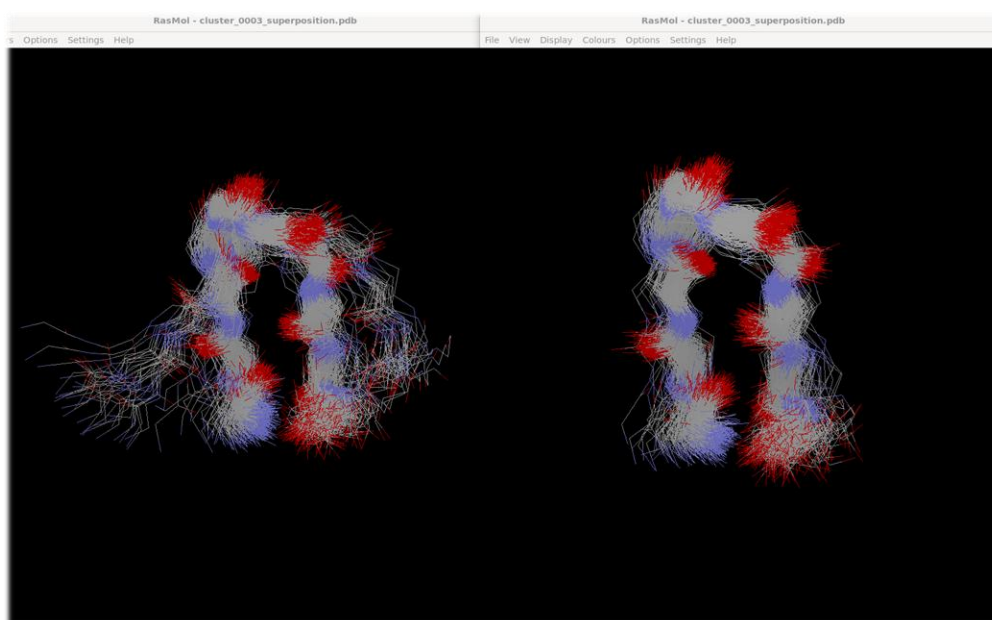


Figure 2.2 - Structural superpositions before (left) and after (right) optional RMSD filtering. The retained subset shows reduced structural dispersion; original HDBSCAN labels and populations remain unchanged.

RMSD filtering and segment selection affect downstream structural characterization only. They do not modify the input projection coordinates, original HDBSCAN assignments, or cluster populations. A filtered ensemble therefore represents a structurally selected subset of a cluster rather than a newly defined density-based cluster.

2.5 Visualization and Data Export

The visualization stage generates two-dimensional projections of the retained PCA/dPCA coordinates, colored by multidimensional cluster assignments, together with a population-derived free-energy landscape in the PC1–PC2 plane. Observed cluster-pair changes can be overlaid as descriptive connectivity, with displayed edges selected according to their counts and the `--top` setting. Where sufficient clusters are available, a principal coordinate analysis

(PCoA) representation is constructed from Euclidean distances between component-wise cluster medians in the retained projection space. These medians are distinct from the arithmetic centroids used to select representative structures.

The workflow also exports cluster assignments, including unassigned configurations, observed-label-change summaries, structural outputs, and execution reports. Reports record analysis settings, population accounting, processing outcomes, software versions, timings, and Python process memory usage at report generation. These records support reproducibility and distinguish completed analysis stages from skipped or failed operations.

CHAPTER 3: Software Implementation and Computational Optimization

The transition from theoretical design to practical application required the creation of a computational environment capable of managing the volume of data generated by Molecular Dynamics simulations. This chapter analyzes the technical architecture of the *clusterMD* computational pipeline, focusing on the development tools, memory management strategies, and mechanisms for ensuring reproducibility.

3.1. Development Stack and Computational Core

clusterMD is implemented in Python, which coordinates projection-data analysis and external structural-processing operations. The documented validation environment uses Python 3.12.3. NumPy provides array operations for coordinate handling, population accounting, and representative-frame selection, while HDBSCAN clustering is performed through the `fast_hdbscan` package. SciPy supplies pairwise-distance calculations for the PCoA representation of cluster medians, and Matplotlib generates the graphical outputs. The `psutil` library provides process-memory information and the number of logical CPU cores detected on the system.

Structural extraction, fitting, averaging, and RMSD analysis are delegated to the external *carma* executable. Python's `subprocess` module invokes *carma* within temporary working directories, while captured output and return codes support execution logging and error reporting. This organization separates intermediate structural files from the main analysis outputs.

3.2. Computational Resource Management and Scalability

Processing millions of projection rows requires attention to memory allocation and computational cost. During ingestion, *clusterMD* estimates the required array size and preallocates a NumPy array for the sampled records. This reduces repeated allocation during loading, although the selected coordinates and subsequent clustering operations remain memory-resident. The `--step` option reduces the number of analyzed rows through subsampling. Its effect on cluster populations, assignments, and observed label changes must nevertheless be evaluated for the dataset under investigation.

When `--auto` is enabled, *clusterMD* sets the HDBSCAN parameters using a square-root heuristic based on the number N of analyzed projection rows:

$$\begin{aligned}\text{min_cluster_size} &= \max\left(100, \left\lfloor 500 \sqrt{\frac{N}{1,000,000}} \right\rfloor\right), \\ \text{min_samples} &= \max\left(10, \left\lfloor \frac{\text{min_cluster_size}}{10} \right\rfloor\right).\end{aligned}$$

The lower bounds prevent the automatic parameters from decreasing below the specified minimum values for small datasets. This rule provides a practical default, but it does not establish optimal parameters or guarantee comparable partitions across dataset sizes. The square-root dependence describes parameter scaling rather than the runtime or memory

complexity of the complete pipeline. Performance scaling and parameter sensitivity therefore require separate evaluation.

The execution report records elapsed times for the main processing stages and the Python process resident set size (RSS) at report generation. This RSS value is a memory snapshot rather than a peak-memory measurement and does not include the combined memory consumption of external *carma* processes. Similarly, the number of logical CPU cores detected describes the system hardware rather than the number of cores actively used throughout execution.

3.3. User Interface, Diagnostic Mode, and Reproducibility

The command-line interface exposes the principal analysis settings, including HDBSCAN parameters, cluster-selection method, retained dimensionality, projection-row stride, temperature, and structural-processing options. The `--dim` option selects the leading projected components used for clustering, with 0 retaining all available components. When an appropriate eigenvalue file is available, the report estimates the fraction of projection-space variance represented by the retained components. This is a variance summary rather than a measure of thermodynamic completeness, and its interpretation requires the eigenvalues to correspond to the analyzed projection.

The `--cluster-only` option performs ingestion and clustering, exports the assignments, and exits before downstream analysis. A separate diagnostic mode, accessed through `--transitions` or its alias `--changes`, inspects an existing assignment file. It displays the most frequent observed cluster-pair changes or reports individual chronological events for a specified pair without repeating clustering or structural extraction.

For full pipeline runs, execution messages are recorded in a log file, and the analysis report documents the command invocation, selected inputs, parameter values, population accounting, processing outcomes, software-version information, and resource measurements. These records support reproducibility but should be retained together with the input data, dependency environment, and the specific source-code revision used for the analysis.

Option	Purpose	Default
<code>--size INT</code>	HDBSCAN minimum cluster size	500
<code>--samples INT</code>	HDBSCAN minimum samples	50
<code>--auto</code>	Set parameters using the analyzed-row-count heuristic	Disabled
<code>--method {eom,leaf}</code>	Cluster-selection method	eom
<code>--dim INT</code>	Number of leading projected components; 0 retains all	0
<code>--step INT</code>	Additional row stride during projection ingestion	1
<code>--temp FLOAT</code>	Temperature for population-derived free-energy estimates, in K	298.0
<code>--rmsd FLOAT</code>	Downstream structural-filtering threshold, in Å; 0 disables filtering	0.0
<code>--segid STRING</code>	Restrict structural processing to a segment identifier	Unspecified

Option	Purpose	Default
--cluster-only	Export clustering assignments and exit before downstream analysis	Disabled
--top INT	Number of highest-count cluster pairs displayed; 0 disables displayed edges	15
--transitions [A B] / --changes [A B]	Inspect saved assignments; no IDs gives an overview, two IDs select a pair	Inactive
--events INT	Maximum individual events printed during diagnostic inspection	3

3.4. Environment Deployment and Open Source

The *clusterMD* source code and installation instructions are publicly available through the GitHub repository. The repository provides `requirements.txt` for dependency installation and `requirements-lock.txt` for reproducing the documented Python package versions. A virtual environment is created and activated separately. The `setup_env.sh` script checks the expected repository structure, makes the scripts executable, installs dependencies through Python's package manager, and prints the command needed to add the scripts directory to `PATH`.

The documented execution environment uses Linux or the Windows Subsystem for Linux (WSL), with *carma* installed separately and available in `PATH` when structural processing is required. The external *carma* executable is not bundled with *clusterMD*. Reproducing an analysis requires preserving the source revision, input files, analysis parameters, Python dependency versions, and relevant external-software versions. Deployment on a research cluster additionally requires verification of the local software environment and execution configuration.

CHAPTER 4: Application Examples and Computational Evaluation

This chapter presents applications of *clusterMD* to two peptide molecular-dynamics datasets. The examples illustrate multidimensional clustering, population summaries, structural characterization, and inspection of observed label changes. Computational measurements and a temporal-subsampling comparison are also presented to evaluate practical execution and sensitivity under the tested conditions.

4.1 Datasets and Analysis Settings

Two peptide datasets were selected to illustrate the analytical workflow and its application to different data volumes.

1. **CLN025:** The first dataset contains 1,000,000 trajectory frames of CLN025, a 10-residue chignolin variant with sequence YYDPETGTWY and reference structure PDB 2RVD. This dataset was used to examine cluster populations, structural ensembles, optional RMSD filtering, and observed changes between assigned cluster labels.
2. **CDP-1:** The second dataset contains approximately 7.5 million trajectory frames of cysteine-deleted protegrin-1 (CDP-1), a 14-residue antimicrobial peptide with reference structure PDB 2MQ2. This larger dataset was used to examine practical execution at increased data volume and structural characterization restricted to the selected peptide segment.

4.2 Multidimensional Clustering and Population Summaries of CLN025

Precomputed dPCA coordinates generated through the grcarma workflow were supplied to *clusterMD* for analysis of the CLN025 dataset. HDBSCAN clustering was performed in the retained multidimensional projection space, while the PC1–PC2 figures provide two-dimensional visualizations of the sampled coordinates and cluster assignments.

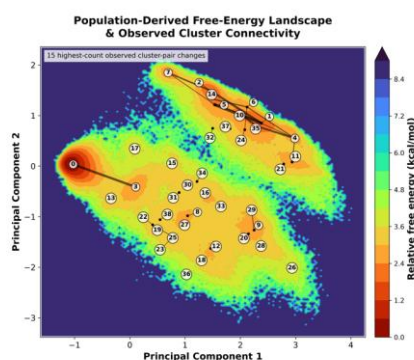


Figure 4.1 - PC1–PC2 population-derived free-energy landscape for CLN025. Colors indicate relative free energy (kcal/mol); connections summarize observed, undirected label changes. This projected landscape does not establish kinetic barriers.

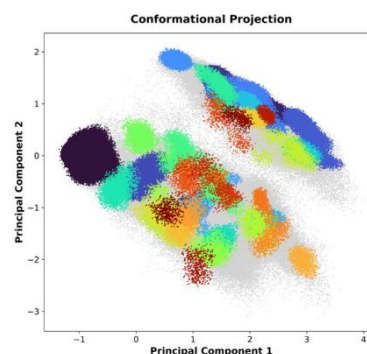


Figure 4.2 - PC1–PC2 projection of CLN025, colored by HDBSCAN assignments in the retained multidimensional space. Unassigned frames retain label -1. Apparent overlap can conceal separation in additional components.

The analyzed dataset yielded 39 assigned clusters. Cluster 0, the most populated cluster, contained 613,098 frames, corresponding to approximately 61.3% of the 1,000,000 analyzed frames. It defines the zero reference for the relative cluster-population free-energy estimates because it has the largest assigned population. The remaining clusters represent additional

density-defined groups whose structural characteristics are examined through the extracted ensembles.

The structural ensemble displayed for Cluster 0 is consistent with a folded β -hairpin conformation. Assignment to the native state should additionally be supported by comparison with the reference structure or an explicitly defined native-structure criterion.

4.3 Structural Characterization and Optional RMSD Filtering

Frames assigned to the same density-defined cluster can retain structural variability, because similarity in the retained projection coordinates does not require identical three-dimensional conformations. Extracted structural ensembles therefore provide an additional means of examining the configurations grouped by HDBSCAN.

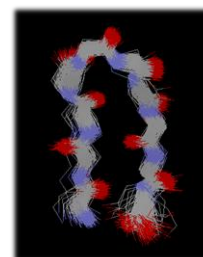


Figure 3.3 - Structural superposition of CLN025 Cluster 0 (613,098 frames; 61.3%). The displayed ensemble is consistent with a folded β -hairpin; native-state assignment requires an explicit reference comparison.

The CLN025 examples compare complete cluster ensembles with subsets retained by optional RMSD filtering. Filtering is applied relative to the initial fitted ensemble average, after which the retained frames are extracted again and the final fitted average and superposition recalculated. This operation changes the structural ensemble displayed downstream without changing the original HDBSCAN assignments or populations.

The reported final maximum RMSD values were 2.09 Å for Cluster 0, 2.28 Å for Cluster 3, and 2.63 Å for Cluster 19. These values describe deviations from the corresponding final ensemble averages rather than maximum pairwise structural distances. Because the average is recalculated after filtering, the final RMSD maximum may exceed the original selection threshold. The superpositions illustrate downstream structural selection without independently establishing kinetic metastability.

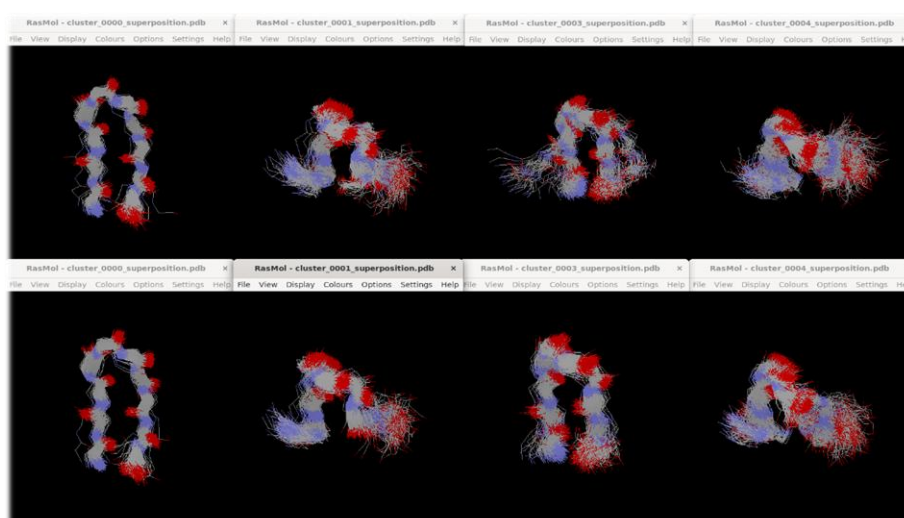


Figure 4.4 - CLN025 structural ensembles before (top) and after (bottom) optional RMSD filtering. Filtering selects subsets relative to the initial fitted averages; clustering assignments and populations are unchanged. Reduced dispersion does not establish metastability.

4.4 Observed Label Changes and Chronological Inspection

The chronological sequence of cluster assignments was examined to summarize changes between assigned labels. Counts were aggregated as undirected cluster pairs after omitting unassigned configurations from the counting sequence. Intervening unassigned frames remain available for diagnostic inspection.

The CLN025 analysis reported 200 unique cluster pairs with observed label changes. The pair 0–3 had 230 changes, while the pair 3–19 had 29. These counts combine both directions and therefore do not establish a directed unfolding sequence from Cluster 0 through Cluster 3 to Cluster 19.

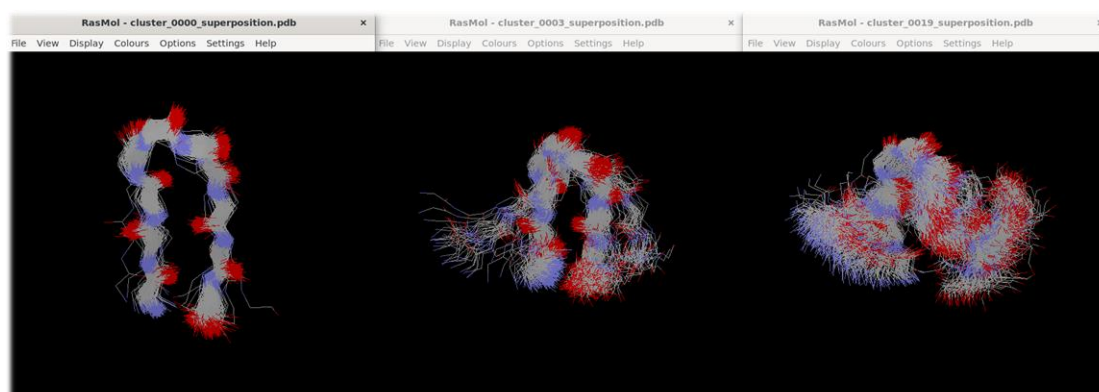


Figure 4.5 - Structural superpositions of CLN025 Clusters 0, 3, and 19, illustrating differences in conformational organization. Observed changes connect pairs 0–3 and 3–19, but do not establish a directed unfolding pathway.

The most frequently reported pair, 1–5, had 312 observed changes. Diagnostic inspection illustrates the preceding and following assigned configurations and any intervening unassigned frames. In the displayed historical example, frame 151134 belongs to Cluster 5 and frame 151147 to Cluster 1, with 12 intervening unassigned frames. The excerpt below retains the original log terminology; “transition,” “jump,” and “fog” denote observed label changes and unassigned frames, not physical transition states or validated pathways.

=====

```
[*] DIAGNOSTIC MODE: VALIDATING TRANSITIONS FOR EDGE (1, 5)
```

=====

```
[+] Total Unique Transition Pathways Discovered: 200
```

```
[+] Top 50 Most Frequent Transitions:
```

```
Edge (A <-> B) | Jump Count
```

```
Edge 1 <-> 5 | 312 jumps
```

```
Edge 0 <-> 3 | 230 jumps
```

```
Edge 1 <-> 2 | 215 jumps
```

```
[... Log Truncated ...]
```

```
--- Transition Event 7 ---
```

LEAVING : Frame 151134 | Cluster 5 | PCs: 1.55219 1.26966 -0.71540 1.09101 0.73149
 ... [12 frames of transition fog (-1)] ...
 FOG : Frame 151135 | Cluster -1 | PCs: 1.55165 1.19205 -0.50369 1.03466 0.38545
 FOG : Frame 151136 | Cluster -1 | PCs: 1.81133 1.00815 -1.04612 0.76794 0.01651
 ARRIVING : Frame 151147 | Cluster 1 | PCs: 1.88833 1.06398 -1.10839 0.31632 0.05952

This example illustrates the chronological information available for an observed cluster-pair change. The resulting connectivity summaries describe the ordering of assigned configurations in the analyzed sequence. They are not kinetic rate estimates, transition probabilities, or evidence that every connecting edge corresponds to a distinct mechanistic pathway.

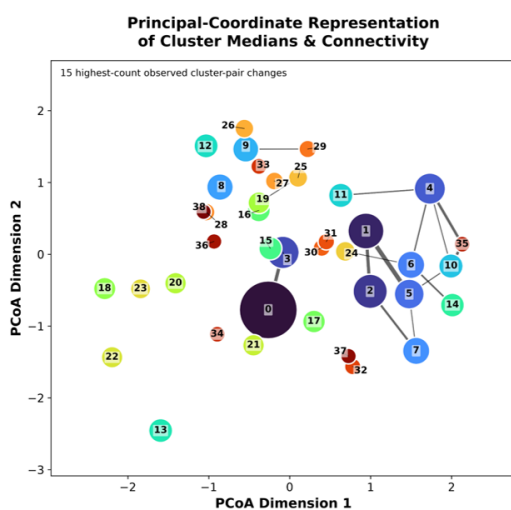


Figure 4.6 - PCoA representation of cluster medians in the retained projection space. Node size indicates population and color indicates relative population-derived free energy. Edges show undirected observed label changes, not kinetic rates.

4.5 Execution on the Larger CDP-1 Dataset

clusterMD was also applied to the larger CDP-1 dataset to examine practical execution at increased data volume. Structural processing was restricted to Segment A using the segment-selection option. This restriction affects extraction and structural fitting, while clustering remains determined by the supplied projection coordinates.

The reported historical analysis processed 2,510,667 sampled projection rows, identified 156 assigned clusters, and completed in 7 minutes and 59 seconds. These measurements describe that execution and should not be combined with later runs using different settings. The analysed-row count is distinct from the approximately 7.5 million frames in the original trajectory; --step applies to the supplied projection rows. Figure 4.7 compares complete structural ensembles with RMSD-filtered subsets.

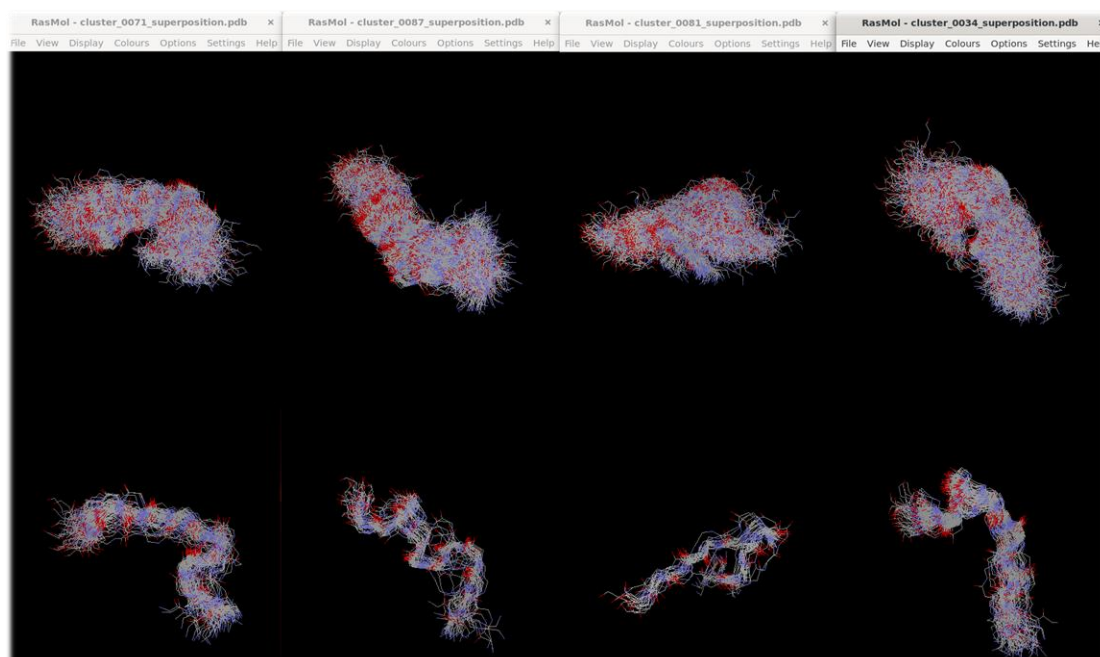


Figure 4.7 - CDP-1 structural superpositions before (top) and after (bottom) optional RMSD filtering. The analysis used 2,510,667 projection rows from a trajectory of approximately 7.5 million frames. Filtering changes structural retention, not HDBSCAN populations.

4.6 Sensitivity to Temporal Subsampling

Temporal subsampling changes the number and density of configurations supplied to HDBSCAN and can therefore affect clustering results. To examine this sensitivity, the CLN025 projection dataset was analyzed using row strides of 1 and 10.

The stride-1 run analyzed 1,000,000 rows and identified 39 clusters, whereas the stride-10 run analyzed 100,000 rows and identified 38 clusters. The most populated cluster represented approximately 61.3% of the analyzed configurations in both runs, and the next two population-ranked clusters had similar proportions. These observations support similarity in the leading population summaries under the two tested settings, but they do not establish identical memberships or general independence from sampling stride.

4.7 Qualitative Comparison of Structural Ensembles

The displayed structural superpositions provide a qualitative comparison between the selected comparator analysis and the HDBSCAN-based *clusterMD* analysis. They illustrate differences in how the trajectory configurations are partitioned and represented structurally.

Interpretation of this comparison requires the comparator algorithm and its parameters to be specified, together with the input projection space, structural fitting procedure, and filtering status. A comparison of complete ensembles should use unfiltered [ER4.1]*clusterMD* outputs so that the effect of clustering is distinguished from subsequent RMSD-based selection.

The figures support visual inspection of structural heterogeneity within the displayed groups. Establishing comparative performance would additionally require controlled settings and quantitative criteria, such as within-cluster structural dispersion and the fraction of analyzed configurations assigned to clusters.

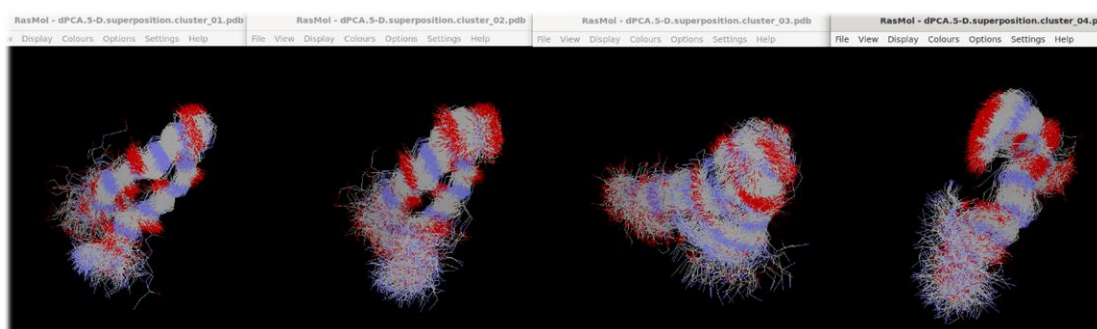


Figure 4.8 - Structural superpositions from the comparator analysis. The displayed ensembles provide a qualitative reference; differences cannot establish comparative performance without matched inputs, fitting, filtering, and documented clustering parameters.

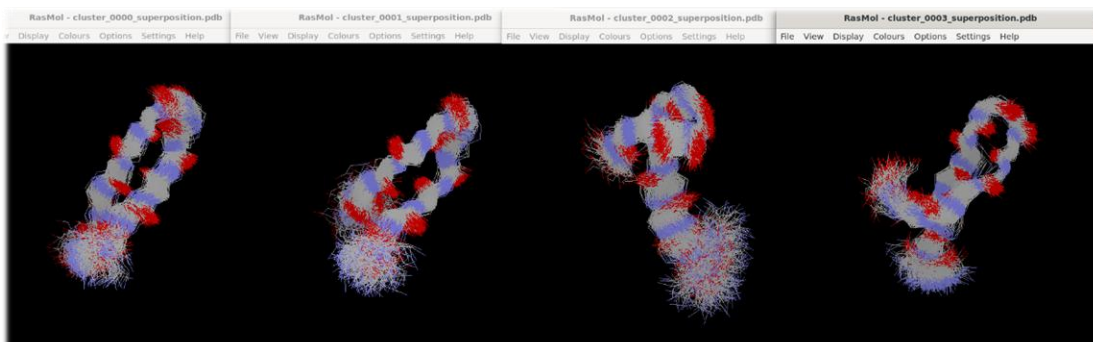


Figure 4.9 - Structural superpositions from the HDBSCAN-based *clusterMD* analysis. The displayed groups illustrate conformational heterogeneity. Comparison with Figure 4.8 requires matched analysis settings and quantitative structural criteria.

CHAPTER 5: Discussion & Conclusions

clusterMD was developed to automate the analysis of multidimensional PCA/dPCA projections and connect density-based cluster assignments to observed molecular structures. The workflow combines HDBSCAN clustering, population summaries, representative-frame selection, structural characterization, and inspection of observed label changes. Clustering in the retained multidimensional space reduces reliance on two-dimensional visualizations, while support for dPCA allows the analysis of internal-coordinate representations. The application examples demonstrate the practical integration of these operations for peptide trajectories, with scientific interpretation remaining dependent on sampling quality, analysis settings, and independent structural or kinetic evidence.

5.1 Methodological Limitations and Heuristics

The interpretation of *clusterMD* outputs depends on both the underlying trajectory and the selected analysis settings. The coordinate representation, retained dimensionality, sampling stride, and HDBSCAN parameters can affect cluster boundaries, populations, and representative structures. The automatic parameter heuristic provides a practical starting point rather than a universally optimal choice. Parameter sensitivity and the adequacy of trajectory sampling should therefore be assessed when conclusions depend on the resulting partition.

Population-derived free-energy estimates assume that the analyzed configurations adequately represent the relevant equilibrium ensemble. The default temperature is 298.0 K and should be replaced through `--temp` when a different temperature is appropriate for the analyzed simulation. The PC1–PC2 free-energy visualization also applies a 9.0 kcal/mol display cap. This cap limits the displayed range of the projected population-derived profile; it does not represent a measured barrier threshold or an exclusion of clusters from the analysis.

Observed label changes are descriptive summaries rather than kinetic measurements. Counts combine both directions for each cluster pair and can connect assigned configurations across intervening unassigned frames. The `--top` option limits the pairs displayed in selected tables and plots without removing lower-count pairs from the complete observed-change report. Reliable kinetic interpretation would require additional treatment of physical time, state definitions, and trajectory continuity. In particular, independent trajectories should not be treated as one continuous sequence across concatenation boundaries.

Optional RMSD filtering selects a downstream structural subset relative to the initial fitted ensemble average. It leaves the original HDBSCAN assignments and populations unchanged. A restrictive threshold can substantially reduce structural retention, and when no frames pass, the implementation reports a warning and retains the complete ensemble. Filtered superpositions should therefore be interpreted together with the threshold and retained-frame fraction, particularly for flexible or intrinsically disordered systems.

Finally, the application examples establish execution under the tested conditions rather than a general scaling law or superiority over alternative methods. The reported Python process RSS is a snapshot at report generation, not peak memory or total memory across external

subprocesses. Broader performance conclusions require controlled measurements across dataset sizes and consistently configured comparisons.

5.2 Potential Applications in Computational Structural Biology

The combination of multidimensional clustering and structural characterization could support several applications in computational structural biology. The present thesis illustrates the workflow using peptide datasets; the broader applications described below would require system-specific evaluation and appropriate supporting analyses.

- **Protein folding analysis:** Density-defined clusters and their extracted structures can help organize sampled conformations and identify candidate folded, partially folded, or unfolded structural groups. Inspection of their chronological occurrence can guide further investigation, although folding mechanisms and kinetic rates require dedicated analyses beyond observed label-change counts.
- **Comparative variant analysis:** Applying a consistent analysis strategy to wild-type and variant trajectories could reveal differences in sampled conformational distributions, cluster populations, and representative structures. Meaningful comparisons require compatible coordinate representations, comparable sampling, and a method for matching clusters between analyses. Cluster identifiers assigned by population rank are not persistent state identities across separate runs.
- **Characterization of intrinsically disordered proteins:** Density-based clustering may help identify recurring conformational groups within heterogeneous ensembles without requiring every configuration to be assigned to a cluster. Structural variability remains an important property of these systems, and optional RMSD filtering should not be used to imply that a narrow structural subset represents the complete ensemble.

5.3 Future Work and Pipeline Expansion

Further development should strengthen the evidence supporting reproducibility and interpretation of the workflow. Priorities include versioned validation fixtures, controlled parameter-sensitivity studies, quantitative structural comparisons, and performance measurements that capture peak memory and external subprocess consumption. Explicit input-file selection and support for separate trajectory boundaries would also reduce ambiguity in structural mapping and chronological analysis.

A further implementation improvement is parallel processing of independent cluster ensembles through separate *carma* working directories. This could reduce structural-processing time, but the benefit should be evaluated against increased memory use, storage activity, and competition for CPU resources. A modular package organization could support this development while preserving verified clustering and structural-processing behavior.

Alternative coordinate representations could be evaluated in future work, including methods designed to emphasize slow conformational dynamics and nonlinear learned representations. Such extensions should be assessed against the existing PCA/dPCA workflow using defined structural and dynamical criteria, rather than assuming that a more complex representation necessarily improves interpretation.

Integration with Markov state modeling would require more than converting the current cluster-pair counts into probabilities. It would require suitable state definitions, transition counting at a specified physical lag time, preservation of independent trajectory boundaries, explicit treatment of unassigned configurations, and model validation.

The principal contribution of this thesis is an automated workflow linking multidimensional density-based clustering to population summaries and observed molecular structures. The peptide examples illustrate how *clusterMD* can organize large projection datasets and support reproducible structural inspection. Its outputs provide a basis for further investigation of conformational organization, while thermodynamic and kinetic conclusions require evidence beyond the clustering assignments themselves.

Bibliography

Altis, A., Otten, M., Nguyen, P. H., Hegger, R., & Stock, G. (2008). Construction of the free energy landscape of biomolecules via dihedral angle principal component analysis. *The Journal of Chemical Physics*, 128(24), 245102.

Campello, R. J., Moulavi, D., & Sander, J. (2013). Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining: 17th Pacific-Asia Conference, PAKDD 2013* (pp. 160-172). Springer.

Dinner, A. R., Šali, A., Smith, L. J., Dobson, C. M., & Karplus, M. (2000). Understanding protein folding via free-energy surfaces from theory and experiment. *Trends in Biochemical Sciences*, 25(7), 331-339.

Glykos, N. M. (2006). Carma: A molecular dynamics analysis program. *Journal of Computational Chemistry*, 27(14), 1765-1768.

Koukos, P. I., & Glykos, N. M. (2013). Grcarma: A fully automated task-oriented interface for the analysis of molecular dynamics trajectories. *Journal of Computational Chemistry*, 34(26), 2310-2312.

Lindorff-Larsen, K., Piana, S., Dror, R. O., & Shaw, D. E. (2011). How fast-folding proteins fold. *Science*, 334(6055), 517-520.

McInnes, L., Healy, J., & Astels, S. (2017). hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), 205.

Melvin, R. L., Godwin, R. C., Xiao, J., Thompson, W. G., Berenhaut, K. S., & Salsbury, F. R. (2016). Uncovering large-scale conformational change in molecular dynamics without prior knowledge. *Journal of Chemical Theory and Computation*, 12(12), 6130-6146.

Appendix: *clusterMD* Source Code

The source listing below matches the thesis implementation at GitHub commit [1d8761258bc83f2c5311713d47912c5ad4897fc4](https://github.com/tasosmelas/md-hdbscan-clustering/tree/1d8761258bc83f2c5311713d47912c5ad4897fc4). The archived revision is available at <https://github.com/tasosmelas/md-hdbscan-clustering/tree/1d8761258bc83f2c5311713d47912c5ad4897fc4>. Subsequent repository changes are outside the scope of this listing.

```
#!/usr/bin/env python3
"""
clusterMD: multidimensional HDBSCAN clustering and structural analysis
of PCA/dPCA coordinates derived from molecular-dynamics trajectories.

Executes the complete clustering, observed-label-change, visualization, and structural-extraction sequence.
"""

import os
import sys
import glob
import time
import datetime
import argparse
import subprocess
import warnings
import psutil
import logging
import math
import shutil
import tempfile
import numpy as np
import scipy
import re
import shlex

# =====
# GLOBAL CONSTANTS & PHYSICAL PARAMETERS
# =====
R_KCAL = 0.0019872041          # Ideal gas constant in kcal/(mol*K)
MAX_FREE_ENERGY_KCAL = 9.0    # Cap for free-energy surface plots (kcal/mol)
FALLBACK_ATOM_FLAGS = ["-atomid", "CA", "-atomid", "C", "-atomid", "N", "-atomid", "O"]
```

```

ANSI_ESCAPE = re.compile(r'\x1b\[([0-9;]*)m')

class Colors:
    LIGHT_BLUE = '\033[96m'    # Processing / Info steps
    BLUE = '\033[94m'          # Success / Milestones
    YELLOW = '\033[93m'        # Warnings
    RED = '\033[91m'           # Errors
    BOLD = '\033[1m'
    RESET = '\033[0m'

class PlainFileFormatter(logging.Formatter):
    """Strips ANSI color codes for clean file logging."""
    def format(self, record):
        message = super().format(record)
        return ANSI_ESCAPE.sub('', message)

def get_clusterMD_file_handler():
    """Safely retrieves the designated file handler by name."""
    for handler in logging.getLogger().handlers:
        if getattr(handler, "name", None) == "clusterMD_file":
            return handler
    return None

class PlotStyle:
    """Centralized formatting standard for all scientific figures."""
    FIG_SIZE = (10, 8)
    DPI = 600
    # Absolute margins guarantee the internal axes rect is pixel-identical across all 2D plots
    MARGINS = {'left': 0.12, 'right': 0.80, 'bottom': 0.12, 'top': 0.88}
    CBAR_AXES = [0.82, 0.12, 0.03, 0.76]
    LEGEND_BBOX = (1.02, 0.5)

    CMAP_SCATTER = 'turbo'
    CMAP_FES = 'turbo_r'

    NOISE_COLOR = 'lightgrey'

```

```

NOISE_SIZE = 0.5
NOISE_ALPHA = 0.5
CLUSTER_SIZE = 1.0
CLUSTER_ALPHA = 0.8

TITLE_FMT = {'fontsize': 16, 'fontweight': 'bold', 'pad': 15}
LABEL_FMT = {'fontsize': 14, 'fontweight': 'bold'}
TICK_SIZE = 12

@staticmethod
def apply_standard_layout(fig, ax, title, xlabel, ylabel, xlim=None, ylim=None):
    fig.subplots_adjust(**PlotStyle.MARGINS)
    ax.set_title(title, **PlotStyle.TITLE_FMT)
    ax.set_xlabel(xlabel, **PlotStyle.LABEL_FMT)
    ax.set_ylabel(ylabel, **PlotStyle.LABEL_FMT)
    if xlim: ax.set_xlim(xlim)
    if ylim: ax.set_ylim(ylim)
    ax.tick_params(labelsize=PlotStyle.TICK_SIZE)

@staticmethod
def get_cluster_colors(unique_clusters, cmap_name):
    import matplotlib.pyplot as plt
    cmap = plt.get_cmap(cmap_name)
    n_clusters = len(unique_clusters)
    return {cid: cmap(i / max(1, n_clusters - 1)) for i, cid in enumerate(unique_clusters)}

@staticmethod
def repel_labels(node_positions, xlim, ylim):
    """Deterministic spring-repel algorithm to prevent label overlap."""
    labels_pos = {k: np.array(v, dtype=float) for k, v in node_positions.items()}
    x_range = xlim[1] - xlim[0]
    y_range = ylim[1] - ylim[0]
    min_dist_x = x_range * 0.05
    min_dist_y = y_range * 0.05

    for _ in range(150):

```

```

moved = False
forces = {k: np.zeros(2) for k in labels_pos}
for k1, p1 in labels_pos.items():
    forces[k1] += (node_positions[k1] - p1) * 0.1 # Spring to origin
    for k2, p2 in labels_pos.items():
        if k1 == k2: continue
        dx = p1[0] - p2[0]
        dy = p1[1] - p2[1]
        dist_x = abs(dx)
        dist_y = abs(dy)
        if dist_x < min_dist_x and dist_y < min_dist_y:
            tie_sign = -1.0 if k1 < k2 else 1.0
            repel_x = (
                (min_dist_x - dist_x) * np.sign(dx)
                if dx != 0
                else (min_dist_x - dist_x) * tie_sign
            )
            repel_y = (
                (min_dist_y - dist_y) * np.sign(dy)
                if dy != 0
                else (min_dist_y - dist_y) * tie_sign
            )
            forces[k1] += np.array([repel_x, repel_y]) * 0.3
            moved = True
for k in labels_pos:
    labels_pos[k] += forces[k]
if not moved:
    break
return labels_pos

```

```

# =====
# 1. FORMATTING & ENV UTILITIES
# =====
def format_duration(seconds):
    """Converts raw seconds into a human-readable time format."""

```

```

m, s = divmod(int(seconds), 60)
h, m = divmod(m, 60)
if h > 0: return f"{h}h {m}m {s}s"
elif m > 0: return f"{m}m {s}s"
else: return f"{seconds:.2f}s"

def format_record(raw_line, frame_val, cid_val):
    """Parses a raw data line and aligns it into a clean, readable column format."""
    coords_str = " ".join([f"{c:9.5f}" for c in raw_line])
    return f"Frame {frame_val:7d} | Cluster {cid_val:4d} | Coordinates: {coords_str}"

def get_software_env():
    """Detects software and library versions for the provenance report."""
    try:
        import importlib.metadata
        fh_ver = importlib.metadata.version('fast_hdbscan')
    except Exception:
        try:
            import fast_hdbscan
            fh_ver = fast_hdbscan.__version__ if hasattr(fast_hdbscan, '__version__') else "not detected"
        except ImportError:
            fh_ver = "not found"

carma_path = shutil.which("carma")
carma_ver = "not detected"
if carma_path:
    try:
        res = subprocess.run(["carma"], capture_output=True, text=True)
        for line in (res.stderr + "\n" + res.stdout).splitlines():
            if "version" in line.lower() or " v." in line.lower() or "carma v" in line.lower():
                match = re.search(r'v\.\.?s*(\d+\.\.\d+)', line.lower())
                if match:
                    carma_ver = match.group(1)
            else:
                parts = line.strip().split()
                carma_ver = parts[1] if len(parts) > 1 else line.strip()

```

```

        break
    except Exception:
        pass

    return {
        "clusterMD_ver": "1.1.0",
        "python_ver": sys.version.split()[0],
        "numpy_ver": np.__version__,
        "scipy_ver": scipy.__version__,
        "fh_ver": fh_ver,
        "carma_path": carma_path if carma_path else "NOT FOUND",
        "carma_ver": carma_ver
    }

# =====
# 2. LOADING (Memory-Efficient I/O)
# =====
def ingest_trajectory_data(file_path, step=1):
    """Loads and validates a strided projection dataset using preallocated memory."""
    if os.path.getsize(file_path) == 0:
        logging.error(f"{Colors.RED}[-]{Colors.RESET} Critical error: Coordinate file is empty.")
        sys.exit(1)

    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Pre-allocating memory and loading data (Stride: {step})...")

    try:
        out = subprocess.check_output(['wc', '-l', file_path])
        total_lines = int(out.split()[0])
    except (subprocess.CalledProcessError, FileNotFoundError):
        with open(file_path, 'r') as f:
            total_lines = sum(1 for _ in f)

    parse_stats = {
        'total_lines': total_lines,
        'selected_by_stride': 0,

```

```

    'valid_sampled': 0,
    'malformed_sampled': 0,
    'first_sampled_frame': None,
    'last_sampled_frame': None,
    'sampled_monotonic': True,
    'sampled_duplicates': False
}

from collections import Counter
col_counts = []
with open(file_path, 'r') as f:
    for i, line in enumerate(f):
        if i >= 500:
            break
        parts = line.split()
        if parts:
            col_counts.append(len(parts))

if not col_counts:
    logging.error(f"{Colors.RED}[-]{Colors.RESET} Critical error: Coordinate file contains no parseable data.")
    sys.exit(1)

num_cols = Counter(col_counts).most_common(1)[0][0]

subset_size = math.ceil(total_lines / step)
data = np.zeros((subset_size, num_cols))

last_frame_val = None
seen_frames = set()

with open(file_path, 'r') as f:
    idx = 0
    for i, line in enumerate(f):
        if i % step == 0:
            parse_stats['selected_by_stride'] += 1
            parts = line.split()

```

```

if parts:
    if len(parts) == num_cols:
        try:
            frame_val = float(parts[0])
            if not frame_val.is_integer() or frame_val <= 0:
                logging.error(f"{Colors.RED}[-]{Colors.RESET} Critical error: Frame IDs must be positive
integers. Found '{parts[0]}' at line {i+1}.")
                sys.exit(1)

            frame_int = int(frame_val)
            if parse_stats['first_sampled_frame'] is None:
                parse_stats['first_sampled_frame'] = frame_int
            parse_stats['last_sampled_frame'] = frame_int

            if last_frame_val is not None and frame_int <= last_frame_val:
                parse_stats['sampled_monotonic'] = False
            if frame_int in seen_frames:
                parse_stats['sampled_duplicates'] = True

            seen_frames.add(frame_int)
            last_frame_val = frame_int

            row_vals = [float(x) for x in parts]
            if not all(math.isfinite(x) for x in row_vals):
                logging.error(f"{Colors.RED}[-]{Colors.RESET} Critical error: Non-finite value (NaN/Inf)
detected at line {i+1}. Analysis aborted.")
                sys.exit(1)

            data[idx] = row_vals
            idx += 1
            parse_stats['valid_sampled'] += 1
        except ValueError:
            parse_stats['malformed_sampled'] += 1
            logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Format anomaly at line {i+1}: non-
numeric data. Row skipped.")
    else:

```

```

        parse_stats['malformed_sampled'] += 1
        logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Format anomaly at line {i+1}: expected
{num_cols} cols, found {len(parts)}. Frame skipped.")

    return data[:idx], parse_stats

# =====
# 3. CLUSTERING (HDBSCAN Engine & Sorting)
# =====
def execute_clustering_algorithm(data_nd, size, samples, method):
    """Executes the core HDBSCAN clustering algorithm."""
    import fast_hdbscan
    t0 = time.time()
    clusterer = fast_hdbscan.HDBSCAN(min_cluster_size=size, min_samples=samples, cluster_selection_method=method)
    labels = clusterer.fit_predict(data_nd)
    runtime = time.time() - t0
    return labels, runtime

def sort_cluster_populations(labels):
    """Sorts cluster assignments by descending population size."""
    unique_clusters = np.unique(labels[labels != -1])
    cluster_counts = {cid: np.sum(labels == cid) for cid in unique_clusters}
    sorted_old_cids = sorted(cluster_counts, key=cluster_counts.get, reverse=True)

    id_mapping = {-1: -1}
    for new_id, old_id in enumerate(sorted_old_cids): id_mapping[old_id] = new_id

    reordered_labels = np.copy(labels)
    for old_id, new_id in id_mapping.items():
        if old_id != -1: reordered_labels[labels == old_id] = new_id

    return reordered_labels, np.unique(reordered_labels[reordered_labels != -1])

def export_cluster_coordinates(frames, labels, data_nd, out_file, n_dimensions):
    """Exports the assigned coordinate matrices to a .dat file."""

```

```

out_data = np.column_stack((frames, labels, data_nd))
fmt_string = f"%10d %4d " + ' '.join(['%11.7f'] * n_dimensions)
np.savetxt(out_file, out_data, fmt=fmt_string)

# =====
# 4. POPULATIONS, FREE-ENERGY ESTIMATES & OBSERVED LABEL CHANGES
# =====
def calculate_populations_and_energies(labels, temp):
    """Computes assigned-cluster populations and population-derived relative free-energy estimates."""
    valid_mask = labels != -1
    valid_labels = labels[valid_mask]
    states, counts = np.unique(valid_labels, return_counts=True)
    population_map = {state: count for state, count in zip(states, counts)}

    k_B_T = R_KCAL * temp
    max_count = np.max(counts) if len(counts) > 0 else 1
    free_energies = {state: -k_B_T * np.log(count / max_count) for state, count in zip(states, counts)}

    return states, counts, population_map, free_energies, k_B_T

def map_observed_label_changes(labels):
    """Counts undirected changes between consecutive assigned labels after omitting unassigned/noise configurations."""
    valid_labels = labels[labels != -1]
    label_changes = {}
    for i in range(len(valid_labels) - 1):
        sA, sB = valid_labels[i], valid_labels[i+1]
        if sA != sB:
            edge = tuple(sorted([sA, sB]))
            label_changes[edge] = label_changes.get(edge, 0) + 1
    return label_changes

def print_top_observed_changes(label_changes, top_n):
    """Outputs the most frequent observed changes between assigned cluster labels."""
    sorted_changes = sorted(label_changes.items(), key=lambda item: item[1], reverse=True)

```

```

logging.info("\n    OBSERVED LABEL CHANGES")
logging.info("    " + "-"*49)
logging.info(f"    {'Cluster pair':<32}{'Observed changes':>17}")
logging.info("    " + "-"*49)

for edge, count in sorted_changes[:top_n]:
    pair_str = f"{edge[0]} <--> {edge[1]}"
    logging.info(f"    {pair_str:<32}{count:>17d}")

logging.info("    " + "-"*49)
logging.info(f"    {'Unique cluster pairs':<32}{len(label_changes):>17d}")

return sorted_changes

def hunt_chronological_label_changes(target_edge, labels, frames, data_nd, max_events):
    """Inspects the chronology of observed changes between assigned cluster labels."""
    logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Inspecting chronological change events for cluster pair {target_edge}...")
    events_found = 0
    last_valid_state, last_valid_idx = None, None
    intervening_noise_indices = []

    for idx, label in enumerate(labels):
        if label != -1:
            if last_valid_state is not None and last_valid_state != label:
                current_edge = tuple(sorted([last_valid_state, label]))
                if current_edge == target_edge:
                    events_found += 1
                    if events_found <= max_events:
                        logging.info(f"\n{Colors.LIGHT_BLUE}--- Observed Change Event {events_found} ---{Colors.RESET}")
                        logging.info(f"PREVIOUS ASSIGNED : {format_record(data_nd[last_valid_idx], frames[last_valid_idx], last_valid_state)}")
                    if intervening_noise_indices:
                        logging.info(f"    ... [ {len(intervening_noise_indices)} intervening unassigned/noise frames (-1) ] ...")

```

```

        for f_idx in intervening_noise_indices[:2]: logging.info(f"    UNASSIGNED:
{format_record(data_nd[f_idx], frames[f_idx], -1)}")
        if len(intervening_noise_indices) > 2: logging.info("    ...    (additional unassigned/noise
frames hidden to save space)")
        logging.info(f"NEXT ASSIGNED    : {format_record(data_nd[idx], frames[idx], label)}")
        last_valid_state, last_valid_idx = label, idx
        intervening_noise_indices = []
    else:
        if last_valid_state is not None: intervening_noise_indices.append(idx)

    if events_found == 0:
        logging.error(f"{Colors.YELLOW}[WARN]{Colors.RESET} No observed label changes found between Cluster
{target_edge[0]} and Cluster {target_edge[1]}.")
    elif events_found > max_events:
        logging.info(f"\n{Colors.BLUE}[+]{Colors.RESET} ... and {events_found - max_events} additional events hidden.")
    logging.info(f"="*85 + "\n")

# =====
# 5. VISUALIZATION
# =====
def render_conformational_landscape(x, y, lx, ly, p1, p2, unique_clusters, labels, zero_var, out_prefix):
    import matplotlib.pyplot as plt
    fig, ax = plt.subplots(figsize=PlotStyle.FIG_SIZE, dpi=PlotStyle.DPI)

    if np.any(labels == -1):
        ax.scatter(x[labels == -1], y[labels == -1], color=PlotStyle.NOISE_COLOR, s=PlotStyle.NOISE_SIZE,
alpha=PlotStyle.NOISE_ALPHA, label='Unassigned / noise')

    colors = PlotStyle.get_cluster_colors(unique_clusters, PlotStyle.CMAP_SCATTER)
    for cid in unique_clusters:
        mask = (labels == cid)
        ax.scatter(x[mask], y[mask], color=colors[cid], s=PlotStyle.CLUSTER_SIZE, alpha=PlotStyle.CLUSTER_ALPHA,
label=f'Cluster {cid}')

```

```

    PlotStyle.apply_standard_layout(fig, ax, "Conformational Projection", f"Principal Component {p1}", f"Principal
Component {p2}", lx, ly)

    if zero_var:
        ax.text(0.5, 0.5, "Zero-variance projection\n(coincident configurations)", transform=ax.transAxes, ha='center',
va='center', fontsize=14, fontweight='bold', color='gray', alpha=0.7)
    elif len(unique_clusters) <= 35:
        ax.legend(markerscale=5, loc='center left', bbox_to_anchor=PlotStyle.LEGEND_BBOX, framealpha=0.8)

    fig.savefig(f"{out_prefix}.png", dpi=PlotStyle.DPI)
    plt.close(fig)

def render_fes_connectivity(pc1, pc2, unique_clusters, labels, k_B_T, x112, y112, label_changes, top_edges, out_file):
    import matplotlib.pyplot as plt

    hist, xedges, yedges = np.histogram2d(pc1, pc2, bins=150, range=[x112, y112])
    hist = hist.T # Explicit transposition for correct Cartesian rendering

    free_energy = -k_B_T * np.log((hist + 1e-10) / np.max(hist))
    free_energy[free_energy > MAX_FREE_ENERGY_KCAL] = MAX_FREE_ENERGY_KCAL

    X, Y = np.meshgrid((xedges[:-1] + xedges[1:]) / 2, (yedges[:-1] + yedges[1:]) / 2)

    fig, ax = plt.subplots(figsize=PlotStyle.FIG_SIZE, dpi=PlotStyle.DPI)
    contour = ax.contourf(X, Y, free_energy, levels=np.linspace(0, MAX_FREE_ENERGY_KCAL, 16), cmap=PlotStyle.CMAP_FES,
extend='max')
    ax.contour(X, Y, free_energy, levels=np.linspace(0, MAX_FREE_ENERGY_KCAL, 16), colors='black', alpha=0.1,
linewidths=0.5)

    node_positions = {}
    for cid in unique_clusters:
        mask = (labels == cid)
        node_positions[cid] = (np.median(pc1[mask]), np.median(pc2[mask]))

    if top_edges:
        max_w = max([label_changes[e] for e in top_edges]) if top_edges else 1

```

```

# Reverse drawing order ensures highest-count (thickest) edges are rendered on top
for edge in reversed(top_edges):
    sA, sB = edge
    if sA in node_positions and sB in node_positions:
        w = (label_changes[edge] / max_w) * 4.0 + 0.5
        ax.plot([node_positions[sA][0], node_positions[sB][0]], [node_positions[sA][1], node_positions[sB][1]],
color='black', alpha=0.6, linewidth=w, zorder=2)

labels_pos = PlotStyle.repel_labels(node_positions, xl12, yl12)
for cid in unique_clusters:
    nx, ny = node_positions[cid]
    lx, ly = labels_pos[cid]
    if np.linalg.norm(np.array([nx, ny]) - np.array([lx, ly])) > 1e-3:
        ax.plot([nx, lx], [ny, ly], color='black', linewidth=0.5, zorder=3)
        ax.scatter(nx, ny, color='black', s=10, zorder=3)
        ax.text(lx, ly, str(cid), fontsize=9, fontweight='bold', color='black', ha='center', va='center',
bbox=dict(facecolor='white', alpha=0.8, edgecolor='black', boxstyle='circle,pad=0.2'), zorder=4)

PlotStyle.apply_standard_layout(fig, ax, "Population-Derived Free-Energy Landscape\n& Observed Cluster Connectivity",
"Principal Component 1", "Principal Component 2", xl12, yl12)

cax = fig.add_axes(PlotStyle.CBAR_AXES)
cbar = fig.colorbar(contour, cax=cax)
cbar.set_label('Relative free energy (kcal/mol)', **PlotStyle.LABEL_FMT)

if len(top_edges) > 0:
    ax.text(0.02, 0.98, f"{len(top_edges)} highest-count observed cluster-pair changes", transform=ax.transAxes,
ha='left', va='top', fontsize=10, bbox=dict(facecolor='white', alpha=0.8, edgecolor='none'))

fig.savefig(out_file, dpi=PlotStyle.DPI)
plt.close(fig)

def render_pcoa_connectivity(num_states, data_nd, labels, states, population_map, label_changes, top_edges, out_file):
    import matplotlib.pyplot as plt
    from scipy.spatial.distance import squareform, pdist
    cluster_medians = np.array([np.median(data_nd[labels == state], axis=0) for state in states])

```

```

D = squareform(pdist(cluster_medians, metric='euclidean'))
H = np.eye(num_states) - np.ones((num_states, num_states)) / num_states
evals, evects = np.linalg.eigh(-0.5 * H.dot(D**2).dot(H))
idx = np.argsort(evals)[::-1]; evals, evects = evals[idx], evects[:, idx]
pcoa_coords = evects[:, :2] * np.sqrt(np.maximum(evals[:2], 0))
pos_pcoa = {state: (pcoa_coords[i, 0], pcoa_coords[i, 1]) for i, state in enumerate(states)}

pad_x = max((np.max(pcoa_coords[:, 0]) - np.min(pcoa_coords[:, 0])) * 0.15, 0.1)
pad_y = max((np.max(pcoa_coords[:, 1]) - np.min(pcoa_coords[:, 1])) * 0.15, 0.1)
xlim = (np.min(pcoa_coords[:, 0]) - pad_x, np.max(pcoa_coords[:, 0]) + pad_x)
ylim = (np.min(pcoa_coords[:, 1]) - pad_y, np.max(pcoa_coords[:, 1]) + pad_y * 1.5) # Asymmetric top pad

fig, ax = plt.subplots(figsize=PlotStyle.FIG_SIZE, dpi=PlotStyle.DPI)

if top_edges:
    max_w = max([label_changes[e] for e in top_edges]) if top_edges else 1
    for edge in reversed(top_edges):
        sA, sB = edge
        if sA in pos_pcoa and sB in pos_pcoa:
            w = (label_changes[edge] / max_w) * 4.0 + 0.5
            ax.plot([pos_pcoa[sA][0], pos_pcoa[sB][0]], [pos_pcoa[sA][1], pos_pcoa[sB][1]], color='black', alpha=0.6,
                    linewidth=w, zorder=1)

colors = PlotStyle.get_cluster_colors(states, PlotStyle.CMAP_SCATTER)
labels_pos = PlotStyle.repel_labels(pos_pcoa, xlim, ylim)

for state in states:
    cx, cy = pos_pcoa[state]
    lx, ly = labels_pos[state]
    node_size = float(np.log10(population_map[state])) ** 3.5 * 8
    ax.scatter(cx, cy, s=node_size, color=colors[state], edgecolors='white', linewidths=1.5, zorder=2)
    if np.linalg.norm(np.array([cx, cy]) - np.array([lx, ly])) > 1e-3:
        ax.plot([cx, lx], [cy, ly], color='black', linewidth=0.5, zorder=3)
        ax.text(lx, ly, str(state), color='black', fontweight='bold', fontsize=10, ha='center', va='center',
                bbox=dict(facecolor='white', alpha=0.6, edgecolor='none', pad=0.1, boxstyle='round'), zorder=4)

```

```

    PlotStyle.apply_standard_layout(fig, ax, "Principal-Coordinate Representation\nof Cluster Medians & Connectivity",
    "PCoA Dimension 1", "PCoA Dimension 2", xlim, ylim)

    if len(top_edges) > 0:
        ax.text(0.02, 0.98, f"{len(top_edges)} highest-count observed cluster-pair changes", transform=ax.transAxes,
        ha='left', va='top', fontsize=10, bbox=dict(facecolor='white', alpha=0.8, edgecolor='none'))

    fig.savefig(out_file, dpi=PlotStyle.DPI)
    plt.close(fig)

# =====
# 6. STRUCTURAL EXTRACTION (CARMA)
# =====
def parse_atom_extraction_flags(target_dir):
    """Parses original log to maintain extraction atom consistency."""
    log_files = glob.glob(os.path.join(target_dir, 'last_carma_run.log'))
    if not log_files: return FALLBACK_ATOM_FLAGS, "fallback default backbone selection"
    try:
        with open(log_files[0], 'r') as f:
            for line in f:
                if "atoms are declared in the PSF file" in line:
                    if "non-hydrogen" in line: return ["-atmid", "HEAVY"], "parsed from last_carma_run.log (heavy atoms)"
                    parts = line.split()
                    atoms_idx = parts.index("atoms")
                    atom_names = parts[1:atoms_idx]
                    flags = []
                    for atom in atom_names: flags.extend(["-atomid", atom])
                    return flags, "parsed from last_carma_run.log"
    except (FileNotFoundError, ValueError, IndexError) as e:
        logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Error parsing atom IDs ({e}). Reverting to default backbone selection.")
    return FALLBACK_ATOM_FLAGS, "fallback default backbone selection (parsing failed)"

def execute_carma_sandbox(cmd_list, run_dir='.'):
    """Runs carma in an isolated working directory and records its output in the execution log."""

```

```

result = subprocess.run(cmd_list, capture_output=True, text=True, cwd=run_dir)

file_handler = get_clusterMD_file_handler()
if file_handler:
    if result.stdout:
        for line in result.stdout.splitlines():
            clean_line = ANSI_ESCAPE.sub('', line)
            file_handler.stream.write(clean_line + "\n")
    if result.stderr:
        for line in result.stderr.splitlines():
            clean_line = ANSI_ESCAPE.sub('', line)
            file_handler.stream.write("carma ERROR: " + clean_line + "\n")

    if result.returncode != 0:
        file_handler.stream.write(f"carma execution returned non-zero exit code ({result.returncode}) for command: {'
'.join(cmd_list)}\n")

    file_handler.flush()

return result.stdout, result.stderr, result.returncode

def _process_single_cluster(cid, index, total_clusters, labels, frames, data_nd, abs_original_dcd, abs_original_psf,
abs_results_dir, extraction_flags, args, carma_stats, cluster_rmsds, cluster_reps, cluster_struct_counts, fh):
    """Processes one cluster for structural extraction, alignment, filtering, and representative-frame recovery."""
    total_str = f"({index+1:4d}/{total_clusters:4d})"
    sys.stdout.write(f"\r      -> Cluster {cid:4d} {total_str} : Processing structural ensemble...      ")
    sys.stdout.flush()
    if fh:
        fh.stream.write(f"      -> Processing structural ensemble: Cluster {cid:4d}...\n")
        fh.flush()

    cluster_mask = (labels == cid)
    cluster_frames = frames[cluster_mask]
    cluster_coords = data_nd[cluster_mask]
    initial_frame_count = len(cluster_frames)
    cluster_struct_counts[cid] = initial_frame_count

```

```

centroid = np.mean(cluster_coords, axis=0)
distances = np.linalg.norm(cluster_coords - centroid, axis=1)
nearest_centroid_idx = np.argmin(distances)
nearest_centroid_frame = cluster_frames[nearest_centroid_idx]
nearest_centroid_idx_carma = nearest_centroid_idx + 1

with tempfile.TemporaryDirectory(dir=abs_results_dir) as tmp_dir:
    temp_frames_dat = os.path.join(tmp_dir, "temp_frames.dat")
    np.savetxt(temp_frames_dat, cluster_frames, fmt='%d')

    t_sub = time.time()
    _, _, rc_sort = execute_carma_sandbox(["carma", "-v", "-sort", "temp_frames.dat", abs_original_dcd],
run_dir=tmp_dir)
    carma_stats['sort'] += (time.time() - t_sub)

    temp_full_dcd = os.path.join(tmp_dir, "carma.reordered.dcd")
    if rc_sort == 0 and os.path.exists(os.path.join(tmp_dir, "carma.reordered.dcd")):
        os.rename(os.path.join(tmp_dir, "carma.reordered.dcd"), temp_full_dcd)
    else:
        sys.stdout.write(f"\r    -> Cluster {cid:4d} {total_str} : [FAILED] Trajectory slicing error.      \n")
        cluster_reps[cid] = "FAILED"
        return False

    if fh:
        fh.stream.write(f"    -> Phase 1: Aligning cluster ensemble to representative frame...\n")
        t_sub = time.time()
        _, _, rc_cov = execute_carma_sandbox(["carma", "-v", "-w", "-cov"] + extraction_flags + [temp_full_dcd,
abs_original_psf], run_dir=tmp_dir)
        _, _, rc_fit = execute_carma_sandbox(["carma", "-v", "-w", "-fit", "-reference", str(nearest_centroid_idx_carma)]
+ extraction_flags + [temp_full_dcd, abs_original_psf], run_dir=tmp_dir)
        carma_stats['rmsd'] += (time.time() - t_sub)

    if rc_fit != 0:
        sys.stdout.write(f"\r    -> Cluster {cid:4d} {total_str} : [FAILED] Structural alignment error.      \n")
        cluster_reps[cid] = "FAILED"

```

```

    return False

temp_fitted_dcd = os.path.join(tmp_dir, "carma.fitted.dcd")
if os.path.exists(os.path.join(tmp_dir, "carma.fitted.dcd")):
    os.rename(os.path.join(tmp_dir, "carma.fitted.dcd"), temp_fitted_dcd)
else:
    temp_fitted_dcd = temp_full_dcd

pseudo_psf = os.path.join(tmp_dir, "carma.selected_atoms.psf")
safe_psf = os.path.join(tmp_dir, "cluster.psf")
if os.path.exists(pseudo_psf):
    os.rename(pseudo_psf, safe_psf)
else:
    safe_psf = abs_original_psf

if fh:
    fh.stream.write(f"        -> Phase 2: Calculating mathematical average & superposition...\n")
    t_sub = time.time()
    _, _, rc_super = execute_carma_sandbox(["carma", "-v", "-cov", "-dot", "-super"] + extraction_flags +
[temp_fitted_dcd, safe_psf], run_dir=tmp_dir)
    carma_stats['super'] += (time.time() - t_sub)

if rc_super != 0 or not os.path.exists(os.path.join(tmp_dir, "carma.average.pdb")):
    sys.stdout.write(f"\r        -> Cluster {cid:4d} {total_str} : [FAILED] Superposition processing error. \n")
    cluster_reps[cid] = "FAILED"
    return False

super_pdb = os.path.join(tmp_dir, "carma.superposition.pdb")
if os.path.exists(super_pdb):
    os.rename(super_pdb, os.path.join(abs_results_dir, f"cluster_{cid:04d}_superposition.pdb"))
avg_pdb = os.path.join(tmp_dir, "carma.average.pdb")
if os.path.exists(avg_pdb):
    os.rename(avg_pdb, os.path.join(abs_results_dir, f"cluster_{cid:04d}_average.pdb"))

rms_average_dat = os.path.join(tmp_dir, "carma.rms-average.dat")
cluster_rmsds[cid] = -1.0

```

```

if os.path.exists(rms_average_dat):
    try:
        rmsd_data = np.loadtxt(rms_average_dat)
        if rmsd_data.ndim == 1:
            rmsd_data = rmsd_data.reshape(-1, 2)
        rmsd_values = rmsd_data[:, 1]
        cluster_rmsds[cid] = float(np.max(rmsd_values))

    if args.rmsd > 0.0:
        if fh:
            fh.stream.write(f"          -> Phase 3: Applying RMSD-based structural filtering relative to the
average structure (Threshold: {args.rmsd} A)...\n")
            fh.flush()
            core_mask = rmsd_values <= args.rmsd
            retained_count = np.sum(core_mask)

            if retained_count > 0:
                core_cluster_frames = cluster_frames[core_mask]
                if fh:
                    fh.stream.write(f"          -> Filter applied successfully: Retained
{retained_count}/{initial_frame_count} frames in the RMSD-filtered structural ensemble.\n")
                    sys.stdout.write(f"\r          -> Cluster {cid:4d} {total_str} : Retained
{retained_count}/{initial_frame_count} frames after RMSD filtering.          \n")
                    cluster_struct_counts[cid] = int(retained_count)

                np.savetxt(temp_frames_dat, core_cluster_frames, fmt='%d')
                t_sub = time.time()
                _, _, rc_filter = execute_carma_sandbox(["carma", "-v", "-sort", "temp_frames.dat",
abs_original_dcd], run_dir=tmp_dir)
                carma_stats['sort'] += (time.time() - t_sub)

            if rc_filter == 0 and os.path.exists(os.path.join(tmp_dir, "carma.reordered.dcd")):
                core_dcd_path = os.path.join(abs_results_dir, f"cluster_{cid:04d}_core.dcd")
                os.rename(os.path.join(tmp_dir, "carma.reordered.dcd"), core_dcd_path)

```

```

        core_fitted = os.path.join(tmp_dir, "core_fitted.dcd")
        _, _, rc_core_fit = execute_carma_sandbox(["carma", "-v", "-w", "-fit", "-reference", "1"] +
extraction_flags + [core_dcd_path, abs_original_psf], run_dir=tmp_dir)
        if rc_core_fit == 0 and os.path.exists(os.path.join(tmp_dir, "carma.fitted.dcd")):
            os.rename(os.path.join(tmp_dir, "carma.fitted.dcd"), core_fitted)
        else:
            core_fitted = core_dcd_path

        execute_carma_sandbox(["carma", "-v", "-cov", "-dot", "-super"] + extraction_flags +
[core_fitted, safe_psf], run_dir=tmp_dir)

        if os.path.exists(os.path.join(tmp_dir, "carma.superposition.pdb")):
            os.rename(os.path.join(tmp_dir, "carma.superposition.pdb"), os.path.join(abs_results_dir,
f"cluster_{cid:04d}_superposition.pdb"))
        if os.path.exists(os.path.join(tmp_dir, "carma.average.pdb")):
            os.rename(os.path.join(tmp_dir, "carma.average.pdb"), os.path.join(abs_results_dir,
f"cluster_{cid:04d}_average.pdb"))

        rms_core_dat = os.path.join(tmp_dir, "carma.rms-average.dat")
        if os.path.exists(rms_core_dat):
            c_rmsd = np.loadtxt(rms_core_dat)
            if c_rmsd.ndim == 1: c_rmsd = c_rmsd.reshape(-1, 2)
            cluster_rmsds[cid] = float(np.max(c_rmsd[:, 1]))

        cluster_frames = core_cluster_frames
    else:
        sys.stdout.write(f"\r    -> Cluster {cid:4d} {total_str} : [WARNING] 0 frames passed filter.
\n")

        logging.warning(f"\n        {Colors.YELLOW}[WARN]{Colors.RESET} No frames satisfy the RMSD threshold
for Cluster {cid}. Retaining the complete structural ensemble.")
        shutil.copy(temp_fitted_dcd, os.path.join(abs_results_dir, f"cluster_{cid:04d}.dcd"))
    else:
        sys.stdout.write(f"\r    -> Cluster {cid:4d} {total_str} : Extracted {initial_frame_count} frames.
\n")

        shutil.copy(temp_fitted_dcd, os.path.join(abs_results_dir, f"cluster_{cid:04d}.dcd"))
except (ValueError, IndexError, OSError) as e:

```

```

        logging.warning(f"\n          {Colors.YELLOW}[WARN]{Colors.RESET} Core RMSD evaluation failed ({e}) for
Cluster {cid}. Retaining complete ensemble.")
        shutil.copy(temp_fitted_dcd, os.path.join(abs_results_dir, f"cluster_{cid:04d}.dcd"))
    else:
        sys.stdout.write(f"\r      -> Cluster {cid:4d} {total_str} : Extracted {initial_frame_count} frames (No RMSD
log).      \n")
        shutil.copy(temp_fitted_dcd, os.path.join(abs_results_dir, f"cluster_{cid:04d}.dcd"))

    _, _, rc_rep = execute_carma_sandbox(["carma", "-v", "-first", str(nearest_centroid_frame), "-last",
str(nearest_centroid_frame), "-pdb"] + extraction_flags + [abs_original_dcd, abs_original_psf], run_dir=tmp_dir)
    extracted_pdbs = [f for f in glob.glob(os.path.join(tmp_dir, "*.pdb")) if "superposition" not in f and "average"
not in f]

    if rc_rep == 0 and extracted_pdbs:
        os.rename(extracted_pdbs[0], os.path.join(abs_results_dir, f"cluster_{cid:04d}_representative.pdb"))
        cluster_reps[cid] = nearest_centroid_frame
    else:
        sys.stdout.write(f"\r      -> Cluster {cid:4d} {total_str} : [FAILED] Representative structure error.      \n")
        cluster_reps[cid] = "FAILED"
        return False

    sys.stdout.flush()
    if fh:
        fh.flush()
    return True

def execute_structural_extraction(unique_clusters, labels, frames, data_nd, target_analysis_dir, results_dir, args,
carma_stats, cluster_rmsds, cluster_reps, cluster_struct_counts, skip_processing=False):
    """Main loop managing trajectory slicing, alignment, and representative-structure extraction via carma."""
    search_link_dir = target_analysis_dir
    if 'carma_results_' not in target_analysis_dir:
        carma_dirs = glob.glob(os.path.join(target_analysis_dir, 'carma_results_*'))
        if carma_dirs: search_link_dir = max(carma_dirs, key=os.path.getmtime)

    atom_flags, source_str = parse_atom_extraction_flags(search_link_dir)
    segid_flag = ["-segid", args.segid] if args.segid else []

```

```

extraction_flags = atom_flags + segid_flag

dcd_links, psf_links = [], []
for f in os.listdir(search_link_dir):
    full_path = os.path.join(search_link_dir, f)
    if os.path.islink(full_path) and os.path.exists(full_path):
        if f.endswith('.dcd'): dcd_links.append(full_path)
        elif f.endswith('.psf'): psf_links.append(full_path)

dcd_links = sorted(dcd_links)
psf_links = sorted(psf_links)

base_dir = os.path.dirname(search_link_dir) if 'carma_results_' in search_link_dir else search_link_dir
if not base_dir: base_dir = '.'

fallback_dcds = sorted([os.path.join(base_dir, f) for f in os.listdir(base_dir) if f.endswith('.dcd') and not
f.startswith(('carma', 'cluster', 'transitions', 'dPCA'))])
fallback_psfes = sorted([os.path.join(base_dir, f) for f in os.listdir(base_dir) if f.endswith('.psf') and not
f.startswith(('carma', 'cluster', 'transitions', 'dPCA'))])

if len(dcd_links) > 1:
    logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple primary trajectory candidates detected:")
    for cand in dcd_links: logging.warning(f"          {os.path.basename(cand)}")
    logging.warning(f"          Selected: {os.path.basename(dcd_links[0])}")
elif not dcd_links and len(fallback_dcds) > 1:
    logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple fallback trajectory candidates detected:")
    for cand in fallback_dcds: logging.warning(f"          {os.path.basename(cand)}")
    logging.warning(f"          Selected: {os.path.basename(fallback_dcds[0])}")

if len(psf_links) > 1:
    logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple primary topology candidates detected:")
    for cand in psf_links: logging.warning(f"          {os.path.basename(cand)}")
    logging.warning(f"          Selected: {os.path.basename(psf_links[0])}")
elif not psf_links and len(fallback_psfes) > 1:
    logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple fallback topology candidates detected:")
    for cand in fallback_psfes: logging.warning(f"          {os.path.basename(cand)}")

```

```

        logging.warning(f"          Selected: {os.path.basename(fallback_psfs[0])}")

original_dcd = dcd_links[0] if dcd_links else (fallback_dcds[0] if fallback_dcds else None)
original_psf = psf_links[0] if psf_links else (fallback_psfs[0] if fallback_psfs else None)

abs_original_dcd = os.path.abspath(original_dcd) if original_dcd and os.path.exists(original_dcd) else None
abs_original_psf = os.path.abspath(original_psf) if original_psf and os.path.exists(original_psf) else None
abs_results_dir = os.path.abspath(results_dir)

traj_discovery = "symlink in carma results directory" if dcd_links else "parent-directory fallback"
top_discovery = "symlink in carma results directory" if psf_links else "parent-directory fallback"

has_struct = bool(abs_original_dcd and abs_original_psf)

struct_info = {
    'dcd': abs_original_dcd if abs_original_dcd else "N/A",
    'psf': abs_original_psf if abs_original_psf else "N/A",
    'traj_discovery': traj_discovery if abs_original_dcd else "N/A",
    'top_discovery': top_discovery if abs_original_psf else "N/A",
    'flags': " ".join(atom_flags).replace('-atomid', '').replace('-atmid', '').strip() if has_struct else "N/A",
    'source': source_str if has_struct else "not evaluated (no structural trajectory/topology)",
    'compatible': True,
    'bounds_valid': True,
    'preflight_error': None,
    'success_count': 0,
    'failed_count': 0
}

if skip_processing:
    return struct_info

if not shutil.which("carma"):
    logging.warning(f"\n{Colors.YELLOW}[WARN]{Colors.RESET} 'carma' binary not found in system $PATH. Bypassing
structural extraction.")
    return struct_info

```

```

if not has_struct:
    logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Missing fundamental trajectory (.dcd) or topology (.psf)
files in {base_dir}. Bypassing extraction phase.")
    return struct_info

logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Trajectory file: {original_dcd}")
logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Topology file: {original_psf}\n")

# PREFLIGHT ATOM SELECTION VERIFICATION
requested_atoms = set()
for i in range(len(extraction_flags) - 1):
    if extraction_flags[i] == '-atomid':
        requested_atoms.add(extraction_flags[i+1])

if abs_original_psf:
    psf_atoms_present = set()
    psf_segments_present = set()
    try:
        with open(abs_original_psf, 'r') as f:
            in_atoms_section = False
            for line in f:
                if '!NATOM' in line:
                    in_atoms_section = True
                    continue
                if in_atoms_section:
                    line_strip = line.strip()
                    if not line_strip or '!NBOND' in line_strip:
                        break
                    parts = line.split()
                    if len(parts) >= 5 and parts[0].isdigit():
                        psf_segments_present.add(parts[1])
                        psf_atoms_present.add(parts[4])
    except Exception:
        pass # Fallback to carma preflight if parse fails

if args.segid and psf_segments_present and args.segid not in psf_segments_present:

```

```

        logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Requested segment '{args.segid}' was not
found in the topology.")
        struct_info['compatible'] = False
        struct_info['preflight_error'] = 'missing_segment'
        struct_info['missing_segment'] = args.segid
        return struct_info

    if requested_atoms and psf_atoms_present:
        missing_atoms = requested_atoms - psf_atoms_present
        if missing_atoms:
            missing_str = " ".join(sorted(list(missing_atoms)))
            logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Structural atom selection contains atom
identifiers not present in the topology.")
            logging.error(f"      Missing identifiers : {missing_str}")
            logging.error(f"      Requested selection : {struct_info['flags']}")
            struct_info['compatible'] = False
            struct_info['preflight_error'] = 'missing_atoms'
            struct_info['missing_atoms'] = missing_str
            return struct_info

    # PREFLIGHT DCD/PSF CHECK & BOUNDS VALIDATION
    logging.info(f"      {Colors.LIGHT_BLUE}[*]{Colors.RESET} Executing preflight DCD/PSF compatibility check...")
    max_sampled_frame = int(np.max(frames))
    dcd_frames = None
    dcd_atoms = None
    psf_atoms = None

    with tempfile.TemporaryDirectory(dir=abs_results_dir) as tmp_dir:
        out, err, rc_preflight = execute_carma_sandbox(["carma", "-v", "-first", "1", "-last", "1", "-pdb"] +
extraction_flags + [abs_original_dcd, abs_original_psf], run_dir=tmp_dir)
        combined_out = out + "\n" + err
        combined_lower = combined_out.lower()

    for line in combined_out.splitlines():
        line_lower = line.lower()
        if 'coordinate sets' in line_lower and dcd_frames is None:

```

```

        m = re.search(r'coordinate sets\s+is\s+(\d+)', line_lower)
        if m: dcd_frames = int(m.group(1))

    m_mismatch = re.search(r'number of atoms in psf and dcd do not match\s*\((\d+)\s+vs\s+(\d+)\)', line_lower)
    if m_mismatch:
        psf_atoms = int(m_mismatch.group(1))
        dcd_atoms = int(m_mismatch.group(2))

    m_dcd = re.search(r'number of atoms in sets is\s+(\d+)', line_lower)
    if m_dcd and dcd_atoms is None:
        dcd_atoms = int(m_dcd.group(1))

    struct_info['dcd_atoms'] = dcd_atoms
    struct_info['psf_atoms'] = psf_atoms
    struct_info['dcd_frames'] = dcd_frames
    struct_info['max_frame'] = max_sampled_frame

    if rc_preflight != 0:
        struct_info['compatible'] = False
        struct_info['preflight_error'] = 'generic'

        if 'less than two' in combined_lower and 'atoms found' in combined_lower:
            struct_info['preflight_error'] = 'selection'
            if args.segid:
                logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Requested segment '{args.segid}' or
selection '{struct_info['flags']}' matched fewer than the required atoms.")
            else:
                logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Structural atom selection
'{struct_info['flags']}' matched fewer than the required atoms.")
        elif m_mismatch or (dcd_atoms and psf_atoms and dcd_atoms != psf_atoms):
            struct_info['preflight_error'] = 'mismatch'
            logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Structural trajectory and topology are
incompatible.")
            logging.error(f"      Trajectory atom count : {dcd_atoms}")
            logging.error(f"      Topology atom count   : {psf_atoms}")
            logging.error(f"      {Colors.RED}[-]{Colors.RESET} DCD/PSF atom counts differ.")

```

```

        else:
            logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Structural preflight failed; see
execution log.")
            return struct_info

        if dcd_frames is not None and max_sampled_frame > dcd_frames:
            logging.error(f"      {Colors.RED}[-]{Colors.RESET} Critical error: Maximum sampled frame ID
({max_sampled_frame}) exceeds available DCD coordinate sets ({dcd_frames}).")
            struct_info['bounds_valid'] = False
            return struct_info

        logging.info(f"      {Colors.BLUE}[+]{Colors.RESET} Preflight check passed: structural inputs, atom selection, and
frame bounds are valid.")

    total_clusters = len(unique_clusters)
    fh = get_clusterMD_file_handler()

    # Cluster extraction loop
    for index, cid in enumerate(unique_clusters):
        success = _process_single_cluster(
            cid, index, total_clusters, labels, frames, data_nd,
            abs_original_dcd, abs_original_psf, abs_results_dir,
            extraction_flags, args, carma_stats, cluster_rmsds, cluster_reps, cluster_struct_counts, fh
        )
        if success:
            struct_info['success_count'] += 1
        else:
            struct_info['failed_count'] += 1

    logging.info(f"\n      {Colors.LIGHT_BLUE}[*]{Colors.RESET} Processing unassigned/noise configurations (label -1)...")
    noise_mask = (labels == -1)
    if np.any(noise_mask):
        with tempfile.TemporaryDirectory(dir=abs_results_dir) as tmp_dir:
            noise_file = os.path.join(tmp_dir, "temp_frames_noise.dat")
            np.savetxt(noise_file, frames[noise_mask], fmt='%d')
            t_sub = time.time()

```

```

_, _, rc_noise = execute_carma_sandbox(["carma", "-v", "-sort", "temp_frames_noise.dat", abs_original_dcd],
run_dir=tmp_dir)
    carma_stats['sort'] += (time.time() - t_sub)
    if rc_noise == 0 and os.path.exists(os.path.join(tmp_dir, "carma.reordered.dcd")):
        os.rename(os.path.join(tmp_dir, "carma.reordered.dcd"), os.path.join(abs_results_dir,
"unassigned_noise.dcd"))

    logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Structural extraction phase concluded.")
    return struct_info

# =====
# 7. PIPELINE EXECUTION
# =====
def get_shared_limits(pc_data, pad_ratio=0.05, min_pad=0.1):
    """Calculates shared axis bounds robust to zero-variance input."""
    ptp = np.ptp(pc_data)
    if ptp == 0:
        return (pc_data[0] - min_pad, pc_data[0] + min_pad), True
    pad = max(ptp * pad_ratio, min_pad)
    return (np.min(pc_data) - pad, np.max(pc_data) + pad), False

def run_full_pipeline(args, data_file, results_dir, target_analysis_dir, command_line):
    """Executes the complete clustering, observed-label-change, visualization, and structural-extraction sequence."""
    import matplotlib
    matplotlib.use('Agg') # Safe headless execution

    pipeline_start = time.time()
    process = psutil.Process(os.getpid())
    timing_stats = {'load': 0, 'hdbscan': 0, 'export': 0, 'plots': 0, 'extract': 0}
    carma_stats = {'sort': 0, 'rmsd': 0, 'super': 0}
    cluster_rmsds = {}
    cluster_reps = {}
    cluster_struct_counts = {}

    # 1. LOADING

```

```

t0 = time.time()
raw_data, parse_stats = ingest_trajectory_data(data_file, step=args.step)

if len(raw_data) < 2:
    logging.error(f"{Colors.RED}[-]{Colors.RESET} Critical error: Insufficient valid frames sampled ({len(raw_data)}).
Minimum 2 required.")
    sys.exit(1)

if args.step > 1:
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Applied frame stride of {args.step} during ingestion.")

frames = raw_data[:, 0].astype(int)
data_nd = raw_data[:, 1:]
available_dims = data_nd.shape[1]

requested_dim = args.dim
if args.dim > 0:
    if args.dim > available_dims:
        logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Requested --dim {args.dim} exceeds available dimensions
({available_dims}). Capping to {available_dims}.")
        args.dim = available_dims
    data_nd = data_nd[:, :args.dim]
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Constraining multidimensional analysis to the primary
{args.dim} dimensions.")

n_dimensions = data_nd.shape[1]
total_frames = data_nd.shape[0]
timing_stats['load'] = time.time() - t0
logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Data ingested successfully: {total_frames:,} frames ({n_dimensions}D).")

if args.auto:
    scale_factor = math.sqrt(total_frames / 1_000_000.0)
    dynamic_size = max(100, int(500 * scale_factor))
    dynamic_samples = max(10, int(dynamic_size / 10.0))
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Auto-scaling activated: min_cluster_size={dynamic_size}")
else:

```

```

dynamic_size = args.size
dynamic_samples = args.samples if args.samples is not None else max(2, int(dynamic_size / 10.0))

auto_status_console = "Active" if args.auto else "Inactive (manual parameters)"
logging.info("\n" + "="*70)
logging.info(f"{Colors.BOLD} ANALYSIS SETUP{Colors.RESET}")
logging.info("="*70)
logging.info(f"  Projection file      : {os.path.basename(data_file)}")
logging.info(f"  Frames analyzed      : {total_frames:,}")
logging.info(f"  Frame stride         : {args.step}")
logging.info(f"  Dimensions used      : {n_dimensions}")
logging.info(f"  Temperature         : {args.temp} K")
logging.info(f"  HDBSCAN method       : {args.method}")
logging.info(f"  min_cluster_size     : {dynamic_size}")
logging.info(f"  min_samples         : {dynamic_samples}")
logging.info(f"  Auto-scaling         : {auto_status_console}")
logging.info("="*70)

# 2. CLUSTERING
logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Running HDBSCAN clustering...")
labels, timing_stats['hdbscan'] = execute_clustering_algorithm(data_nd, dynamic_size, dynamic_samples, args.method)

initial_cluster_count = len(np.unique(labels[labels != -1]))
logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Identified {initial_cluster_count} clusters in
{timing_stats['hdbscan']:.2f} seconds.")

logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Sorting cluster assignments by descending population size...")
labels, unique_clusters = sort_cluster_populations(labels)

noise_frames = np.sum(labels == -1)
assigned_frames = total_frames - noise_frames
logging.info("\n    CLUSTER POPULATIONS")
logging.info("    " + "-"*49)
logging.info(f"    {'Cluster':<10}{'Frames':>12}{'Population':>15}")
logging.info("    " + "-"*49)
for cid in unique_clusters:

```

```

        count = np.sum(labels == cid)
        logging.info(f"        {cid:<10d}{count:>12,d} {(count / total_frames) * 100:>14.1f}%")
logging.info(f"        " + "-"*49)
logging.info(f"        {'Unassigned':<10}{noise_frames:>12,d} {(noise_frames / total_frames) * 100:>14.1f}%")
logging.info(f"        " + "-"*49)
logging.info(f"        Assigned frames      : {assigned_frames:,} ({(assigned_frames / total_frames) * 100:.1f}%)")
logging.info(f"        Unassigned / noise   : {noise_frames:,} ({(noise_frames / total_frames) * 100:.1f}%)")
logging.info(f"        Clusters identified: {len(unique_clusters)}")

t0 = time.time()
out_filename = os.path.join(results_dir, f"hdbscan.{n_dimensions}-D.clusters.dat")
logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Exporting assigned coordinate matrices to {out_filename}...")
export_cluster_coordinates(frames, labels, data_nd, out_filename, n_dimensions)
timing_stats['export'] = time.time() - t0

has_duplicates = parse_stats['sampled_duplicates']
non_monotonic = not parse_stats['sampled_monotonic']
fatal_frame_error = has_duplicates or non_monotonic

skip_label_changes = non_monotonic or has_duplicates
skip_structural = has_duplicates

if has_duplicates:
    logging.warning(f"\n{Colors.YELLOW}[WARN]{Colors.RESET} Duplicate sampled frame identifiers detected.")
    logging.warning(f"        Clustering assignments are valid in coordinate space,")
    logging.warning(f"        but one-to-one structural frame mapping is ambiguous.")
if non_monotonic:
    prefix = "" if has_duplicates else "\n"
    logging.warning(f"{prefix}{Colors.YELLOW}[WARN]{Colors.RESET} Sampled frame identifiers are not strictly
increasing.")
    logging.warning(f"        Clustering assignments are valid in coordinate space,")
    logging.warning(f"        but chronological interpretation of this assignment file is not valid.")

if args.cluster_only:
    elapsed = time.time() - pipeline_start
    logging.info(f"\n{Colors.BLUE}[+]{Colors.RESET} Cluster-only mode complete.")

```

```

logging.info(f"      Frames analyzed      : {total_frames:,}")
logging.info(f"      Clusters identified   : {len(unique_clusters)}")
logging.info(f"      Assigned frames      : {assigned_frames:,} ({assigned_frames / total_frames * 100:.2f}%)")
logging.info(f"      Unassigned / noise   : {noise_frames:,} ({noise_frames / total_frames * 100:.2f}%)")
logging.info(f"      HDBSCAN runtime      : {timing_stats['hdbscan']:.2f}s")
logging.info(f"      Assignment file      : {out_filename}")
logging.info(f"      Total runtime        : {elapsed:.2f}s")
return

if has_duplicates:
    logging.error(f"\n{Colors.RED}[-]{Colors.RESET} Critical error: Duplicate sampled frame identifiers detected.")
if non_monotonic:
    prefix = "" if has_duplicates else "\n"
    logging.error(f"{prefix}{Colors.RED}[-]{Colors.RESET} Critical error: Sampled frame identifiers are not strictly
increasing.")

if skip_label_changes and skip_structural:
    logging.warning(f"{Colors.YELLOW}[*]{Colors.RESET} Bypassing observed label-change analysis, connectivity
visualizations, and structural extraction.")
    elif skip_label_changes:
        logging.warning(f"{Colors.YELLOW}[*]{Colors.RESET} Bypassing observed label-change analysis and connectivity
visualizations.")

# 3. OBSERVED LABEL CHANGES
logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Mapping observed changes between assigned cluster labels...")
states, counts, population_map, _, k_B_T = calculate_populations_and_energies(labels, args.temp)
num_states = len(states)

if not skip_label_changes:
    label_changes = map_observed_label_changes(labels)
    sorted_changes = print_top_observed_changes(label_changes, args.top)
    top_edges = [edge for edge, count in sorted_changes[:args.top]]

    changes_file = os.path.join(results_dir, "observed_label_changes_report.txt")
    with open(changes_file, "w") as cf:
        cf.write("=====\n")

```

```

cf.write("OBSERVED LABEL CHANGES\n")
cf.write("=====\n\n")
cf.write(f"{'Cluster pair':<40}{'Observed changes':>20}\n")
cf.write("-" * 60 + "\n")
for edge, count in sorted_changes:
    pair_str = f"{edge[0]} <--> {edge[1]}"
    cf.write(f"{pair_str:<40}{count:>20d}\n")
cf.write("-" * 60 + "\n")
cf.write(f"{'Unique cluster pairs':<40}{len(label_changes):>20d}\n")
logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Observed-change report saved to: {changes_file}")

if args.transitions is not None and len(args.transitions) == 2:
    logging.info("\n" + "="*85)
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} DIAGNOSTIC MODE: INSPECTING OBSERVED CHANGES FOR CLUSTER
PAIR {tuple(sorted(args.transitions))}")
    logging.info("="*85)
    hunt_chronological_label_changes(tuple(sorted(args.transitions)), labels, frames, data_nd, args.events)
else:
    label_changes, top_edges = {}, []
    logging.info(f"    -> Skipped due to chronological validation failure.")

# 4. VISUALIZATION
plot_status = {
    'landscape': 'Completed',
    'fes_connectivity': 'Completed',
    'pcoa_connectivity': 'Completed'
}

logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Generating static conformational projections and free-energy
surfaces...")
t0_plots = time.time()

pc1 = data_nd[:, 0]
pc2 = data_nd[:, 1] if n_dimensions >= 2 else None

if n_dimensions >= 2:

```

```

xlim, x_degen = get_shared_limits(pc1)
ylim, y_degen = get_shared_limits(pc2)
zero_var_2d = x_degen or y_degen

render_conformational_landscape(pc1, pc2, xlim, ylim, "1", "2", unique_clusters, labels, zero_var_2d,
os.path.join(results_dir, "landscape_PC1_vs_PC2"))

if zero_var_2d:
    logging.warning(f"      {Colors.YELLOW}[WARN]{Colors.RESET} Zero variance detected in primary dimensions.
Bypassing free-energy surface connectivity overlay.")
    plot_status['fes_connectivity'] = 'Skipped (zero variance in projection)'
else:
    if num_states > 0:
        if skip_label_changes:
            logging.info("      -> Rendering population-derived FES without connectivity...")
        else:
            logging.info("      -> Rendering FES with observed cluster connectivity...")
            render_fes_connectivity(pc1, pc2, unique_clusters, labels, k_B_T, xlim, ylim, label_changes, top_edges,
os.path.join(results_dir, "FES_Connectivity_PC1_vs_PC2.png"))
            if skip_label_changes:
                plot_status['fes_connectivity'] = 'FES completed; connectivity not performed (chronological validation
failure)'
            else:
                logging.info(f"      -> Bypassing FES connectivity overlay (0 clusters assigned).")
                plot_status['fes_connectivity'] = 'Skipped (0 clusters assigned)'
    else:
        logging.info(f"      -> Bypassing 2D visualizations (1D input detected).")
        for p in ['landscape', 'fes_connectivity']:
            plot_status[p] = 'Skipped (1D input)'

if n_dimensions >= 3:
    zlim, z_degen = get_shared_limits(data_nd[:, 2])
    if not (x_degen or z_degen):
        render_conformational_landscape(data_nd[:, 0], data_nd[:, 2], xlim, zlim, "1", "3", unique_clusters, labels,
False, os.path.join(results_dir, "landscape_PC1_vs_PC3"))
        if not (y_degen or z_degen):

```

```

        render_conformational_landscape(data_nd[:, 1], data_nd[:, 2], ylim, zlim, "2", "3", unique_clusters, labels,
False, os.path.join(results_dir, "landscape_PC2_vs_PC3"))

    if not skip_label_changes:
        if num_states > 2:
            logging.info(f"    -> Rendering principal coordinate analysis (PCoA) of cluster medians...")
            render_pcoa_connectivity(num_states, data_nd, labels, states, population_map, label_changes, top_edges,
os.path.join(results_dir, "PCoA_Connectivity_Cluster_Medians.png"))
        else:
            plot_status['pcoa_connectivity'] = 'Skipped (requires >2 clusters)'
    else:
        plot_status['pcoa_connectivity'] = 'Not performed (chronological validation failure)'

    timing_stats['plots'] = time.time() - t0_plots

# 5. EXTRACTING
if not skip_structural:
    logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Extracting cluster-associated structural ensembles via
carma...")
    t0_extract = time.time()
    struct_info = execute_structural_extraction(unique_clusters, labels, frames, data_nd, target_analysis_dir,
results_dir, args, carma_stats, cluster_rmsds, cluster_reps, cluster_struct_counts, skip_processing=False)
    if not struct_info: struct_info = {}
    timing_stats['extract'] = time.time() - t0_extract
else:
    struct_info = execute_structural_extraction(unique_clusters, labels, frames, data_nd, target_analysis_dir,
results_dir, args, carma_stats, cluster_rmsds, cluster_reps, cluster_struct_counts, skip_processing=True)
    if not struct_info: struct_info = {}

# 6. REPORT GENERATION
logging.info(f"\n{Colors.LIGHT_BLUE}[*]{Colors.RESET} Writing detailed analysis report...")

fatal_struct_error = False
if struct_info.get('dcd') and struct_info.get('dcd') != 'N/A':
    if struct_info.get('compatible') is False or struct_info.get('bounds_valid') is False:
        fatal_struct_error = True

```

```

        elif struct_info.get('failed_count', 0) > 0:
            fatal_struct_error = True

overall_fatal_error = fatal_frame_error or fatal_struct_error

generate_execution_report(
    results_dir, pipeline_start, process, total_frames, args, n_dimensions, available_dims, requested_dim,
    dynamic_size, dynamic_samples, target_analysis_dir, data_file, timing_stats,
    carma_stats, unique_clusters, labels, cluster_rmsds, cluster_struct_counts, parse_stats, struct_info,
    cluster_reps,
    command_line, skip_label_changes, skip_structural, fatal_frame_error, fatal_struct_error, plot_status
)

if overall_fatal_error:
    sys.exit(1)

def run_diagnostic_validation(args, dat_file):
    """Bypasses the full pipeline and inspects chronological observed label changes."""
    raw_data, _ = ingest_trajectory_data(dat_file, step=1)

    frames = raw_data[:, 0].astype(int)
    labels = raw_data[:, 1].astype(int)
    data_nd = raw_data[:, 2:]

    label_changes = map_observed_label_changes(labels)

    logging.info("\n" + "="*85)
    if len(args.transitions) == 2:
        logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} DIAGNOSTIC MODE: INSPECTING OBSERVED CHANGES FOR CLUSTER PAIR
{tuple(sorted(args.transitions))}")
    else:
        logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} DIAGNOSTIC MODE: OBSERVED CHANGE OVERVIEW")
        logging.info("="*85)

    print_top_observed_changes(label_changes, args.top)

```

```

if len(args.transitions) == 2:
    hunt_chronological_label_changes(tuple(sorted(args.transitions)), labels, frames, data_nd, args.events)

    logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Validation complete. Exiting smoothly without triggering extraction
pipeline.\n")
    sys.exit(0)

def generate_execution_report(results_dir, pipeline_start, process, total_frames, args, n_dimensions, available_dims,
requested_dim, dynamic_size, dynamic_samples, target_analysis_dir, data_file, timing_stats, carma_stats, unique_clusters,
labels, cluster_rmsds, cluster_struct_counts, parse_stats, struct_info, cluster_reps, command_line,
skip_label_changes=False, skip_structural=False, fatal_frame_error=False, fatal_struct_error=False, plot_status=None):

    if plot_status is None:
        plot_status = {}

    def calculate_variance_explained(dim_used, target_dir):
        eig_files = glob.glob(os.path.join(target_dir, '*eigenvalues.dat'))
        if not eig_files: return "N/A"

        eig_file = max(eig_files, key=os.path.getmtime)
        if len(eig_files) > 1:
            logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple eigenvalue files found.")
            logging.warning(f"          Selected: {os.path.basename(eig_file)}")

        try:
            eigenvalues = np.loadtxt(eig_file)
            total_variance = np.sum(eigenvalues)
            used_variance = np.sum(eigenvalues[:dim_used])
            return f"{{(used_variance / total_variance) * 100:.1f}}%"
        except (FileNotFoundError, ValueError, IndexError) as e:
            logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Unable to calculate represented variance ({{e}}).")
            return "Error"

    report_filename = os.path.join(results_dir, "clusterMD_analysis_report.txt")
    total_pipeline_time = time.time() - pipeline_start
    resident_ram_gb = process.memory_info().rss / (1024 ** 3)

```

```

cpu_cores = psutil.cpu_count(logical=True)
variance_str = calculate_variance_explained(n_dimensions, target_analysis_dir)
noise_frames = np.sum(labels == -1)
assigned_frames = total_frames - noise_frames
auto_status = "Active (sub-linear heuristic)" if args.auto else "Inactive (manual parameters)"
env = get_software_env()

carma_missing = env['carma_path'] == 'NOT FOUND'
has_struct = struct_info.get('dcd') and struct_info.get('dcd') != 'N/A'

with open(report_filename, "w") as report:
    report.write("=====\n")
    report.write("clusterMD ANALYSIS REPORT\n")
    report.write("=====\n\n")

    report.write("--- RUN INVOCATION ---\n")
    report.write(f"Command line : {command_line}\n\n")

    report.write("--- SOFTWARE ENVIRONMENT ---\n")
    report.write(f"clusterMD version      : {env['clusterMD_ver']}\n")
    report.write(f"Python version          : {env['python_ver']}\n")
    report.write(f"fast_hdbscan version    : {env['fh_ver']}\n")
    report.write(f"NumPy version           : {env['numpy_ver']}\n")
    report.write(f"SciPy version           : {env['scipy_ver']}\n")
    report.write(f"carma executable        : {env['carma_path']}\n")
    report.write(f"carma version           : {env['carma_ver']}\n\n")

    report.write("--- INPUT PROVENANCE ---\n")
    report.write(f"Projection file          : {os.path.basename(data_file)}\n")
    report.write(f"Projection absolute path : {os.path.abspath(data_file)}\n")
    report.write(f"Input lines detected     : {parse_stats['total_lines']}\n")
    report.write(f"Rows selected by stride  : {parse_stats['selected_by_stride']}\n")
    report.write(f"Valid sampled rows       : {parse_stats['valid_sampled']}\n")
    report.write(f"Malformed sampled rows skipped: {parse_stats['malformed_sampled']}\n")
    report.write(f"Rows entering HDBSCAN   : {parse_stats['valid_sampled']}\n")

```

```

req_dim_str = f"{requested_dim}" if requested_dim > 0 else "ALL (dim=0)"
report.write(f"Requested dimensions           : {req_dim_str}\n")
report.write(f"Available dimensions           : {available_dims}\n")
report.write(f"Dimensions analyzed           : {n_dimensions}\n")

report.write(f"First sampled frame ID           : {parse_stats['first_sampled_frame']}\n")
report.write(f>Last sampled frame ID           : {parse_stats['last_sampled_frame']}\n")
report.write(f"Sampled frame IDs monotonic       : {'YES' if parse_stats['sampled_monotonic'] else 'NO'}\n")
report.write(f"Duplicate sampled frame IDs       : {'YES' if parse_stats['sampled_duplicates'] else 'NO'}\n")
report.write(f"Frame stride                     : {args.step}\n\n")

report.write("--- STRUCTURAL INPUTS ---\n")
report.write(f"Trajectory file                   : {struct_info.get('dcd', 'N/A')}\n")
report.write(f"Topology file                     : {struct_info.get('psf', 'N/A')}\n")
report.write(f"Trajectory discovery              : {struct_info.get('traj_discovery', 'N/A')}\n")
report.write(f"Topology discovery                : {struct_info.get('top_discovery', 'N/A')}\n")
report.write(f"Structural atom selection         : {struct_info.get('flags', 'N/A')}\n")
report.write(f"Selection source                  : {struct_info.get('source', 'N/A')}\n\n")

report.write("--- VALIDATION & ASSUMPTIONS ---\n")
report.write(f"[PASS] Projection file parsed successfully.\n")

if parse_stats['malformed_sampled'] > 0:
    malformed_pct = (parse_stats['malformed_sampled'] / max(1, parse_stats['selected_by_stride'])) * 100
    report.write(f"[WARN] {parse_stats['malformed_sampled']} of {parse_stats['selected_by_stride']} sampled input
rows ({malformed_pct:.2f}%) were malformed and skipped.\n")

report.write(f"[PASS] {parse_stats['valid_sampled']} valid sampled rows entered clustering.\n")

if parse_stats['sampled_monotonic']:
    report.write("[PASS] Sampled frame identifiers are strictly increasing.\n")
else:
    report.write("[FAIL] Sampled frame identifiers are not strictly increasing; chronological label-change
analysis would not represent trajectory order.\n")

if not parse_stats['sampled_duplicates']:

```

```

        report.write("[PASS] No duplicate frame identifiers detected.\n")
    else:
        report.write("[FAIL] Duplicate sampled frame identifiers detected. Each projection row must map unambiguously
to one trajectory frame.\n")

    report.write(f"[PASS] No non-finite PCA/dPCA coordinates were detected in sampled rows.\n")

    if requested_dim == 0:
        report.write(f"[PASS] All available dimensions were used as requested.\n")
    elif requested_dim > available_dims:
        report.write(f"[WARN] Requested dimensionality ({requested_dim}) exceeded available dimensions
({available_dims}); analysis was capped to {n_dimensions} dimensions.\n")
    else:
        report.write(f"[PASS] Requested dimensionality ({requested_dim}) is available.\n")

    if not carma_missing:
        report.write(f"[PASS] carma executable found: {env['carma_path']}\n")
        report.write(f"[PASS] carma version detected: {env['carma_ver']}\n")
    else:
        report.write(f"[WARN] carma executable NOT FOUND.\n")

    report.write(f"[{'PASS' if has_struct else 'WARN'}] Structural trajectory and topology {'located' if has_struct
else 'NOT FOUND'}.\n")

    if carma_missing:
        report.write(f"[INFO] DCD/PSF compatibility was not assessed because carma is unavailable.\n")
        report.write(f"[INFO] Trajectory frame bounds were not assessed because carma is unavailable.\n")
    elif has_struct and not skip_structural:
        if struct_info.get('compatible') is False:
            err_type = struct_info.get('preflight_error')
            if err_type == 'selection':
                if args.segid:
                    report.write(f"[FAIL] Requested segment '{args.segid}' or selection '{struct_info.get('flags')}'
matched fewer than the required atoms. Structural extraction aborted.\n")
                else:

```

```

        report.write(f"[FAIL] Structural atom selection '{struct_info.get('flags')}' matched fewer than
the required atoms. Structural extraction aborted.\n")
    elif err_type == 'missing_segment':
        report.write(f"[FAIL] Requested segment '{struct_info.get('missing_segment')}' was not found in the
topology.\n")
        report.write(f"        Structural extraction aborted.\n")
        report.write(f"[INFO] DCD/PSF compatibility and trajectory frame bounds were not assessed because
segment validation failed.\n")
    elif err_type == 'missing_atoms':
        report.write(f"[FAIL] Structural atom selection contains atom identifiers not present in the
topology.\n")
        report.write(f"        Missing identifiers : {struct_info.get('missing_atoms')}\n")
        report.write(f"        Requested selection : {struct_info.get('flags')}\n")
        report.write(f"        Structural extraction aborted.\n")
        report.write(f"[INFO] DCD/PSF compatibility and trajectory frame bounds were not assessed because
atom-selection validation failed.\n")
    elif err_type == 'mismatch':
        report.write(f"        Trajectory atom count : {struct_info.get('dcd_atoms')}\n")
        report.write(f"        Topology atom count   : {struct_info.get('psf_atoms')}\n")
        report.write(f"[FAIL] DCD/PSF atom counts differ. Structural extraction aborted.\n")
    else:
        report.write(f"[FAIL] Structural preflight failed; see execution log. Structural extraction
aborted.\n")

    if err_type not in ('missing_atoms', 'missing_segment'):
        if struct_info.get('bounds_valid') is False:
            report.write(f"[FAIL] Maximum sampled frame ID ({struct_info.get('max_frame')}) exceeds available
DCD coordinate sets ({struct_info.get('dcd_frames')}).\n")
            report.write(f"        Structural extraction aborted.\n")
            elif struct_info.get('dcd_frames') is None:
                report.write(f"[INFO] Trajectory frame bounds were not assessed due to structural preflight
failure.\n")
        else:
            report.write(f"[PASS] DCD/PSF compatibility check passed.\n")

    if struct_info.get('bounds_valid') is False:

```

```

        report.write(f"[FAIL] Maximum sampled frame ID ({struct_info.get('max_frame')}) exceeds available DCD
coordinate sets ({struct_info.get('dcd_frames')}).\n")
        report.write("        Structural extraction aborted.\n")
        elif struct_info.get('bounds_valid') is True:
            report.write(f"[PASS] Sampled frame identifiers are within available trajectory bounds.\n")
report.write("\n")

report.write("[INFO] Cluster IDs are reordered by descending population.\n")
report.write("[INFO] Label -1 denotes HDBSCAN-unassigned configurations.\n")

if skip_label_changes:
    report.write("[INFO] Observed label-change analysis and connectivity visualizations were not performed because
sampled frame identifiers failed chronological validation.\n")
else:
    report.write("[INFO] Observed label changes are counted after omitting unassigned frames.\n")

if skip_structural:
    report.write("[INFO] Structural representative extraction was not performed due to duplicate frame
identifiers.\n")

if args.rmsd > 0:
    report.write("[INFO] Representative structures refer to the original HDBSCAN cluster and are selected before
optional downstream RMSD filtering.\n")

if skip_structural or not has_struct or fatal_struct_error or carma_missing:
    report.write("[INFO] When structural extraction is performed, representative structures are observed
trajectory frames nearest the arithmetic centroid in retained coordinate space.\n\n")
else:
    report.write("[INFO] Representative structures are observed trajectory frames nearest the arithmetic centroid
in retained coordinate space.\n\n")

if has_struct:
    report.write("[WARN] Projection-to-DCD frame correspondence is inferred from supplied frame identifiers and
cannot be independently proven.\n")
else:

```

```

report.write("[INFO] Projection-to-trajectory frame correspondence was not assessed because no structural
trajectory was located.\n")

report.write("[WARN] Population-derived free-energy estimates assume representative equilibrium sampling and
should not be interpreted as kinetic barriers.\n\n")

report.write("--- ANALYSIS CONFIGURATION ---\n")
report.write(f"Cumulative variance represented by analyzed components : {variance_str}\n")
report.write(f"Temperature used for free-energy estimates           : {args.temp} K\n")
report.write(f"Target segment                                     : {args.segid if args.segid else 'ALL'}\n\n")

report.write("--- HDBSCAN PARAMETERS ---\n")
report.write(f"Cluster selection method : {args.method}\n")
report.write(f"Auto-scaling                : {auto_status}\n")
report.write(f"min_cluster_size            : {dynamic_size}\n")
report.write(f"min_samples                  : {dynamic_samples}\n\n")

report.write("--- VISUALIZATION STATUS ---\n")
report.write(f"Conformational landscape : {plot_status.get('landscape', 'N/A')}\n")
report.write(f"FES & Connectivity       : {plot_status.get('fes_connectivity', 'N/A')}\n")
report.write(f"PCoA & Connectivity      : {plot_status.get('pcoa_connectivity', 'N/A')}\n\n")

report.write("--- STRUCTURAL PROCESSING ---\n")
report.write(f"RMSD structural filter   : {'Active' if args.rmsd > 0 else 'Inactive'}\n")
if args.rmsd > 0:
    report.write(f"RMSD threshold          : {args.rmsd} Å\n")
report.write("Unassigned interpretation: retained as unassigned/noise configurations\n\n")

report.write("--- RESULTS SUMMARY ---\n")
report.write(f"Clusters identified      : {len(unique_clusters)}\n")
report.write(f"Assigned frames          : {assigned_frames:,} ({(assigned_frames / total_frames) * 100:.2f}%) \n")
report.write(f"Unassigned/noise frames  : {noise_frames:,} ({(noise_frames / total_frames) * 100:.2f}%) \n\n")

report.write("--- SYSTEM & RUNTIME ---\n")
report.write(f"Logical CPU cores detected : {cpu_cores}\n")
report.write(f"Process RSS at report generation : {resident_ram_gb:.2f} GB\n")

```

```

report.write(f"Total pipeline runtime           : {format_duration(total_pipeline_time)}\n\n")

report.write("[ Measured stage runtimes ]\n")
report.write(f"  Data ingestion (I/O)           : {timing_stats.get('load', 0):7.2f} s\n")
report.write(f"  HDBSCAN clustering              : {timing_stats.get('hdbscan', 0):7.2f} s\n")
report.write(f"  Coordinate export              : {timing_stats.get('export', 0):7.2f} s\n")
report.write(f"  Visualization                  : {timing_stats.get('plots', 0):7.2f} s\n")
report.write(f"  carma structural process       : {timing_stats.get('extract', 0):7.2f} s\n")
report.write(f"      └─ Frame extraction        : {carma_stats.get('sort', 0):7.2f} s\n")
report.write(f"      └─ Alignment/RMSD         : {carma_stats.get('rmsd', 0):7.2f} s\n")
report.write(f"      └─ Superposition          : {carma_stats.get('super', 0):7.2f} s\n\n")

report.write("--- CLUSTER POPULATION & STRUCTURAL SUMMARY ---\n")
if args.rmsd > 0:
    report.write(f"{'Cluster':<8} {'HDBSCAN Frames':>15} {'Structural Frames':>18} {'Retained %':>12} {'Rep. Frame':>12} {'Max RMSD to avg (Å)':>21}\n")
    report.write("-" * 88 + "\n")
else:
    report.write(f"{'Cluster':<8} {'Frames':>10} {'Population':>12} {'Rep. Frame':>12} {'Max RMSD to avg (Å)':>21}\n")
    report.write("-" * 67 + "\n")

for cid in unique_clusters:
    count = np.sum(labels == cid)
    percentage = (count / total_frames) * 100
    rep_frame = cluster_reps.get(cid, "N/A")
    rmsd_str = f"{cluster_rmsds.get(cid, -1.0):.2f}" if cluster_rmsds.get(cid, -1.0) >= 0 else "N/A"
    if args.rmsd > 0:
        struct_frames = cluster_struct_counts.get(cid, count)
        ret_pct = (struct_frames / count) * 100
        report.write(f"{'cid':<8d} {'count':15,d} {'struct_frames':18,d} {'ret_pct':11.2f}% {'rep_frame':>12} {'rmsd_str':>21}\n")
    else:
        report.write(f"{'cid':<8d} {'count':10,d} {'percentage':11.2f}% {'rep_frame':>12} {'rmsd_str':>21}\n")
report.write("=====\n")

```

```

logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Detailed report saved to: {report_filename}")
logging.info("\n" + "="*70)

overall_fatal_error = fatal_frame_error or fatal_struct_error

if overall_fatal_error:
    logging.info(f"{Colors.BOLD} PARTIAL ANALYSIS COMPLETE{Colors.RESET}")
else:
    logging.info(f"{Colors.BOLD} ANALYSIS COMPLETE{Colors.RESET}")

logging.info("="*70)
logging.info(f"  Frames analyzed          : {total_frames:,}")
logging.info(f"  Clusters identified          : {len(unique_clusters)}")

if overall_fatal_error:
    logging.info(f"  Validation status           : FAILED (See report for details)")
else:
    logging.info(f"  Validation status           : PASSED")

logging.info(f"  Observed changes mapping : {'Not performed' if skip_label_changes else 'Completed'}")

# Summarize visualization completion from individual plot outcomes.
plot_values = list(plot_status.values())
if skip_label_changes:
    vis_status = "Partial (visualizations lacking temporal data)"
elif any("zero variance" in status.lower() for status in plot_values):
    vis_status = "Partial (skipped zero variance)"
elif any(status.startswith(("Skipped", "Not performed")) for status in plot_values):
    if plot_status.get('pcoa_connectivity', '').startswith('Skipped (requires >2 clusters)'):
        vis_status = "Partial (PCoA requires >2 clusters)"
    else:
        vis_status = "Partial"
else:
    vis_status = "Completed"

logging.info(f"  Visualizations              : {vis_status}")

```

```

if skip_structural:
    struct_status = "Not performed (duplicate frame IDs)"
elif carma_missing:
    struct_status = "Skipped (carma unavailable)"
elif not has_struct:
    struct_status = "Skipped (DCD/PSF unavailable)"
elif struct_info.get('compatible') is False:
    err_type = struct_info.get('preflight_error')
    if err_type == 'selection':
        struct_status = "Not performed (invalid atom selection)"
    elif err_type == 'missing_segment':
        struct_status = "Not performed (invalid segment selection)"
    elif err_type == 'missing_atoms':
        struct_status = "Not performed (missing atom identifiers)"
    elif err_type == 'mismatch':
        struct_status = "Not performed (DCD/PSF incompatible)"
    else:
        struct_status = "Not performed (structural preflight failed)"
elif struct_info.get('bounds_valid') is False:
    struct_status = "Not performed (frame IDs exceed trajectory bounds)"
else:
    success_count = struct_info.get('success_count', 0)
    failed_count = struct_info.get('failed_count', 0)
    if failed_count > 0:
        if success_count == 0:
            struct_status = f"Failed ({success_count}/{success_count+failed_count} successful)"
        else:
            struct_status = f"Partial ({success_count}/{success_count+failed_count} successful)"
    else:
        struct_status = "Completed"

logging.info(f"  Structural extraction      : {struct_status}")
logging.info(f"  Results directory      : {results_dir}/")
logging.info("=*70)

```

```

# =====
# COMMAND-LINE INTERFACE
# =====
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Automated HDBSCAN clustering, observed change mapping, and structural
extraction pipeline.")
    parser.add_argument('--size', type=int, default=500, help='HDBSCAN min_cluster_size (Default: 500)')
    parser.add_argument('--samples', type=int, default=50, help='HDBSCAN min_samples (Default: 50)')
    parser.add_argument('--auto', action='store_true', help='Set min_cluster_size and min_samples using the frame-count
heuristic')
    parser.add_argument('--cluster-only', action='store_true', help='Run ingestion and HDBSCAN clustering, export
assignments, then exit before downstream analysis.')
    parser.add_argument('--method', type=str, default='eom', choices=['eom', 'leaf'], help='HDBSCAN cluster-selection
method (Default: eom)')
    parser.add_argument('--dim', type=int, default=0, help='Number of projected dimensions to analyze; 0 uses all
available dimensions')
    parser.add_argument('--rmsd', type=float, default=0.0, help='Optional RMSD threshold for downstream structural
filtering in Angstrom; 0 disables filtering')
    parser.add_argument('--temp', type=float, default=298.0, help='Temperature in Kelvin used for population-derived free-
energy estimates (Default: 298.0)')
    parser.add_argument('--step', type=int, default=1, help='Frame stride applied during projection-file ingestion
(Default: 1)')
    parser.add_argument('--segid', type=str, default=None, help='Restrict structural extraction to the specified segment
identifier')
    parser.add_argument('--top', type=int, default=15, help="Number of observed changes to display in the diagnostic table
and plots (Default: 15. 0 disables displayed edges)")
    parser.add_argument('--transitions', '--changes', dest='transitions', nargs='*', type=int, help="Pass 0 args to view
the most frequent observed changes, or 2 cluster IDs to inspect a specific cluster pair.")
    parser.add_argument('--events', type=int, default=3, help="Maximum chronological change events to print in diagnostic
mode (Default: 3)")

    args = parser.parse_args()

    command_line = " ".join([shlex.quote(os.path.basename(sys.argv[0]))] + [shlex.quote(arg) for arg in sys.argv[1:]])

```



```

        sys.exit(1)

    if len(fluctuation_files) > 1:
        logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple fluctuation files found. Selecting the most
recently modified.")
        data_file = max(fluctuation_files, key=os.path.getmtime)

    logging.info(f"{Colors.BLUE}[+]{Colors.RESET} Input file: {data_file}")

    target_analysis_dir = os.path.dirname(data_file)
    if not target_analysis_dir: target_analysis_dir = '.'
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Analysis directory: {target_analysis_dir}")

    timestamp = datetime.datetime.now().strftime("%d.%m.%Y_%H.%M.%S.%f")[:-3]
    results_dir = f"clusterMD_results_{timestamp}"
    os.makedirs(results_dir, exist_ok=True)

    file_handler = logging.FileHandler(os.path.join(results_dir, "clustering_execution.log"))
    file_handler.name = "clusterMD_file"
    file_handler.setFormatter(PlainFileFormatter('%(message)s'))
    logging.getLogger().addHandler(file_handler)

    logging.info(f"Command line: {command_line}")
    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Output directory: {results_dir}")

    try:
        run_full_pipeline(args, data_file, results_dir, target_analysis_dir, command_line)
    except KeyboardInterrupt:
        logging.error(f"\n{Colors.RED}[-]{Colors.RESET} Process interrupted by user. Shutting down gracefully.")
        sys.exit(130)

else:
    logging.info(f"{Colors.BOLD}=== clusterMD | Post-Run Observed-Change Inspection ==={Colors.RESET}")
    dat_files = glob.glob("hdbscan.*-D.clusters.dat")

    if not dat_files:

```

```

        logging.error(f"\n{Colors.RED}[-]{Colors.RESET} Critical error: 'hdbscan.*-D.clusters.dat' not found in the
current directory.")
        logging.error(f"{Colors.YELLOW}[*]{Colors.RESET} You are running in diagnostic mode without full pipeline
flags.")
        logging.error(f"{Colors.YELLOW}[*]{Colors.RESET} Please navigate INSIDE a 'clusterMD_results_*' folder to run
this command.")
        sys.exit(1)

    if len(dat_files) > 1:
        logging.warning(f"{Colors.YELLOW}[WARN]{Colors.RESET} Multiple cluster dat files found. Selecting the most
recently modified.")
        dat_file = max(dat_files, key=os.path.getmtime)

    logging.info(f"{Colors.LIGHT_BLUE}[*]{Colors.RESET} Target data located: {os.path.abspath(dat_file)}")
    try:
        run_diagnostic_validation(args, dat_file)
    except KeyboardInterrupt:
        logging.error(f"\n{Colors.RED}[-]{Colors.RESET} Diagnostic run interrupted by user.")
        sys.exit(130)

```